Virtual Stationary Timed Automata for Mobile Networks

by

Tina Ann Nolte

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

© Massachusetts Institute of Technology 2009. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
October 24, 2008

Certified by . . . .
Nancy Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

Virtual Stationary Timed Automata for Mobile Networks

by

Tina Ann Nolte

Submitted to the Department of Electrical Engineering and Computer Science
on October 24, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis, we formally define a programming abstraction for mobile networks called the *Virtual Stationary Automata* programming layer, consisting of real mobile clients, virtual timed I/O automata called virtual stationary automata (VSAs), and a communication service connecting VSAs and client nodes. The VSAs are located at prespecified regions that tile the plane, defining a static virtual infrastructure. We present a theory of self-stabilizing emulation and use this theory to prove correct a self-stabilizing algorithm to emulate a timed VSA using the real mobile nodes that are currently residing in the VSA's region. We also specify two important services for mobile networks: motion coordination and end-to-end routing. We split the implementation of the end-to-end routing service into three smaller pieces, consisting of geographic routing and location management services with an end-to-end routing service built on top of them. We provide stabilizing implementations of each of these services using the VSA abstraction, and provide formal correctness analyses for each implementation.

Thesis Supervisor: Nancy Lynch
Title: NEC Professor of Software Science and Engineering

Acknowledgments

The years I spent at MIT were filled with interactions with people that I will never forget. Their support is what made this thesis possible.

First, I would like to thank my thesis advisor, Nancy Lynch. Of course, her technical guidance was necessary and invaluable, but, just as important, whenever I felt as though the light at the end of the tunnel was a little faint, a conversation with her would put me back on my feet. She was always able to help me remember that problems that I was thinking about were both interesting and of interest. I would never have been able to write this thesis without that.

I would also like to thank the other members of my thesis committee, Shlomi Dolev and Madhu Sudan, for their time and feedback.

Next, I would like to acknowledge the people that I had the honor of collaborating with on work that led up to this thesis. A visit by Shlomi Dolev and one of his students, Limor Lahiani, was the spark for this work. Shlomi is a bottomless well of interesting ideas and questions, and this initial visit created too many ideas to put into one paper. The first two VSA papers [29, 37] came from this visit.

Seth Gilbert was an indispensable sounding board, on the first VSA paper [29] and others [45], [12]. If I had difficulty figuring out how to explain something in my head, I would often talk to Seth. Somehow he could peer in and see what I was trying to say. We also had the opportunity to work together on some of the motion coordination work that appears in this thesis [45]. This motion coordination work was joint work with Sayan Mitra and was based on an earlier paper that Sayan and I worked on [66]. Working with Sayan on that earlier paper was the most fast-paced paper-writing experience I have ever had; it was a ton of fun, and the meal at his house when it was done was two tons of fun.

I also enjoyed working with Seth Gilbert and Calvin Newport (a well-read conversationalist who often came to my aid when nontechnical diversions were required) on a project with Matt Brown and Mike Spindel on implementing some of the virtual infrastructure ideas [12].

I had the pleasure of working with people outside the scope of my thesis as well. These

collaborators and friends include Mandana Vaziri, Ling Cheung, Rui Fan, Murat Demirbas, and Elad Schiller. I would also like to thank my master's thesis supervisor, Daniel Jackson, for helping me complete my first big technical work.

I need to thank my family and friends for their support. Without my family I would not be here, on planet Earth, let alone finishing an undertaking such as this. Without my friends, I would not have (most of) my sanity intact today. And without Stangelaliana and the Al/ Logan/ Nick/ Junior team in particular, I would probably have starved to death in my last months in Boston.

Finally, I have to particularly thank my husband, Jason, who always disagreed with me whenever I said that I was never going to finish, even if it was only because he wanted me to start working so that he could quit his day job and join the PGA tour (just you try it, buddy). Since Jason often complains that he has no records for the times when I admit that he was right and I was wrong, here it is:

"Jason, you were right, and I was wrong".

Contents

1 Introduction **15**

1.1 Mobile ad hoc networks . . . 16

1.1.1 Point-to-point based algorithms . . . 16

1.1.2 Location aware algorithms . . . 17

1.2 Virtual programming layers . . . 18

1.2.1 Virtual objects . . . 18

1.2.2 Virtual Mobile Nodes . . . 19

1.2.3 Our approach – Timed Virtual Stationary Automata . . . 19

1.3 Theory of stabilization and emulation for timed systems . . . 20

1.4 A stabilizing VSA layer emulation algorithm . . . 22

1.5 Thesis overview . . . 24

1.6 Research acknowledgments . . . 28

I Theory of stabilization and VSA layer emulation **30**

2 Mathematical preliminaries **33**

2.1 Basic mathematical preliminaries . . . 33

2.2 Timed I/O Automata . . . 34

3 Self-stabilization **41**

4 Emulations **51**

4.1 Emulation . . . 51

4.2 Emulation stabilization 53

5 Failure transform **57**

6 Layers: Physical layer model **61**

6.1 Network tiling 61

6.2 Mobile physical nodes 63

6.3 *RW*: Real World 64

6.4 *Pbcast*: Local broadcast service 69

6.4.1 Properties of *Pbcast* 70

6.4.2 Reachable states of *Pbcast* 72

6.4.3 Reachable states of $RW||Pbcast$ 74

6.5 P-algorithms and *PLayers* 78

7 Layers: Virtual Stationary Automata layer model **79**

7.1 Network tiling and *RW* 80

7.2 Virtual time and failer service *VW* 80

7.3 Mobile client nodes 81

7.4 Virtual Stationary Automata (VSAs) 82

7.5 *VBDelay* delay buffers 83

7.5.1 Client *VBDelay* 83

7.5.2 VSA *VBDelay* 85

7.6 *Vbcast*: Virtual local broadcast service 85

7.7 V-algorithms and *VLayers* 90

8 VSA layer emulations **93**

II VSA layer emulation algorithm 96

9 Totally ordered broadcast service **99**

9.1 *TOBspec*: Specification of totally ordered broadcast 99

9.1.1 *TObcast* 101

9.1.2 *TOBDelay* 105
9.1.3 *TOBFilter* 105
9.1.4 *TOBspec* 107
9.2 *TOBimpl*: Implementation 117
9.3 Correctness of the implementation 121
9.3.1 Legal sets 122
9.3.2 Simulation relation 133
9.3.3 Self-stabilization 142

10 Leader election service **147**
10.1 *LeadSpec*: Specification of the leader election service 147
10.1.1 *LeadMain* 149
10.1.2 *LeadCl* 151
10.1.3 *LeadSpec* 153
10.2 *LeadImpl*: Implementation 160
10.3 Correctness of the implementation 162
10.3.1 Legal sets 163
10.3.2 Simulation relation 165
10.3.3 Self-stabilization 172

11 Implementation of the VSA layer **175**
11.1 Client implementation 175
11.2 VSA implementation 175
11.3 Correctness of the implementation 182
11.3.1 Legal sets 186
11.3.2 Simulation relation 200
11.3.3 Self-stabilization 216
11.3.4 Stabilizing emulations 221
11.3.5 Message complexity 228
11.4 Extending the implementation to allow more failures 228

III VSA layer applications **230**

12 GeoCast **233**

12.1 Specification . . . 234

12.2 Properties of executions of the geocast service . . . 237

12.3 Legal sets . . . 239

12.3.1 Legal set L^1_{geo} . . . 239

12.3.2 Legal set L^2_{geo} . . . 241

12.4 Self-stabilization . . . 246

13 Location Management **251**

13.1 Location service specification . . . 253

13.1.1 Client algorithm . . . 255

13.1.2 VSA algorithm . . . 256

13.2 Properties of executions of the location service . . . 257

13.3 Legal sets . . . 259

13.3.1 Legal set L^1_{hls} . . . 260

13.3.2 Legal set L^2_{hls} . . . 261

13.3.3 Legal set L^3_{hls} . . . 262

13.3.4 Legal set L^4_{hls} . . . 263

13.3.5 Legal set L^5_{hls} . . . 264

13.4 Self-stabilization . . . 266

13.5 Extensions . . . 271

14 End-to-end Routing **273**

14.1 Client end-to-end routing specification . . . 273

14.1.1 Client algorithm . . . 274

14.1.2 VSA algorithm . . . 276

14.2 Properties of executions of the end-to-end routing service . . . 277

14.3 Legal sets . . . 280

14.3.1 Legal set L^1_{e2e} . . . 281

14.3.2 Legal set L_{e2e}^2 . . . 282
14.3.3 Legal set L_{e2e}^3 . . . 283
14.3.4 Legal set L_{e2e}^4 . . . 284
14.4 Self-stabilization . . . 285
14.5 Extensions . . . 290

15 Motion Coordination **293**
15.1 Background . . . 293
15.2 Motion Coordination using Virtual Nodes . . . 296
15.2.1 Problem Statement . . . 297
15.2.2 Overview of Solution using the VSA Layer . . . 297
15.2.3 *RW′*: modified *RW* . . . 298
15.2.4 *CN*: Client Node Algorithm . . . 299
15.2.5 *VN*: Virtual Stationary Node Algorithm . . . 300
15.2.6 *MC*: Complete System . . . 303
15.3 Correctness of the Algorithm . . . 304
15.3.1 Approximately Proportional Distribution . . . 305
15.3.2 Uniform Spacing . . . 315
15.4 Self-stabilization of the Algorithm . . . 317
15.4.1 Legal Sets . . . 318
15.4.2 Relationship between L_{MC} and reachable states . . . 320
15.4.3 Stabilization to L_{MC} . . . 326
15.5 Conclusion . . . 329

16 Conclusions **331**
16.1 Contributions . . . 331
16.2 Evaluation . . . 333
16.3 Open questions and avenues for research . . . 335

List of Figures

3-1 Execution α_B for Lemma 3.6. 43

6-1 P_p. 63

6-2 *RW*$[v_{max}, \epsilon_{sample}]$. 65

6-3 *RW* derived variables. 65

6-4 *Pbcast*$[d_{phys}, r_{real}]$. 69

7-1 Virtual Stationary Automata layer. VSAs and clients communicate locally using *Vbcast*. VSA and client outputs may be delayed in *VBDelay* buffers. *VW* provides timing and failure information to VSAs, and *RW* provides timing and mobile node location information. 79

7-2 *VW[*ϵ_{sample}*]*, Virtual time and failer service. 80

7-3 V_u. 82

7-4 *VBDelay*$_p$, Message delay service for clients. 83

7-5 *VBDelay[e]*$_u$, Message delay service for VSAs. 85

7-6 *Vbcast[d]*. 86

9-1 Totally ordered broadcast service. Client outputs may be delayed in *TOB-Delay* buffers, and messages are filtered out based on region and time alive information in *TOBFilter* buffers. *RW* provides timing and mobile node location information. 100

9-2 *TObcast[d]*, Message ordering service. 101

9-3 *TOBDelay*$_p$, Message delay service. 105

9-4 *TOBFilter[d]*$_p$, Message filtering service. 106

9-5 *TOBimpler*$_p$, providing ordered broadcast. 117

10-1 Leader election service. A *LeadCl* for a client performs a prefer$'(f)$ to indicate that its client should be considered by *LeadMain* as the leader of its client's region. *LeadMain* determines the winners of the leader competition for each region and communicates the results to each *LeadCl*. A winning process's *LeadCl* might then produce a leader output to its client, indicating the client is a leader. 148

10-2 $LeadMain$, electing a leader. 149

10-3 $LeadCl_p$, client portion for electing a leader. 151

10-4 Leader$_p$, electing a leader. 160

11-1 VSA layer implementation. Each process runs a collection of algorithms: $LeadCl, TOBDelay$, and $TOBFilter$, defined previously, together with $CE[alg]$ and $VSAE[alg]$, the client and VSA emulator algorithms. 176

11-2 VSAE$[alg]_p$, emulator at p of $alg \in VAlgs$. 179

11-3 Relationship between virtual and real time. A virtual clock behind real time runs faster until it catches up. 180

11-4 Functions for use in correctness proofs. 184

12-1 VSA geocast automaton at region u, V_u^{Geo}. 234

13-1 Client $C^{HL}[ttl_{hb}]_p$ periodically sends region updates to its local VSA. . . . 253

13-2 VSA $V^{HL}[ttl_{hb}, h : P \rightarrow U]_u$ automaton. 254

14-1 Client C_p^{E2E} automaton. 275

14-2 VSA $V^{E2E}[ttl_{hb}, h]_u$ automaton. 276

15-1 *RW*$'[v_{max}, \epsilon_{sample}]$. 299

15-2 Client node $CN(\delta)_p$ automaton. 300

15-3 $VN(\delta, k, \rho_1, \rho_2)_u$ TIOA, with parameters: safety k, and damping ρ_1, ρ_2. . . . 301

15-4 $VN(k, \rho_1, \rho_2)_u$ TIOA functions. 302

Chapter 1

Introduction

In this thesis, we focus on mobile ad-hoc networks, where mobile processors attempt to coordinate despite minimal infrastructure support. The task of designing algorithms for constantly changing networks is difficult. Highly dynamic networks, however, are becoming increasingly prevalent, especially in the context of pervasive and ubiquitous computing, and it is therefore important to develop and use techniques that simplify this task.

In addition, nodes in these networks are often unreliable, and may suffer from crashes or corruption faults, which cause arbitrary changes to their program states. Self-stabilization [26, 27] is the ability to recover from an arbitrarily corrupt state. This property is important in long-lived, chaotic systems where certain events can result in unpredictable faults. For example, transient interference may disrupt wireless communication, violating our assumptions about the broadcast medium.

In this thesis, I develop new techniques to cope with this dynamic, heterogeneous, and chaotic environment. We mask the unpredictable behavior of mobile networks by defining and emulating a stablizing *virtual* fixed infrastructure, called the *Virtual Stationary Automata layer*, consisting of *timing-aware* and *location-aware* machines at fixed locations, that mobile nodes can interact with. The static virtual infrastructure allows application developers to use simpler algorithms — including many previously developed for fixed networks. In order to facilitate the reasoning about this layer, in this thesis I also define a formal model for stabilization and stabilizing emulation in timed systems.

1.1 Mobile ad hoc networks

Mobile ad hoc networks are made up of mobile nodes (devices) that can be deployed in an ad hoc way over some deployment space. These networks can be made up of machines ranging from small sensors such as Berkeley motes [86], to cars, cell phones, and laptop computers. The nodes making up these networks are equipped with wireless communication, rather than access to a fixed "landline". They can provide communication or coordination services in situations where it is too costly or impractical to build a fixed infrastructure. Commonly cited examples of places where mobile ad hoc networks are especially useful are ones from search and rescue operations or battlefield coordination.

Direct communication in these networks is between devices that are close enough to each other to receive broadcasts. Since the devices are mobile and long distance communication requires multiple transmission hops, it can be difficult to ensure reliable communication between devices that are not within broadcast range. We assume for this thesis that mobile nodes enjoy reliable communication with other mobile nodes that are within a certain broadcast radius.

The machines making up these networks are also commonly fault-prone, since they are often small battery-powered devices, making them susceptible to crashes or sudden displacement. Also, their power constraints feed into constraints on broadcast power, implying the possibility of unexpected interruption or interference in communication. In this thesis, the mobile nodes are susceptible to crash failures and restarts, as well as corruption failures.

Increasingly, it is common for mobile devices to be equipped with access to a reasonably reliable time and location service that can provide devices with synchronized time updates and real-time location information. We assume in this thesis that mobile nodes have access to such an *oracle*.

1.1.1 Point-to-point based algorithms

Many of the initial algorithms for mobile ad hoc networks concentrated on achieving reliable point-to-point routing [56, 78, 79], one of the most important services in traditional

wired networks. This can be used to handle the dynamic nature of the networks by removing the concepts of geography and location from the consideration; a wireless network could be forced to appear as some wired network, oblivious of the location of its nodes.

Unfortunately, while such an approach might be sufficient if point-to-point communication is the only service that is desired, there are many circumstances where communication and coordination tied to actual geographic location is preferable in a mobile network. For example, in a traffic coordination scenario, safety would be best served by having cars near the same intersection coordinate with one another to avoid collision, rather than coordinate with the particular vehicles it has in their "car-phonebooks".

1.1.2 Location aware algorithms

In contrast to the point-to-point approach, there are a number of prior papers that take advantage of geography to facilitate the coordination of mobile nodes. For example, the GeoCast algorithms [14, 73], GOAFR [59], and algorithms for "routing on a curve" [72] route messages based on the location of the source and destination, using geography to delivery messages efficiently. Other papers [51,62,82] use geographic locations as a repository for data. These algorithms associate each piece of data with a region of the network and store the data at certain nodes in the region. This data can then be used for routing or other applications. All of these papers take a relatively ad-hoc or application-specific approach to using geography and location. We suggest a more systematic approach; we believe that the algorithms presented in these papers would benefit from using a fixed, predictable timing-enabled infrastructure to simplify coordination.

In industry there have been a number of attempts to provide specialized applications for ad-hoc networks by organizing some sort of virtual infrastructure over the mobile nodes. PacketHop and Motorola envision mobile devices cooperating to form mesh networks to provide communication services in areas with wireless-broadcast equipped devices but not a lot of fixed infrastructure [64]. These virtual infrastructures could allow on-the-fly network formation that can be used at disaster sites, or other areas where fixed infrastructure does not exist or has been destroyed. BMW and several other car manufacturers are de-

veloping systems that allow cars to communicate with one another about local road or car conditions, aiding in accident avoidance [87].

Another approach is that of Persistent Nodes [9]. Persistent nodes are virtual objects that move in a static sensor network, taking advantage of changing network conditions to try to ensure availability of data. While mobile, a persistent node only provides a non-atomic read/write object.

However, each of the above examples tackles very specific kinds of applications, like routing or distribution of sensor data. We believe a more general-purpose virtual infrastructure, that organizes mobile nodes into general programmable entities, can make a richer set of applications easier to provide. For example, with the advent of autonomous combat drones [85], the complexity of algorithms coordinating the drones can make it difficult to provide assurance to an understandably concerned public that these firepower-equipped autonomous units are coordinating properly. With a formal model of a flexible and easy-to-understand virtual infrastructure available, it would be easier to both provide and prove correct algorithms for performing sophisticated coordination tasks.

1.2 Virtual programming layers

In this thesis I describe a different approach to taming mobile ad hoc networks– virtual programming layers. Virtual programming layers do not provide a specific service; instead, they are a programming abstraction that allows application developers to design simpler algorithms for mobile networks. Several virtual programming layers have previously been proposed for mobile ad-hoc networks.

1.2.1 Virtual objects

The GeoQuorums algorithm [32] was the first to use virtual nodes; this algorithm defined a Focal Point Abstraction where mobile nodes in fixed, designated geographic regions of the network, called *focal points*, would cooperate to emulate atomic read/write shared memory. Atomicity is a strong property for a shared memory object that can be accessed concurrently by multiple processes; it requires that the invocations and responses of the object look as if

the object was only being accessed one at a time, and in an order consistent with the order of actual invocations and responses [65]. The focal points in the Focal Point abstraction were allowed to fail, but could not subsequently recover. This abstraction utilized a local broadcast service and a GeoCast communication service to facilitate communication between mobile clients and focal points. Quorums (sets) of focal points were then used in the paper to provide a fault-tolerant atomic memory service.

1.2.2 Virtual Mobile Nodes

More general virtual mobile automata were suggested in [31]. In this Virtual Mobile Node (VMN) abstraction, the virtual nodes are discrete I/O automata [65] that move on a predefined path through the network. The implementation of a VMN using the network's mobile nodes offered fault-tolerance through finite state replication among the mobile nodes emulating the VMN. A VMN is capable of recovery after failure, and utilizes just a local broadcast communication service to communicate with mobile clients, rather than both the local broadcast and GeoCast services used in the GeoQuorums work.

1.2.3 Our approach – Timed Virtual Stationary Automata

In Part I of this thesis, I present a new theoretical programming abstraction for mobile networks that consists of a static infrastructure of fixed, timed virtual machines with an explicit notion of real time, called *Virtual Stationary Automata* (VSAs), distributed at known locations over the plane, and emulated by the real mobile nodes in the system. Each VSA represents a predetermined geographic area and has broadcast capabilities similar to those of the mobile nodes, though perhaps suffering from an additional additive broadcast delay, allowing nearby VSAs and mobile nodes to communicate with one another. This programming layer provides mobile nodes with a virtual infrastructure with which to coordinate their actions. Many practical algorithms depend significantly on timing, and it is reasonable to assume that many mobile nodes have access to reasonably synchronized clocks. In the VSA programming layer, the virtual automata also have access to *virtual* clocks, guaranteed to not drift too far from real time.

VSAs are machines whose computational model is more powerful than those in [31], in that ours include timing capabilities, which are important for many applications. However, our automata are stationary, and are arranged in a connected pattern that is similar to a traditional wired network. This allows application developers to reuse a number of previously designed algorithms for stationary networks. Note that the virtual nodes described in [31, 32] could all be implemented using the infrastructure we describe here.

We present several applications in part III of this thesis whose implementations are significantly simplified by the VSA infrastructure. We consider both low-level services, such as routing and location management, as well as more sophisticated applications, such as motion coordination. The key idea in all cases is to locate data and computation at timed virtual automata throughout the network, thus relying on the fixed, predictable infrastructure to simplify coordination in ad-hoc networks. It is interesting to note that this infrastructure can be used to implement services such as routing that are oftentimes thought of as the lowest-level services in a network.

1.3 Theory of stabilization and emulation for timed systems

One contribution of this thesis is the formal modeling and analysis of the VSA programming layer, its implementation, and the implementations of various applications using the layer. In this thesis, we model systems using the timed I/O automata (TIOA) formalism [58]. These formal specification models provide unambiguous and simple descriptions of system behaviour and allow us to formally reason about system behaviour. Formal specifications also make clear those hidden system assumptions that can derail deployment of a distributed system.

As part of the project to formally model and analyze algorithms to provide the VSA programming layer, this thesis presents a formal semantics for emulation of a system. This provides proof obligations required to conclude that one system successfully emulates another system. An emulation is a kind of implementation relationship between two sets of

timed machines. Intuitively, one set of machines $\mathcal{B}$ emulates another set of machines $\mathcal{C}$ if each machine (program) C in $\mathcal{C}$ is mapped to a machine (emulation of the program) in $\mathcal{B}$ that has externally observable traces that look like some constrained set of externally observable traces of C.

Another significant contribution of this thesis is a theory of stabilization in TIOA systems, which we had to develop to explain the stabilization properties of our algorithm for emulating the VSA layer. Self-stabilization [26, 27] is the ability to recover from an arbitrarily corrupt state. This property is important in long-lived, chaotic systems where certain events can result in unpredictable faults. For example, transient interference may disrupt the wireless communication, violating our assumptions about the broadcast medium. This might result in inconsistency and corruption in the emulation of the VSA. Our self-stabilizing implementation of the VSA layer, however, can recover after corruptions to correctly emulate a VSA.

Prior work in self-stabilization for TIOA systems was informal. Our formal theory of stabilization in TIOA systems is based on *hybrid sequences*, sequences consisting of trajectories (modeling the evolution of a collection of variables over a time interval) interleaved with discrete actions. One set of hybrid sequences B is said to *stabilize in time* t to another set of hybrid sequences C if each suffix of β starting t time after the beginning of β happens to be in the set C. In this thesis, we demonstrate that these definitions work by concocting a "formula" that we use throughout the thesis for reasoning about the stabilization of an implementation of one system by another (described in the beginning of Section 9.3).

Our definition of stabilization using hybrid sequences is general enough to not only allow us to talk about executions (or traces) of one timed system stabilizing to executions (or traces) of another timed system, but also to talk about fragments of executions or traces starting in a certain set of states stabilizing to some set of fragments starting in another set of states. This generality is very useful in stabilization proofs for two reasons: (1) it makes it easy for us to break stabilization of an algorithm down into multiple phases, where one phase takes fragments starting in one set of states to fragments starting in a second set, another phase takes fragments starting in the second set to those in a third, etc.; (2) it makes it easy to talk about stabilization of algorithms with access to reliable external oracles; and

(3) it provides a way to talk about stabilization of algorithms for which there is no obvious "reset" state. By the last I mean that our definition of stabilization allows us to talk about stabilization of long-lived services with an invocation / response or send / receive behavior. In execution fragments of implementations of these services, it is possible for there to never be a point where there is no outstanding invocation or send. However, the implementation might be guaranteed to reach a point where it behaves just like some suffix of an execution of the service. Our definition of stabilization allows us to discuss these kinds of algorithms.

This thesis also presents a formal semantics for stabilizing emulation of timed systems. Since one part of this thesis is providing an emulation algorithm that implements a VSA layer but is also stabilizing, it is necessary to consider what such an emulation algorithm can guarantee. Hence, this thesis also presents a formal semantics for stabilizing emulation of timed systems. Say one set of machines $\mathcal{B}$ emulates another set of machines $\mathcal{C}$. We want to define the idea that for any program C in $\mathcal{C}$, the emulation of the program can be started in an arbitrary state but eventually produce externally observable behaviors that are related to those of C. What kind of behaviors of C should be the emulation produce? Intuitively, after a period where the emulation produces nonsense, it should manage to produce traces that look like traces of the program C, *though not necessarily starting from an initial or reachable state of that program*. Notice that this means that if corruption failures or arbitrary initial states are a possibility at emulators, then not only should the emulation algorithm be a stabilizing emulation, but the programs being emulated should be stabilizing.

These contributions are useful outside the scope of virtual nodes, potentially aiding in the specification of emulations of other systems or simplifying the reasoning about stabilizing timed systems in general.

1.4 A stabilizing VSA layer emulation algorithm

In part II of this thesis, I present an algorithm for implementing a VSA layer using a mobile ad hoc network consisting of mobile nodes that may suffer from crash failures and restarts. In order to provide this implementation, I first implement two other services over the mo-

bile nodes: totally ordered broadcast and leader election. Each mobile node is assumed to have access to a GPS service informing it of the region it is currently in. The totally ordered broadcast service ensures that processes in the same region receive the same messages in the same order. Under the assumption of reliable broadcast communication, this service is easily implemented using a technique similar to the one used in [61] to implement replicated state machines. The leader election service uses a round-based algorithm to periodically elect a new leader in each geographic area.

Given these two services, our clock-equipped VSA layer can then be emulated by the real mobile nodes in the network. A VSA for a particular geographic region is emulated by a subset of the mobile nodes currently populating its region: the VSA state is maintained in the memory of the real nodes emulating it, and the real nodes perform VSA actions on behalf of the VSA. If no mobile nodes are in the region, the VSA fails; if mobile nodes later arrive, the VSA restarts. The emulation is shared by the nodes while one node designated as leader is responsible for performing the outputs of the VSA and keeping the other nodes consistent in their VSA state.

An important property of our implementation is that it is self-stabilizing. Traditionally, studies of self-stabilizing systems are concerned with those systems that can be started from arbitrary configurations and eventually regain consistency *without external help*. However, mobile clients often have access to some reliable external information from a service such as GPS. Our algorithms use an external GPS service as a reliable *oracle*, providing periodic real time clock and location information to base stabilization upon. For example, our algorithms often use timestamps and location information to tag events. In an arbitrary state, recorded events may have corrupted timestamps. Corrupted timestamps indicating future times can be identified and reset to predefined values; new events receive newer timestamps than any in the arbitrary initial state. This could eventually allow nodes in the system to totally order events. We use the eventual total order to provide consistency of information and distinguish between incarnations of activity (such as retransmissions of messages).

1.5 Thesis overview

Here we provide an overview of the thesis. The thesis is divided into three main parts. The first part of the thesis focuses on introducing the theory of timed stabilization, stabilizing emulation, and VSA layers. The second part of the thesis focuses on a stabilizing emulation algorithm for the VSA layer. The third part of the thesis provides some examples of applications for the VSA layer.

Part I

The first part of the thesis provides the theoretical foundation for the rest of the thesis. It describes definitions and results for stabilization in timed systems, failures, stabilizing emulations, and the VSA layer.

I first provide some mathematical tools for talking about stabilization, system failure, and emulation in timed I/O automata systems. I also describe a system model for GPS-equipped mobile ad-hoc networks, and then describe a formal TIOA model of the VSA programming layer.

Chapter 2

I begin by reviewing the Timed I/O Automata model of [58] for discussing timed systems.

Chapter 3

I then provide some mathematical definitions and tools for talking about stabilization in timed systems. The definition of stabilization for timed systems is based on hybrid sequences; we define stabilization as being from one set of hybrid sequences to another. In this chapter I also show some useful results about stabilization, including results about transitivity and *legal sets*. Legal sets are sets of states that are closed under execution fragments; they are used often in this thesis to describe states with desirable properties.

Chapter 4

In this chapter, I define the concept of an emulation and a stabilizing emulation. Emulations define a kind of implementation relationship between two sets of machines. The definition of emulation is followed with a definition of a stabilizing emulation. An algorithm is a stabilizing emulation if an emulation of a program can be started in an arbitrary state but eventually behave as though it is the emulated program, though from a potentially arbitrary state.

Chapter 5

This chapter discusses a model for node failure and restart. It describes a general crash stop and restart transformation for TIOA programs. Such a transformation is useful in that it removes ambiguity about the semantics of failures and restarts.

Chapter 6

Here I introduce a model of a mobile ad hoc network physical layer. This layer consists of mobile physical nodes, a GPS oracle, and a broadcast communication service. Communication is local in this model.

Chapter 7

In this chapter, I consider the Virtual Stationary Automata layer model. The VSA layer consists of mobile client nodes (analogous to mobile physical nodes), a GPS oracle, a virtual broadcast communication service, a virtual time service (a GPS time service for the Virtual Stationary Automata), and Virtual Stationary Automata. A VSA is a TIOA with a real-time clock, restricted external interface, allowing it to only send and receive messages using the virtual broadcast service; its broadcasts can be delayed for up to a constant amount of time.

Chapter 8

Here the results and definitions of Chapter 4 are specialized for the case of an emulation of the VSA layer by the physical layer.

Part II

We provide an implementation of that layer using the underlying mobile ad-hoc system, and prove that the implementation provides a stabilizing emulation of the VSA programming layer. This implementation is in three parts: totally ordered broadcast, leader election, and a main emulation component.

Chapter 9

It is useful to have access to a totally ordered broadcast service that allows nodes in the same region to receive the same sets of messages in the same order. The totally ordered broadcast service is intended to allow a non-failed node p that knows it is in some region u to broadcast a message m, via $\mathsf{tocast}(m)_p$, and to have the message be received exactly $d, d > d_{phys}$, time later via $\mathsf{torcv}(m)_q$, by nodes that are in region u or a neighboring region for at least d time.

Chapter 10

It is also useful to have access to a leader election service that allows nodes in the same region to periodically compete to be named sole leader of the region for some time. Our leader election service is a round-based service that collects information from potential leaders at the beginning of each round, determines up to one leader per region, and performs leader outputs for those leaders that remain alive and in their region for long enough.

Chapter 11

We describe a fault-tolerant implementation of a VSA by mobile nodes in its region of the network. At a high level, the individual mobile nodes in a region share emulation of the virtual machine through a deterministic state replication algorithm while also being

coordinated by a leader. Each mobile node runs its portion of the totally ordered broadcast service, leader election service, and a Virtual Node Emulation ($VSAE$) algorithm, for each virtual node.

In this chapter we also prove that the implementation is a stabilizing emulation of the VSA layer.

Part III

We conclude with a description of two applications that we implement using the VSA layer. In the thesis, each implementation, whether of the VSA programming layer or of applications built on the layer, is proved correct using the TIOA formal framework.

The first VSA application, end-to-end routing, is implemented in three pieces: a region-to-region geocast service, a location management service, and an end-to-end routing service built on the geocast and location management services. The second application is a motion coordination service.

Chapter 12

We describe a stabilizing region-to-region communication service in this chapter. The algorithm is based on a shortest path procedure. When a region receives a geocast message it has not previously seen from region u to region v for which it is on a shortest path from u to v, it forwards the message closer to region v. The program described in this chapter is a part of a VSA layer program to provide end-to-end routing.

Chapter 13

This chapter describes how to provide the location management piece of the end-to-end routing service on the VSA layer. The solution is based on the concept of *home location servers*, where each mobile client identifier hashes to a home location, a region of the network that is periodically updated with the location of the client and that is responsible for answering queries about the client's location. The periodic location updates and the forwarding of queries and responses are done using the geocast service of Chapter 12.

Chapter 14

We describe a simple self-stabilizing algorithm over the VSA layer to provide a mobile client end-to-end routing service. A client sends a message to another client by forwarding the message to its local VSA, which then uses the home location service to discover the destination client's region and forwards the message to that region using the geocast service.

Chapter 15

In this chapter, we study how the VSA layer can help us solve the problem of coordinating the behavior of a set of autonomous mobile robots (physical nodes) in the presence of changes in the underlying communication network as well as changes in the set of participating robots. Each VSA must decide based on its own local information which robots to keep in its own region, and which to assign to neighboring regions; for each robot that remains, the VSA determines where on the curve the robot should reside. Unlike in the prior three applications (Geocast, location management, and end-to-end communication), the client motion in the motion coordination protocol is controllable by the client, allowing the client to change its motion trajectory based on instructions from a VSA.

1.6 Research acknowledgments

Much of the research presented in this thesis has been done in collaboration with others, particularly: Shlomi Dolev, Seth Gilbert, Limor Lahiani, Nancy Lynch, and Sayan Mitra. The content in this thesis has been partially drawn from the following papers:

- *Self-stabilization and Virtual Node Layer Emulations* [75]. This paper is a preliminary version of some of the results in Chapters 3, 4, and 8. In it I introduced a set of formal definitions for stabilization in timed systems, as well as a formal definition of stabilizing emulation for the VSA layer. However, the definitions in this thesis are different; the thesis generalizes some of the stabilization and emulation results, and introduces a new formal model for process failures and restarts.

- *Timed Virtual Stationary Automata for Mobile Networks* [29, 30]. These papers presented preliminary models of the physical layer described in Chapter 6 and the VSA layer described in Chapter 7, though the failure modeling in these papers difffers from the modeling in this thesis.

  The initial impetus for these papers came from work with Shlomi Dolev and Limor Lahiani about their ideas on how mobile nodes in predefined geographic regions could share responsibility for implementing a message routing service. These papers were joint work in which we generalized these ideas into an implementation of an early version of a VSA layer, where mobile nodes in predefined geographic regions could share responsibility for general emulation of algorithms.

- *Self-Stabilizing Mobile Node Location Management and Message Routing* [37]. This paper contains early versions of algorithms for implementing geographic broadcast, location management, and message routing services using the VSA layer. It is based on some of the ideas from the same work with Shlomi Dolev and Limor Lahiani mentioned above, and was the first paper demonstrating applications of the VSA layer.

  In Chapters 12-14 of this thesis I use a different set of VSA layer algorithms to implement versions of these services. However, the breakdown in [37] of the message routing problem into three pieces is preserved in this thesis.

- *Self-Stabilizing Mobile Robot Formations with Virtual Nodes* [45]. This paper is a preliminary version of Chapter 15 in this thesis. It is itself based on work in [66], where a simplified virtual node layer was used to coordinate the motion of mobile nodes. The technical definition of the problem of motion coordination (a variant of which appears in this thesis in Section 15.2) and the rules used by the virtual nodes for allocating / directing mobile nodes (Figure 15-4 of this thesis) are primarily the work of Sayan Mitra, as is the proof that these rules lead to motion coordination (reproduced in this thesis in Section 15.3). My contribution in this work is in the VSA layer modeling of the algorithm, as well as the design and proof of stabilization of a self-stabilizing version of the algorithm.

Part I

Theory of stabilization and VSA layer emulation

In Part I of this thesis, I introduce the theory that the rest of this thesis is built on. I open with a brief review in Chapter 2 of the Timed I/O Automata model of [58] for discussing timed systems.

In Chapter 3, I then provide some mathematical definitions and tools for talking about stabilization in timed systems. The definition of stabilization for timed systems is based on hybrid sequences; we define stabilization as being from one set of hybrid sequences to another. In this chapter I also show some useful results about stabilization, including results about transitivity and *legal sets*. Legal sets are sets of states that are closed under execution fragments; they are used often in this thesis to describe states with desirable properties.

Next, in Chapter 4, I define the concepts of an emulation and a stabilizing emulation. An emulation defines a kind of implementation relationship between two sets of machines. The definition of emulation is followed with a definition of a stabilizing emulation. An algorithm is a stabilizing emulation if an emulation of a program can be started in an arbitrary state but eventually behave as though it is the emulated program, though from a potentially arbitrary state.

Chapter 5 is where I discuss a model for node failure and restart. It describes a general crash stop and restart transformation for TIOA programs. Such a transformation is useful in that it removes ambiguity about the semantics of failures and restarts.

In Chapter 6, I introduce a model of a mobile ad hoc network physical layer. This layer consists of mobile physical nodes, a GPS oracle, and a broadcast communication service. Communication is local in this model.

Chapter 7 is where I describe the Virtual Stationary Automata layer model. The VSA layer consists of mobile client nodes (analogous to mobile physical nodes), a GPS oracle, a virtual broadcast communication service, a virtual time service (a GPS time service for the Virtual Stationary Automata), and Virtual Stationary Automata. A VSA is a TIOA with a real-time clock, restricted external interface, allowing it to only send and receive messages using the virtual broadcast service; its broadcasts can be delayed for up to a constant amount of time.

Finally, Chapter 8 is where the emulation results and definitions of Chapter 4 are specialized for the case of an emulation of the VSA layer by the physical layer.

Chapter 2

Mathematical preliminaries

Here we introduce some terminology and notation for expressing mathematical properties in this thesis.

2.1 Basic mathematical preliminaries

If f is a function, we refer to the domain and range of f as $domain(f)$ and $range(f)$ respectively. If S is a set, we can restrict f to S, written $f \lceil S$, defined to be the function with domain equal to $S \cap domain(f)$ where for each c in its domain, it maps to $f(c)$. If f is a function mapping to a set of functions and S is a set, then $f \downarrow S$ is the function with domain equal to $domain(f)$ and such that for each c in its domain, it maps to $f(c) \lceil S$.

If S is a set, then a sequence σ over S is a function with a domain either equal to the set of all positive integers or the set $\{1, \cdots, k\}$ for some positive integer k, and with a range equal to S. We use $|\sigma|$ to be the cardinality of $domain(\sigma)$. The set of finite sequences over S are denoted by S^*. The empty sequence is denoted by λ. The concatenation of two sequences σ and σ' is written $\sigma\sigma'$. We say that σ is a prefix of σ', written $\sigma \leq \sigma'$, if either $\sigma = \sigma'$ or σ is finite and $\sigma' = \sigma\rho$ for some sequence ρ. If σ is a nonempty sequence, then $head(\sigma)$ refers to the first element of σ and $tail(\sigma)$ refers to σ with its first element removed. $insert(\sigma, s, i)$, for $s \in S$ and $0 \leq i \leq |\sigma|$ is a new sequence equal to σ, except with element s inserted after the element at position i.

2.2 Timed I/O Automata

Here we describe Timed I/O Automata (TIOA) terminology used in this thesis. TIOAs are nondeterministic state machines whose state can change in two ways: instantaneously through a discrete transition, or according to a trajectory describing the evolution, possibly continuous, of variables over time. The TIOA framework can be used to carefully specify and analyse timed systems. (Additional details can be found in [58].) Each algorithm and specification in this thesis is expressed using this framework.

The type of a variable describes the values that a variable can take on, while the dynamic type of a variable describes how a variable's values can change over time.

Definition 2.1 *For each variable v we have the following:*

- $type(v)$, *the* static type *of v, is a nonempty set of values.*
- $dtype(v)$, *the* dynamic type *of v, is a set of functions from left-closed intervals of time starting at 0 to $type(v)$ satisfying the following:*
  - *For each $f \in dtype(v)$ and $t \in \mathbb{R}$, f shifted forward by t time is also in $dtype(v)$.*
  - *For each $f \in dtype(v)$ and each left-closed subinterval J of $domain(f)$, $f \lceil J \in dtype(v)$.*
  - *Consider any sequence of functions $f_0, f_1, \cdots$, each in $dtype(v)$ such that for each f_i except the last, the domain of f_i is right-closed and $max(domain(f_i)) = min(domain(f_{i+1}))$. Then the function f, defined so that $f(t) = f_i(t)$ where i is the minimum index such that $t \in domain(f_i)$, is in $dtype(v)$.*

Variable v is constant *over a left-closed interval of time if its mapping to a value is constant over that interval. Variable v is a* discrete *variable if for every left-closed interval of time, v is constant over the interval.*

Definition 2.2 *A* valuation *for a set V of variables is a function mapping each variable $v \in V$ to a value in $type(v)$. The set of such valuations is $val(V)$.*

A trajectory models the evolution of a collection of variables over a time interval.

Definition 2.3 *A* trajectory*, τ, for V is a function mapping a left-closed interval of time starting at 0 to the set of valuations for V, such that for $v \in V$, τ restricted to v is in the dynamic type of v.*

- *τ is* closed *if $domain(\tau)$ is both left and right-closed.*
- *$\tau.fstate$ is the first valuation of τ, and, for τ closed, $\tau.lstate$ is the last.*
- *The limit time of τ, $\tau.ltime$, is the supremum of $domain(\tau)$.*
- *The concatenation, $\tau\tau'$, of trajectories τ and τ', τ closed, is the trajectory resulting from the pasting of τ', shifted by $\tau.ltime$, to the end of τ. The valuation at $\tau.ltime$ is the one in $\tau\tau'$, overwriting the value of $\tau'.ftime$.*
- *A trajectory for V with a domain equal to the point 0 is called a* point *trajectory for V. If v is a valuation for V, then $\wp(v)$ is the point trajectory for V mapping to v.*

A timed I/O automaton is a state machine with some set of variables describing its state. It also has a set of discrete actions, some internal, some external inputs and some external outputs. Its state can change either through discrete transitions, which result in atomic state changes, or through trajectories, which describe the evolution of the state variables over the time when discrete transitions do not occur.

Definition 2.4 *A* Timed I/O Automaton (TIOA)*, $\mathcal{A} = (X, Q, \Theta, I, O, H, \mathcal{D}, \mathcal{T})$, consists of:*

- *Set X of internal variables.*
- *Set $Q \subseteq val(X)$ of states.*
- *Set $\Theta \subseteq Q$ of start states, nonempty.*
- *Sets I of input actions, O of output actions, and H of internal actions, each disjoint. $A = I \cup O \cup H$ is all actions. $E = I \cup O$ is all external actions.*

- *Set $\mathcal{D} \subseteq Q \times A \times Q$ of discrete transitions.*

  We say action a is enabled in state x if $(x, a, x') \in \mathcal{D}$, for some $x' \in X$. We require $\mathcal{A}$ be input-enabled (every input action is enabled at every state).

- *Set $\mathcal{T} \subseteq$ trajectories of Q. We require:*

  - *For every state x, the point trajectory for x must be in $\mathcal{T}$,*
  - *For every $\tau \in \mathcal{T}$, every prefix and suffix of τ is in $\mathcal{T}$,*
  - *For every sequence of trajectories in $\mathcal{T}$, where for every τ_i but the last, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$, the concatenation of the trajectory sequence is also in $\mathcal{T}$, and*
  - *Time-passage enabling: for every state x, there exists a $\tau \in \mathcal{T}$ where $\tau.fstate = x$, and either $\tau.ltime = \infty$ or τ is closed and some $l \in H \cup O$ is enabled in $\tau.lstate$.*

Definition 2.5 *Two TIOAs $\mathcal{A}$ and $\mathcal{B}$ are* compatible *if they share no internal variables, and their internal actions are not actions of the other.*

Composition, described in the following definition, is useful for describing the behaviour of complex systems. It allows us to describe the system as a collection of separate components that can then be run together after composition.

Definition 2.6 *Two compatible TIOAs $\mathcal{A}$ and $\mathcal{B}$ can be* composed *into a new TIOA $\mathcal{A}\|\mathcal{B}$, which has $\mathcal{A}$ and $\mathcal{B}$ as components where an action performed in one component that is an external action of the other component is also performed in the other component. Each external action of the composition is an output if it is an output of one of the component automata, and an input otherwise. Each internal action of the individual automata remains an internal action.*

The following definition allows us to perform output action hiding on TIOAs, reclassifying a designated set of output actions as internal actions. This is especially useful when we later consider implementation relationships, where we require that the sets of external actions for machines be the same.

Definition 2.7 *Let $\mathcal{A}$ be a TIOA and O be a subset of $O_{\mathcal{A}}$. Then* $\mathsf{ActHide}(O, \mathcal{A})$ *is a TIOA equal to $\mathcal{A}$ except that $O_{\mathsf{ActHide}(O,\mathcal{A})} = O_{\mathcal{A}} - O$ and $H_{\mathsf{ActHide}(O,\mathcal{A})} = H_{\mathcal{A}} \cup O$.*

Hybrid sequences are described in the next definition. These sequences are often used to describe an execution or a trace (observable behaviour) of a TIOA.

Definition 2.8 *Given a set A of actions and a set V of variables, an (A, V)*-sequence *is an alternating sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \cdots$ where: (a) Each a_i is an action in A, (b) Each τ_i is a trajectory for V, (c) If α is finite, it ends with a trajectory, and (d) Each τ_i but the last is closed.*

- *α is* closed *if it is a finite sequence and its final trajectory is closed.*
- *The limit time of α, $\alpha.ltime$, is the sum of limit times of α's trajectories.*
- *The concatenation, $\alpha\alpha'$, of two (A, V)-sequences α and α', α closed, is α followed by α', where the last trajectory of α is concatenated to the first trajectory of α'.*
- *For sets of actions A and A', and sets of variables V and V', the (A', V')*-restriction *of an (A, V)-sequence α, written $\alpha\lceil(A', V')$, is the sequence that results from projecting the trajectories of α on variables in V', removing actions not in A', and concatenating all adjacent trajectories.*

In the following definition, an execution fragment of a TIOA is defined to be a hybrid sequence where each trajectory is a trajectory of the TIOA and for each action in the sequence, the last state of the trajectory preceding it satisfies the precondition of the action, and the first state of the trajectory following it is the state that should result from that discrete transition.

Definition 2.9 *An* execution fragment *of a TIOA $\mathcal{A}$ is an (A, V)-sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \cdots$, where each τ_i is a trajectory in $\mathcal{T}$, and if τ_i is not the last trajectory of α, then $(\tau_i.lstate, a_{i+1}, \tau_{i+1}.fstate) \in \mathcal{D}$. We refer to the set of execution fragments of $\mathcal{A}$ starting from a state in some $S \subseteq Q$ as $frags^S_{\mathcal{A}}$.*

The following definition of an execution just says that an execution is any execution fragment where the very first state of the first trajectory of the fragment is an initial state of the TIOA.

Definition 2.10 *An execution fragment of $\mathcal{A}$, α, is an* execution *of $\mathcal{A}$ if $\alpha.fstate$ is in Θ. We refer to the set of executions of $\mathcal{A}$ as $execs_{\mathcal{A}}$.*

Definition 2.11 *A state of $\mathcal{A}$ is* reachable *if it is the last state of some closed execution of $\mathcal{A}$. We refer to the set of reachable states of $\mathcal{A}$ as $reachable_{\mathcal{A}}$.*

Definition 2.12 *An* invariant *for $\mathcal{A}$ is a property that is true for all reachable states of $\mathcal{A}$.*

A trace, defined below, is the external observable behaviour of a TIOA. The only information it imparts is the length of an execution, whether or not the execution is right-closed, and the timing and order of the external actions of the TIOA in that execution.

Definition 2.13 *A* trace *(external behaviour) of an execution fragment α of $\mathcal{A}$, $trace(\alpha)$, is α restricted to external actions of $\mathcal{A}$ and trajectories over the empty set of variables. $traces_{\mathcal{A}}$ is the set of traces of executions of $\mathcal{A}$. We refer to the set of traces of execution fragments of $\mathcal{A}$ starting from a state in some $S \subseteq Q$ as $tracefrags^{S}_{\mathcal{A}}$.*

The next lemma (Lemma 5.2 in [58]) says that execution fragments of composed TIOAs project to fragments of the components:

Lemma 2.14 *Let $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$ and let α be an execution fragment of $\mathcal{A}$.*
Then $\alpha \lceil (A_1, X_1)$ is an execution fragment of $\mathcal{A}_1$, and $\alpha \lceil (A_2, X_2)$ is an execution fragment of $\mathcal{A}_2$. Also, α is an execution iff both $\alpha \lceil (A_1, X_1)$ and $\alpha \lceil (A_2, X_2)$ are executions.

The next theorem (Theorem 7.3 in [58]) says that traces of composed TIOAs are exactly those empty-variable hybrid sequences whose restrictions to the external actions of component TIOAs are traces of the components:

Theorem 2.15 *Let $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$.*
Then $traces_{\mathcal{A}} = \{\beta \mid \beta$ is an $(E, \emptyset)$-sequence and $\beta \lceil (E_i, \emptyset) \in traces_{\mathcal{A}_i}, i \in \{1, 2\}\}$.

The following two results on execution pasting are from [44]. Say that we are given two compatible TIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$, and executions α_1 and α_2 of $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. The first result says that if there is a hybrid sequence β with the same type as a trace of $\mathcal{A}_1\|\mathcal{A}_2$ and such that β is consistent with the traces of executions α_1 and α_2 in that β restricted to external actions of $\mathcal{A}_1$ is equal to the trace of α_1 (and similarly for $\mathcal{A}_2$), then we can paste together the executions α_1 and α_2 to get an execution of $\mathcal{A}_1\|\mathcal{A}_2$ whose trace is equal to β. The second result is just a generalization for a finite number of machines.

Lemma 2.16 *Let $\mathcal{A} = \mathcal{A}_1\|\mathcal{A}_2$, and let α_1 and α_2 be executions of $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. Let β be an $(E_{\mathcal{A}}, \emptyset)$ sequence such that $\beta\lceil(E_{\mathcal{A}_1}, \emptyset) = trace(\alpha_1)$ and $\beta\lceil(E_{\mathcal{A}_2}, \emptyset) = trace(\alpha_2)$. Then there exists an execution α of $\mathcal{A}$ such that $\alpha_1 = \alpha\lceil(A_1, X_1)$, $\alpha_2 = \alpha\lceil(A_2, X_2)$, and $trace(\alpha) = \beta$.*

Corollary 2.17 *Let $\mathcal{A} = \mathcal{A}_1\|\mathcal{A}_2\| \ldots \|\mathcal{A}_k$ for some finite k, and let α_i be an execution of $\mathcal{A}_i$ for every i. Let β be an $(E_{\mathcal{A}}, \emptyset)$ sequence such that $\beta\lceil(E_{\mathcal{A}_i}, \emptyset) = trace(\alpha_i)$ for each $i \in \{1, \ldots, k\}$. Then there exists an execution α of $\mathcal{A}$ such that $trace(\alpha) = \beta$ and $\alpha_i = \alpha\lceil(A_i, X_i)$, for each $i \in \{1, \ldots, k\}$.*

The following definitions describe the concept of one TIOA implementing another. The intuition is that $\mathcal{A}$ implements $\mathcal{B}$ if for each execution of $\mathcal{A}$, its externally visible behaviour happens to be the same as the externally visible behaviour of $\mathcal{B}$.

Definition 2.18 *Two TIOAs $\mathcal{A}$ and $\mathcal{B}$ are* comparable *if they have the same external interface.*

Definition 2.19 *If $\mathcal{A}$ and $\mathcal{B}$ are comparable, then we say that $\mathcal{A}$* implements *$\mathcal{B}$, written $\mathcal{A} \leq \mathcal{B}$, if $traces_{\mathcal{A}} \subseteq traces_{\mathcal{B}}$.*

The next definition describes properties of a special kind of relation that is useful for showing that one TIOA implements another.

Definition 2.20 *Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TIOAs. A* forward simulation *from $\mathcal{A}$ to $\mathcal{B}$ is a relation $R \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ satisfying the following for all states $x_{\mathcal{A}}$ and $x_{\mathcal{B}}$ of $\mathcal{A}$ and $\mathcal{B}$ respectively:*

1. *If* $x_{\mathcal{A}} \in \Theta_{\mathcal{A}}$ *then there exists a state* $x_{\mathcal{B}} \in \Theta_{\mathcal{B}}$ *such that* $x_{\mathcal{A}} R x_{\mathcal{B}}$.

2. *If* $x_{\mathcal{A}} R x_{\mathcal{B}}$ *and* α *is an execution fragment of* $\mathcal{A}$ *consisting of one action surrounded by two point trajectories, with* $\alpha.fstate = x_{\mathcal{A}}$*, then* $\mathcal{B}$ *has a closed execution fragment* β *with* $\beta.fstate = x_{\mathcal{B}}, trace(\beta) = trace(\alpha)$*, and* $\alpha.lstate R \beta.lstate$.

3. *If* $x_{\mathcal{A}} R x_{\mathcal{B}}$ *and* α *is an execution fragment of* $\mathcal{A}$ *consisting of a single closed trajectory, with* $\alpha.fstate = x_{\mathcal{A}}$*, then* $\mathcal{B}$ *has a closed execution fragment* β *with* $\beta.fstate = x_{\mathcal{B}}, trace(\beta) = trace(\alpha)$*, and* $\alpha.lstate R \beta.lstate$.

A useful theorem, shown in [58], is that if there is a forward simulation from machine $\mathcal{A}$ to $\mathcal{B}$ then the trace of an execution fragment of $\mathcal{A}$ starting in some state related via the simulation relation to a state in $\mathcal{B}$ is a trace of an execution fragment of $\mathcal{B}$ starting in the related state:

Theorem 2.21 *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be comparable TIOAs and let* R *be a forward simulation relation from* $\mathcal{A}$ *to* $\mathcal{B}$*. Let* $x_{\mathcal{A}}$ *and* $x_{\mathcal{B}}$ *be states of* $\mathcal{A}$ *and* $\mathcal{B}$*, respectively, such that* $x_{\mathcal{A}} R x_{\mathcal{B}}$*. Then* $tracefrags_{\mathcal{A}}(x_{\mathcal{A}}) \subseteq tracefrags_{\mathcal{B}}(x_{\mathcal{B}})$.

One immediate corollary is the following, which extends the above result to sets of states the execution fragments may start in:

Corollary 2.22 *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be comparable TIOAs,* R *be a simulation relation from* $\mathcal{A}$ *to* $\mathcal{B}$*,* $L_{\mathcal{A}}$ *be a subset of states of* $\mathcal{A}$*, and* $L_{\mathcal{B}}$ *be a subset of states of* $\mathcal{B}$*. Suppose that for each* $x \in L_{\mathcal{A}}$ *there exists some* $y \in L_{\mathcal{B}}$ *such that* xRy*. Then* $tracefrags_{\mathcal{A}}^{L_{\mathcal{A}}} \subseteq tracefrags_{\mathcal{B}}^{L_{\mathcal{B}}}$.

Another useful corollary of Theorem 2.21, shown in [58], is the following, which says that if a forward simulation relation from one machine to a comparable machine exists, then the first machine implements the second:

Corollary 2.23 *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be comparable TIOAs and let* R *be a forward simulation relation from* $\mathcal{A}$ *to* $\mathcal{B}$*. Then* $\mathcal{A} \leq \mathcal{B}$.

Chapter 3

Self-stabilization

We define stabilization in terms of sets of (A, V)-sequences. This is general enough to talk about stabilization of traces and execution fragments of TIOAs, and about stabilization of transformed versions of these (A, V)-sequences.

First we define the concept of a t-suffix of a hybrid sequence α. This is just a suffix of α such that its corresponding prefix has a limit time of t.

Definition 3.1 *Let α and α' be (A, V)-sequences, and t be a non-negative real. α' is a* t-suffix of α *if a closed (A, V)-sequence α'' exists where $\alpha''.ltime = t$ and $\alpha = \alpha''\alpha'$.*

By the definition of concatenation for hybrid sequences and trajectories, if sequences α'' and α' are concatenated to produce sequence α, the first state of α' is overwritten by the last state of α'' in the concatenation. This means that any sequence that begins with some arbitrary value of the variables of α but otherwise equals α' could also be concatenated to α'' to get α. In the following definition, we define a *state-matched t-suffix* to be a t-suffix with the additional constraint that its first state happens to match the last state of its associated prefix.

Definition 3.2 *Let $\alpha = \alpha''\alpha'$ be an (A, V)-sequence and t be a non-negative real. α' is a* state-matched t-suffix of α *if it is a t-suffix of α, and $\alpha'.fstate = \alpha''.lstate$.*

As long as an (A, V)-sequence either has a limit time greater than some t or is closed with a limit time equal to t, we know that a state-matched t-suffix of the sequence exists.

Lemma 3.3 *Let α be an (A, V)-sequence and t be a non-negative real where either $t < \alpha.ltime$, or $t = \alpha.ltime$ and α is closed. A state-matched t-suffix of α exists.*

Proof: In the case where $t = \alpha.ltime$, the point trajectory $\wp(\alpha.lstate)$ is a state-matched t-suffix of α; $\alpha = \alpha\wp(\alpha.lstate)$.

If $t < \alpha.ltime$, then consider any closed prefix α' of α such that $\alpha'.ltime = t$. By Lemma 3.5 in [58], there exists some α'' such that $\alpha = \alpha'\alpha''$. Consider any such α'' with $\alpha''.fstate$ changed to $\alpha'.lstate$. This modified α'' is a state-matched t-suffix of α. ■

Definition 3.4 *Let B be a set of (A^B, V)-sequences and C be a set of (A^C, V)-sequences, where A^B and A^C are sets of actions and V is a set of variables. Let t be in $\mathbb{R}^{\geq 0}$.*
B stabilizes in time t to C *if each state-matched t-suffix α of each sequence in B is a sequence in C.*

Since executions and traces of TIOAs are (A, V)-sequences, the above definition can be used to talk about executions or traces of one TIOA stabilizing to executions or traces of some other TIOA. The following lemma is a general result that can be used to show, for example, that if executions of one TIOA stabilize to those of another then its traces also stabilize to traces of the other.

Lemma 3.5 *Let B be a set of (A^B, V)-sequences and C be a set of (A^C, V)-sequences, where A^B and A^C are sets of actions and V is a set of variables. Let A be a set of actions and V' a set of variables. If B stabilizes to C in time t, then $\{\alpha_B\lceil(A, V') \mid \alpha_B \in B\}$ stabilizes to $\{\alpha_C\lceil(A, V') \mid \alpha_C \in C\}$ in time t.*

Proof: Say B stabilizes to C in time t. Consider any sequence $\alpha \in \{\alpha_B\lceil(A, V') \mid \alpha_B \in B\}$, and state-matched t-suffix α' of α. We must show that $\alpha' \in \{\alpha_C\lceil(A, V') \mid \alpha_C \in C\}$.

By definition of a state-matched t-suffix, there must exist some α'' such that $\alpha = \alpha''\alpha'$ and $\alpha'.fstate = \alpha''.lstate$. By definition of α, there must exist some $\alpha_B \in B$ such that $\alpha_B\lceil(A, V') = \alpha$ and some prefix α''_B of α_B such that $\alpha''_B\lceil(A, V) = \alpha''$ and $\alpha''_B.ltime = \alpha''.ltime = t$. Since α''_B is a prefix of α_B, there is some α'_B such that $\alpha_B = \alpha''_B\alpha'_B$. Consider any such α'_B and replace its first state with $\alpha''_B.lstate$. This α'_B is a state-matched t-suffix of α_B.

Lemma 3.9 of [58] tells us that $\alpha''_B \alpha'_B \lceil (A, V) = \alpha''_B \lceil (A, V)\ \alpha'_B \lceil (A, V)$. This means that $\alpha'' \alpha' = \alpha'' \alpha'_B \lceil (A, V)$. Since $\alpha'_B.fstate$ is equal to $\alpha''_B.lstate$, $(\alpha'_B \lceil (A, V)).fstate = (\alpha''_B \lceil (A, V)).lstate$, meaning $\alpha'_B \lceil (A, V) = \alpha'$.

Since B stabilizes to C in time t and α'_B is a state-matched t-suffix of a sequence in B, α'_B is in C. This implies that $\alpha'_B \lceil (A, V') \in \{\alpha_C \lceil (A, V') \mid \alpha_C \in C\}$, and hence $\alpha' \in \{\alpha_C \lceil (A, V') \mid \alpha_C \in C\}$. ■

Lemma 3.6 (Transitivity) *Let B be a set of (A^B, V)-sequences, C be a set of (A^C, V)-sequences, and D be a set of (A^D, V)-sequences, where A^B, A^C, and A^D are sets of actions, and V is a set of variables. If B stabilizes to C in time t_1, and C stabilizes to D in time t_2, then B stabilizes to D in time $t_1 + t_2$.*

Proof: Assume B stabilizes to C in time t_1, and C stabilizes to D in time t_2. Consider any sequence α_B in B such that $\alpha_B.ltime \geq t_1 + t_2$ and any state-matched $t_1 + t_2$-suffix α_B^3 of α_B. By our definition of stabilization (Definition 3.4), we must show that $\alpha_B^3 \in D$.

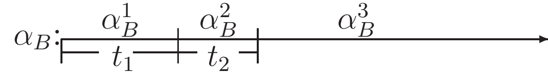


Figure 3-1: Execution α_B for Lemma 3.6.

By our definition of a state-matched $t_1 + t_2$-suffix, there must exist some α'_B such that $\alpha_B = \alpha'_B \alpha_B^3$, $\alpha_B^3.fstate = \alpha'_B.lstate$ and $\alpha'_B.ltime = t_1 + t_2$.

Since $\alpha'_B.ltime \geq t_1$, by Lemma 3.3 there must exist some state-matched t_1-suffix α_B^2 of α'_B. This means that there must exist some α_B^1 such that $\alpha'_B = \alpha_B^1 \alpha_B^2$, $\alpha_B^1.ltime = t_1$, and $\alpha_B^2.fstate = \alpha_B^1.lstate$. This also implies that $\alpha_B = \alpha_B^1 \alpha_B^2 \alpha_B^3$, as in Figure 3-1.

Since $\alpha_B^1.ltime = t_1$ and $\alpha_B^2 \alpha_B^3.fstate = \alpha_B^2.fstate = \alpha_B^1.lstate$, $\alpha_B^2 \alpha_B^3$ is a state-matched t_1-suffix of α_B. Since B stabilizes to C in time t_1 and $\alpha_B^2 \alpha_B^3$ is a state-matched t_1-suffix of α_B, $\alpha_B^2 \alpha_B^3$ is in C.

Also, since $\alpha_B^3.fstate = \alpha'_B.lstate = \alpha_B^2.lstate$, α_B^3 is a state-matched t_2-suffix of $\alpha_B^2 \alpha_B^3$. Since C stabilizes to D in time t_2 and α_B^3 is a state-matched t_2-suffix of a sequence in C, α_B^3 is in D.

We conclude that B stabilizes to D in time $t_1 + t_2$. ■

We can generalize the transitivity lemma to a sequence of sets of hybrid sequences. The proof follows by induction on n, where the inductive step simply applies the above transitivity lemma:

Lemma 3.7 *Let $A_0, A_1, \ldots, A_n$ be a sequence of sets of actions, and let V be a set of variables. Let $B_0, B_1, \ldots, B_n$ be a sequence where for each i such that $0 \leq i \leq n$, B_i is a set of (A_i, V)-sequences. Let $t_1, t_2, \ldots, t_n$ be a sequence of non-negative reals where for each i such that $1 \leq i \leq n$, B_{i-1} stabilizes in time t_i to B_i.*
Then B_0 stabilizes in time t to B_n, for $t = \Sigma_{i,1 \leq i \leq n} t_i$.

The following definitions capture the idea of a TIOA being self-stabilizing when composed with another TIOA, allowing us to write algorithms that can be started in an arbitrary state but take advantage of separate oracles or even other self-stabilizing TIOAs, in order to eventually reach some legal state of the composed automaton. The idea of a TIOA stabilizing relative to another TIOA can be thought of as similar to that of fair composition, described in [27], showing that under certain conditions, if you have a self-stabilizing implementation A of a service that's used by a self-stabilizing implementation B of a higher level service, then B using A is still stabilizing.

We begin by defining a function that takes a TIOA and a state set L and returns the same TIOA with its start state set changed to L.

Definition 3.8 *Let $\mathcal{A}$ be any TIOA and L be any nonempty subset of $Q_{\mathcal{A}}$.*
Then $Start(\mathcal{A}, L)$ is defined to be $\mathcal{A}$ except with $\Theta_{Start(\mathcal{A},L)} = L$.

We then define some shorthand that we will use often in this thesis.

Definition 3.9 *We use notation $U(\mathcal{A})$ for $Start(\mathcal{A}, Q_{\mathcal{A}})$ (or $\mathcal{A}$ started in an arbitrary state), and $R(\mathcal{A})$ for $Start(\mathcal{A}, reachable_{\mathcal{A}})$ (or $\mathcal{A}$ started in a reachable state).*

Lemma 3.10 *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs, $L \subseteq Q_{\mathcal{A}}$, and $L' \subseteq Q_{\mathcal{O}}$. Then:*

1. $Start(\mathcal{A}, L) \| Start(\mathcal{O}, L') = Start(\mathcal{A} \| \mathcal{O}, L \times L')$.
   This says that one can change two automata's start states and then compose them, or compose two automata and then change the resulting automaton's start state, and still get the same result.

2. $frags_{\mathcal{A}}^{L} = execs_{Start(\mathcal{A},L)}$.

This says that any execution fragment of $\mathcal{A}$ starting in L is an execution of $\mathcal{A}$ after its start states are updated to be the set L, and vice versa.

3. *For any $M \subseteq Q_{\mathcal{A}}$, $frags_{Start(\mathcal{A},M)}^{L} = frags_{\mathcal{A}}^{L}$.*

This says that it does not make a difference if an automaton's start states are changed from its original start states when you consider execution fragments that are allowed to start in any state in L.

Proof: 1. The only thing that might differ between the two is the start states, but it is easy to check that the start states of both are $L \times L'$.

2. By definition of an execution and definition of $Start(\mathcal{A}, L)$, an execution of $Start(\mathcal{A}, L)$ is an execution fragment of $\mathcal{A}$ starting with a state in L, and vice versa.

3. Since $\mathcal{A}$ and $Start(\mathcal{A}, L)$ are the same except for start states, then an execution fragment of either machine starting with a state in L is an execution fragment of the other machine starting in a state in L.

■

The following corollary simply states that it does not make a difference if two automata's start states are changed from their original start states when you consider execution fragments of their composition that are allowed to start in any state in some set L''.

Corollary 3.11 *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs, $L \subseteq Q_{\mathcal{A}}$, $L' \subseteq Q_{\mathcal{O}}$, and $L'' \subseteq Q_{\mathcal{A}\|\mathcal{O}}$. Then $frags_{Start(\mathcal{A},L)\|Start(\mathcal{O},L')}^{L''} = frags_{\mathcal{A}\|\mathcal{O}}^{L''}$.*

Proof: By Lemma 3.10, part 1, $frags_{Start(\mathcal{A},L)\|Start(\mathcal{O},L')}^{L''} = frags_{Start(\mathcal{A}\|\mathcal{O},L\times L')}^{L''}$. By part 3, letting $M = L \times L'$, this equals $frags_{\mathcal{A}\|\mathcal{O}}^{L''}$. ■

In the following definition, we describe a legal set for a TIOA as a subset of its states that is closed under steps and closed trajectories of the TIOA.

Definition 3.12 *Let $\mathcal{A}$ be a TIOA, and $L \subseteq Q_{\mathcal{A}}$. L is a* legal set *for $\mathcal{A}$ if:*

1. *For every* $(x, a, x') \in \mathcal{D}_{\mathcal{A}}$*, if* $x \in L$ *then* $x' \in L$.

2. *For every closed* $\tau \in \mathcal{T}_{\mathcal{A}}$*, if* $\tau.fstate \in L$ *then* $\tau.lstate \in L$.

This definition implies the following trivial lemma saying that a legal set for a TIOA is a subset of its states that is closed under execution fragments.

Lemma 3.13 *Let* $\mathcal{A}$ *be a TIOA, and* $L \subseteq Q_{\mathcal{A}}$. L *is a* legal set *for* $\mathcal{A}$ *iff for every closed execution fragment* α *of* $\mathcal{A}$ *such that* $\alpha.fstate \in L$, $\alpha.lstate$ *is also in* L.

The following result is trivial and follows almost immediately from the definition of a legal set.

Lemma 3.14 *Let* $\mathcal{A}$ *be a TIOA,* L *be a legal set for* $\mathcal{A}$*, and* $t \in \mathbb{R}^{\geq 0}$.
Then for any α *in* $frags_{\mathcal{A}}^{L}$ *and any state-matched* t*-suffix* β *of* α, β *is in* $frags_{\mathcal{A}}^{L}$.

This immediately implies the following result about trace fragments:

Lemma 3.15 *Let* $\mathcal{A}$ *be a TIOA and* L *be a legal set for* $\mathcal{A}$.
Then for any α *in* $tracefrags_{\mathcal{A}}^{L}$ *and any suffix* β *of* α, β *is in* $tracefrags_{\mathcal{A}}^{L}$.

Another simple observation is that the set of all states of a TIOA is a legal set for the TIOA:

Lemma 3.16 *Let* $\mathcal{A}$ *be a TIOA. Then* $Q_{\mathcal{A}}$ *is a legal set for* $\mathcal{A}$.

A marginally more ambitious result is that the set of reachable states of a TIOA is a legal set for the TIOA:

Lemma 3.17 *Let* $\mathcal{A}$ *be a TIOA. Then* $reachable_{\mathcal{A}}$ *is a legal set for* $\mathcal{A}$.

Proof: Consider any execution fragment α such that $\alpha.fstate$ is a reachable state of $\mathcal{A}$. We must show that $\alpha.lstate$ is a reachable state of $\mathcal{A}$.

By definition of reachability, if $\alpha.fstate$ is a reachable state of $\mathcal{A}$ then there exists some closed execution α' of $\mathcal{A}$ such that $\alpha'.lstate = \alpha.fstate$. Since the extension of α' with α is an execution of $\mathcal{A}$, we have that $\alpha.lstate$ is a reachable state of $\mathcal{A}$. ■

The definition of composition makes the following lemma trivial. The lemma says that given two TIOAs and two legal sets, one for each TIOA, the cartesian product of the two legal sets is itself a legal set of the composition of the TIOAs.

Lemma 3.18 *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs, and let $L_{\mathcal{O}}$ be a legal set for $\mathcal{O}$ and $L_{\mathcal{A}}$ be a legal set for $\mathcal{A}$. Then $L_{\mathcal{O}} \times L_{\mathcal{A}}$ is a legal set for $\mathcal{O} \| \mathcal{A}$.*

The following lemma is the simple observation that the legal set for some TIOA is also a legal set for the TIOA with some set of its output actions hidden, and vice versa.

Lemma 3.19 *Let $\mathcal{A}$ be a TIOA, $L \subseteq Q_{\mathcal{A}}$, and O be a subset of $O_{\mathcal{A}}$. Then L is a legal set for $\mathsf{ActHide}(O, \mathcal{A})$ iff L is a legal set for $\mathcal{A}$.*

The following result is a core result for substitutivity. It says that if one machine's traces stabilize to the traces of a second machine, then the traces of a third machine started in some legal set and composed with the first stabilizes to the traces of the third machine started in the same legal set and composed with the second. (Having the third machine start in a legal set translates into an execution suffix closure property, where each suffix of an execution of the machine composed with the first or second machine above is still an execution of the machine.)

Theorem 3.20 *Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be TIOAs and L be a legal set for $\mathcal{C}$ such that:*

- *$\mathcal{A}$ is comparable with $\mathcal{B}$, and $\mathcal{C}$ is compatible with $\mathcal{A}$ and compatible with $\mathcal{B}$.*
- *$traces_{\mathcal{A}}$ stabilizes in time t to $traces_{\mathcal{B}}$.*

Then $traces_{Start(\mathcal{C},L) \| \mathcal{A}}$ stabilizes in time t to $traces_{Start(\mathcal{C},L) \| \mathcal{B}}$.

Proof: We must show that for any $\alpha' \alpha$ in $traces_{Start(\mathcal{C},L) \| \mathcal{A}}$ where α is the state-matched t-suffix of $\alpha' \alpha$, α is in $traces_{Start(\mathcal{C},L) \| \mathcal{B}}$.

By Theorem 2.15, this is the same as showing that α is a hybrid sequence with an empty variable set (which is obvious since it is a trace) such that: (1) $\alpha \lceil (E_{Start(\mathcal{C},L)}, \emptyset)$ is in $traces_{Start(\mathcal{C},L)}$, and (2) $\alpha \lceil (E_{\mathcal{B}}, \emptyset)$ is in $traces_{\mathcal{B}}$.

For the first condition, since $\alpha \lceil (E_{Start(\mathcal{C},L)}, \emptyset)$ is a suffix of $\alpha' \alpha \lceil (E_{Start(\mathcal{C},L)}, \emptyset)$, itself a trace of $Start(\mathcal{C}, L)$, Lemma 3.15 implies that $\alpha \lceil (E_{Start(\mathcal{C},L)}, \emptyset)$ is in $traces_{Start(\mathcal{C},L)}$.

For the second condition, since $\alpha\lceil(E_{\mathcal{A}}, \emptyset)$ is a state-matched t-suffix of $\alpha'\alpha\lceil(E_{\mathcal{A}}, \emptyset)$, which is in $traces_{\mathcal{A}}$, the definition of stabilization tells us that $\alpha\lceil(E_{\mathcal{A}}, \emptyset)$ is in $traces_{\mathcal{B}}$. Since $E_{\mathcal{A}} = E_{\mathcal{B}}$ by assumption, we have shown the second condition.

We conclude that $traces_{Start(\mathcal{C},L)\|\mathcal{A}}$ stabilizes in time t to $traces_{Start(\mathcal{C},L)\|\mathcal{B}}$. ∎

The following result shows that saying that some subset S of execution fragments of a TIOA stabilizes in time t to a set of fragments starting in some set of legal states L is the same as saying that any state x that occurs at time t in a fragment in S is in L.

Lemma 3.21 *Let $\mathcal{A}$ be a TIOA, L be a legal set for $\mathcal{A}$, $S \subseteq frags_{\mathcal{A}}$, and $t \in \mathbb{R}^{\geq 0}$. Then S stabilizes in time t to $frags^L_{\mathcal{A}}$ iff for each $\alpha \in S$ and each closed prefix α' of α such that $\alpha'.ltime = t$, $\alpha'.lstate \in L$.*

Proof: $(\Rightarrow)$: Say that S stabilizes in time t to $frags^L_{\mathcal{A}}$. We must show that for each $\alpha \in S$ and each closed prefix α' of α such that $\alpha'.ltime = t$, $\alpha'.lstate \in L$. For any such prefix α' there is some α'' such that $\alpha = \alpha'\alpha''$. Choose such an α'' such that $\alpha''.fstate = \alpha'.lstate$. By definition of a state-matched t-suffix, α'' is a state-matched t-suffix of α. By definition of stabilization, α'' is in $frags^L_{\mathcal{A}}$. This implies that $\alpha''.fstate \in L$. Since $\alpha'.lstate = \alpha''.fstate$, $\alpha'.lstate \in L$.

$(\Leftarrow)$: Say that for each $\alpha \in S$ and each closed prefix α' of α such that $\alpha'.ltime = t$, $\alpha'.lstate \in L$. We must show that S stabilizes in time t to $frags^L_{\mathcal{A}}$. This means that we must show that for any $\alpha \in S$ and state-matched t-suffix α'' of α, α'' is in $frags^L_{\mathcal{A}}$. By definition of state-matched t-suffix, there must exist some prefix α' of α such that $\alpha'.ltime = t$, $\alpha'\alpha'' = \alpha$, and $\alpha''.fstate = \alpha'.lstate$. By assumption, $\alpha'.lstate \in L$, meaning $\alpha''.fstate \in L$, and hence that $\alpha'' \in frags^L_{\mathcal{A}}$. ∎

The following lemma says that if you consider an execution of the composition of a machine started in an arbitrary state together with a machine started in a reachable state, a suffix of that execution is still an execution of the composition with one component started in an arbitrary state together with a machine started in a reachable state.

Lemma 3.22 *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs and t be a nonnegative real. If $\alpha \in execs_{U(\mathcal{A})\|R(\mathcal{O})}$ and β is a state-matched t-suffix of α, then $\beta \in execs_{U(\mathcal{A})\|R(\mathcal{O})}$.*

Proof: This follows immediately from Lemmas 3.14, 3.16, 3.17, and 3.18. ■

This immediately implies the following corollary about traces:

Corollary 3.23 *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs.*
If $\alpha \in traces_{U(\mathcal{A})\|R(\mathcal{O})}$ and β is a suffix of α, then $\beta \in traces_{U(\mathcal{A})\|R(\mathcal{O})}$.

The next definition describes self-stabilization. $\mathcal{A}$ is some TIOA that can be started in an arbitrary initial state, while $\mathcal{O}$ is an oracle TIOA that is composed with $\mathcal{A}$. The legal set L can be thought of as some set of good target states for the composition. $\mathcal{A}$ is said to self-stabilize in time t to L relative to $\mathcal{O}$ if within t time, any execution of $\mathcal{A}\|\mathcal{O}$ with $\mathcal{A}$ started in an arbitrary state reaches a legal state.

Definition 3.24 *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs, and L be a legal set for $\mathcal{A}\|\mathcal{O}$.*
$\mathcal{A}$ self-stabilizes in time t to L relative to $\mathcal{O}$ *if $execs_{U(\mathcal{A})\|\mathcal{O}}$ stabilizes in time t to $frags^{L}_{\mathcal{A}\|\mathcal{O}}$.*

Notice in the definition above that when $\mathcal{O} = R(\mathcal{O}')$ for some TIOA $\mathcal{O}'$, we are effectively describing the capability of a self-stabilizing TIOA $\mathcal{A}$ to recover from a corruption fault, where $\mathcal{A}$'s state can be changed arbitrarily at some point in an execution. Consider an execution of $\mathcal{A}\|\mathcal{O}$ in which a corruption fault occurs at $\mathcal{A}$, changing the state of $\mathcal{A}$ to some arbitrary state. Call the resulting state of $\mathcal{A}\|\mathcal{O}$ state s. That state s is in $Q_{\mathcal{A}} \times reachable_{\mathcal{O}'}$. Any execution fragment starting from s is in $execs_{U(\mathcal{A})\|\mathcal{O}}$. By our definition, $execs_{U(\mathcal{A})\|\mathcal{O}}$ stabilizes to $frags^{L}_{\mathcal{A}\|\mathcal{O}}$, meaning that after a corruption fault, the system stabilizes to a legal state.

Chapter 4

Emulations

In this chapter, we introduce a formal theory of emulations. We start by giving the definition of an emulation. Then we describe emulation stabilization, and show a simple result stating that a stabilizing emulation of a self-stabilizing program has traces that eventually look like those of the self-stabilizing program started from a legal state of that program.

4.1 Emulation

Here we define the concept of emulation, a kind of implementation relationship between two sets of TIOAs, $\mathcal{B}$ and $\mathcal{C}$. Say we have some function emu that maps machines in $\mathcal{C}$ to machines in $\mathcal{B}$. We would like to say, intuitively, that the set of machines $\mathcal{B}$ emulates the set of machines $\mathcal{C}$ if for each $C \in \mathcal{C}$, the machine $emu(C)$ in $\mathcal{B}$ has externally visible behaviors that can be restricted so as to be in some constrained set of the externally visible behaviors of C.

Our definition of an emulation exposes a little more information in that we allow the designation of two automata B' and C' that will be composed with each of the machines in $\mathcal{B}$ and $\mathcal{C}$ respectively. B' and C' are system components that always run the same program, and hence don't change based on which element of C is being emulated. Pulling out these special automata will prove to be very useful when we discuss stabilization of emulations (see Section 4.2). $\mathcal{B}$ then emulates $\mathcal{C}$ in the context of B' and C' if for any C in $\mathcal{C}$, each trace of $emu(C)\|B'$ is a trace in a constrained set of traces of $C\|C'$, subject to some action

hiding.

The constraints on traces of $C \| C'$ are expressed in the definition using the S function, which maps each C to some subset of the execution fragments of $C \| C'$. (We map to execution fragments of $C \| C'$, rather than executions of $C \| C'$, in order to allow us to use the same S later when we consider stabilizating emulations.) We can use S to describe properties of or relationships between the states of C and C'. For example, S might describe a consistency condition between states of C and C', requiring that certain state components in C and C' have the same value. These kinds of conditions can be difficult or tedious to describe otherwise.

Definition 4.1 *Let $\mathcal{B}$ and $\mathcal{C}$ be sets of TIOAs, emu be a function of type $\mathcal{C} \rightarrow \mathcal{B}$, B' and C' be TIOAs, and $E_{\mathcal{B}}$ and $E_{\mathcal{C}}$ be sets of actions such that:*

- *For each $B \in \mathcal{B}$, B is compatible with B' and $E_{\mathcal{B}} \subseteq O_{B \| B'}$.*
- *For each $C \in \mathcal{C}$, C is compatible with C' and $E_{\mathcal{C}} \subseteq O_{C \| C'}$.*

Let S be a function that maps each C in $\mathcal{C}$ to a suffix-closed subset of $frags_{\mathsf{ActHide}(E_{\mathcal{C}}, C \| C')}$.

We say that $(\mathcal{B}, B', E_{\mathcal{B}})$ emulates *$(\mathcal{C}, C', E_{\mathcal{C}})$* constrained to *$S$* with *$emu$ if for each C in $\mathcal{C}$, $traces_{\mathsf{ActHide}(E_{\mathcal{B}}, emu(C) \| B')} \subseteq \{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, C \| C')}\}$.*

In the special case where S maps each C to the entire set of execution fragments of $C \| C'$ after action hiding, we actually are not constraining the set of traces of $C \| C'$ that traces of the emulation are supposed to correspond to. In this case, we drop the "constrained to S" phrase:

Definition 4.2 *Let $(\mathcal{B}, B', E_{\mathcal{B}})$ emulate $(\mathcal{C}, C', E_{\mathcal{C}})$ constrained to S with emu, where S is the function that maps each C in $\mathcal{C}$ to the set $frags_{\mathsf{ActHide}(E_{\mathcal{C}}, C \| C')}$.*

Then we say that $(\mathcal{B}, B', E_{\mathcal{B}})$ emulates *$(\mathcal{C}, C', E_{\mathcal{C}})$ with emu.*

We then conclude that for this special case of S, our emulation definition unrolls to give an implementation result:

Lemma 4.3 *Let $(\mathcal{B}, B', E_{\mathcal{B}})$ emulate $(\mathcal{C}, C', E_{\mathcal{C}})$ with emu.*
Then for every C in $\mathcal{C}$, $\mathsf{ActHide}(E_{\mathcal{B}}, emu(C) \| B') \leq \mathsf{ActHide}(E_{\mathcal{C}}, C \| C')$.

4.2 Emulation stabilization

Now we define emulation stabilization, a concept closely related to self-stabilization. Let's say that $(\mathcal{B}, B', E_{\mathcal{B}})$ emulates $(\mathcal{C}, C', E_{\mathcal{C}})$ constrained to S with emu. We want to define the idea that for any C in $\mathcal{C}$, the machine $emu(C)$ started in an arbitrary state and composed with $R(B')$ (B' started in a reachable state) has traces that are eventually related to constrained traces of C and C'.

What sorts of constrained traces of C and C' should they be related to? Think of C' as the oracle piece; we want to ensure that C' is always in a reachable state. Intuitively, after stabilization an emulation should manage to produce traces that are related to traces of constrained execution fragments of the composition of C and $R(C')$. However, a state of C corresponding to the state at the beginning of such a fragment might be arbitrary; an emulation could stabilize to a point after which it looks like it is running the same program as C but not necessarily starting from an initial or reachable state. Hence, we require that the emulation's traces should stabilize to constrained traces of $U(C)$ (C started in an arbitrary state) composed with $R(C')$, subject to some action hiding.

Definition 4.4 *Let $(\mathcal{B}, B', E_{\mathcal{B}})$ emulate $(\mathcal{C}, C', E_{\mathcal{C}})$ constrained to S with emu, and let t be in $\mathbb{R}^{\geq 0}$. We say that $(\mathcal{B}, B', E_{\mathcal{B}})$* emulation stabilizes in time t to *$(\mathcal{C}, C', E_{\mathcal{C}})$ constrained to S with emu if $traces_{\mathsf{ActHide}(E_{\mathcal{B}}, U(emu(C)) \| R(B'))}$ stabilizes in time t to $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, U(C) \| R(C'))}\}$.*

As before with our definition of emulation, we introduce a term for the special case where S maps each C to the entire set of execution fragments of $C \| C'$ after action hiding:

Definition 4.5 *Let $(\mathcal{B}, B', E_{\mathcal{B}})$ emulation stabilize in time t to $(\mathcal{C}, C', E_{\mathcal{C}})$ constrained to S with emu, where S is the function that maps each C in $\mathcal{C}$ to the set $frags_{\mathsf{ActHide}(E_{\mathcal{C}}, C \| C')}$.*

Then we say that $(\mathcal{B}, B', E_{\mathcal{B}})$ emulation stabilizes in time t to *$(\mathcal{C}, C', E_{\mathcal{C}})$ with emu.*

Lemma 4.6 *Let $(\mathcal{B}, B', E_{\mathcal{B}})$ emulation stabilize in time t to $(\mathcal{C}, C', E_{\mathcal{C}})$ with emu.*

Then for every C in $\mathcal{C}$, $traces_{\mathsf{ActHide}(E_{\mathcal{B}}, U(emu(C)) \| R(B'))}$ stabilizes in time t to $traces_{\mathsf{ActHide}(E_{\mathcal{C}}, U(C) \| R(C'))}$.

Finally, if $(\mathcal{B}, B', E_{\mathcal{B}})$ emulation stabilizes to $(\mathcal{C}, C', E_{\mathcal{C}})$ constrained to S with emu, and some C in $\mathcal{C}$ self-stabilizes to some legal set L relative to $R(C')$, we can easily conclude that the traces of $emu(C)$ started in an arbitrary state and composed with $R(B')$ stabilize to the constrained traces of $C \| C'$ started in L, subject to some action hiding. In other words, a stabilizing emulation of a self-stabilizing program has traces that eventually look like constrained traces of the self-stabilizing program started in a legal state.

Theorem 4.7 *1. Let $(\mathcal{B}, B', E_{\mathcal{B}})$ emulation stabilize in time t_1 to $(\mathcal{C}, C', E_{\mathcal{C}})$ constrained to S with emu.*

2. Let C be an element of $\mathcal{C}$, L be a legal set for $C \| C'$, and $t_2 \in \mathbb{R}^{\geq 0}$ be chosen so that C self-stabilizes to L relative to $R(C')$ in time t_2.

Then $traces_{\mathsf{ActHide}(E_{\mathcal{B}}, U(emu(C)) \| R(B'))}$ stabilizes in time $t_1 + t_2$ to $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, Start(C \| C', L))}\}$.

Proof: By definition of emulation stabilization, $traces_{\mathsf{ActHide}(E_{\mathcal{B}}, U(emu(C)) \| R(B'))}$ stabilizes in time t_1 to $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, U(C) \| R(C'))}\}$.

Since C self-stabilizes to L relative to $R(C')$ in time t_2, the definition of self-stabilization says this means that $execs_{U(C) \| R(C')}$ stabilizes in time t_2 to $execs_{Start(C \| C', L)}$. Since $S(C)$ is suffix-closed, this and Lemma 3.5 imply that $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, U(C) \| R(C'))}\}$ stabilizes in time t_2 to $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, Start(C \| C', L))}\}$.

Since $traces_{\mathsf{ActHide}(E_{\mathcal{B}}, U(emu(C)) \| R(B'))}$ stabilizes in time t_1 to $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, U(C) \| R(C'))}\}$, which in turn stabilizes in time t_2 to $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, Start(C \| C', L))}\}$, Lemma 3.6 implies that $traces_{\mathsf{ActHide}(E_{\mathcal{B}}, U(emu(C)) \| R(B'))}$ stabilizes in time $t_1 + t_2$ to $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(E_{\mathcal{C}}, Start(C \| C', L))}\}$ ■

This immediately implies the following result for the special case S that maps each C to the entire set of execution fragments of $C \| C'$ after action hiding:

Corollary 4.8 *1. Let $(\mathcal{B}, B', E_{\mathcal{B}})$ emulation stabilize in time t_1 to $(\mathcal{C}, C', E_{\mathcal{C}})$ with emu.*

2. Let C be an element of $\mathcal{C}$, L be a legal set for $C \| C'$, and $t_2 \in \mathbb{R}^{\geq 0}$ be chosen so that C self-stabilizes to L relative to $R(C')$ in time t_2.

Then $traces_{\mathsf{ActHide}(E_{\mathcal{B}},U(emu(C))\|R(B'))}$ *stabilizes in time* $t_1 + t_2$ *to* $traces_{\mathsf{ActHide}(E_{\mathcal{C}},Start(C\|C',L))}$.

Chapter 5

Failure transform

This chapter describes a general transformation of a TIOA into a new TIOA that can be crashed and restarted. This is done with the addition of fail and $\mathsf{restart}$ actions and a *failed* variable indicating if the automaton is in a failed state. In this definition, a failed machine is one where no locally-controlled action is enabled, inputs do not change its state, and the values of the variables do not change while time passes. A failed machine can be restarted with a $\mathsf{restart}$ action, making the machine non-failed and initializing the variables of the original machine to a start state of that machine.

After we present the definitions, we present several results with respect to the $Fail$ transform.

The first definition describes a TIOA that we can *Fail* transform.

Definition 5.1 *Let* $\mathcal{A} = (X, Q, \Theta, I, O, H, \mathcal{D}, \mathcal{T})$ *be a TIOA such that* $\{\mathsf{fail}, \mathsf{restart}\} \cap (I \cup O \cup H) = \emptyset$ *and* $failed \notin X$. *Then* $\mathcal{A}$ *is* $Fail$-transformable.

Now we present the definition of the *Fail* transformation of a *Fail*-transformable TIOA.

Definition 5.2 *Let* $\mathcal{A} = (X, Q, \Theta, I, O, H, \mathcal{D}, \mathcal{T})$ *be a* $Fail$*-transformable TIOA.*
Then Fail$(\mathcal{A})$[1] *is defined to be the structure:*

- $X' = X \cup \{failed : Bool, \text{a discrete variable}\}$.

[1] In a system with multiple components with *Fail* transforms we employ the appropriate renaming to keep the *failed* variable and fail and $\mathsf{restart}$ actions unique between the transforms. For example, given TIOAs $\mathcal{A}$ and $\mathcal{B}$, we refer to the *failed* variable in $Fail(\mathcal{A})$ as $failed_{\mathcal{A}}$, and the *failed* variable in $Fail(\mathcal{B})$ as $failed_{\mathcal{B}}$, etc.

- $Q' = \{x \in val(X') \mid x \lceil X \in Q\}$.
- $\Theta' = \{x \in Q' \mid failed \vee x \lceil X \in \Theta\}$.
- $H' = H, O' = O, I' = I \cup \{\mathsf{fail}, \mathsf{restart}\}$.
- $\mathcal{D}'$ *equals the set of* $(x, a, x') \in X' \times A \times X'$ *such that one of the following holds:*
  - $x = x' \wedge x(failed) \wedge a \in I$.
  - $(x \lceil X, a, x' \lceil X) \in \mathcal{D} \wedge \neg x(failed) \wedge \neg x'(failed)$.
  - $x'(failed) \wedge a = \mathsf{fail}$.
  - $x' \lceil X \in \Theta \wedge x(failed) \wedge \neg x'(failed) \wedge a = \mathsf{restart}$.
  - $x = x' \wedge \neg x(failed) \wedge a = \mathsf{restart}$.
- $\mathcal{T}'$ *equals the set of trajectories* $\tau \in trajs(Q')$ *such that one of the following holds:*
  - $\neg \tau(0)(failed) \wedge \tau \downarrow X \in \mathcal{T}$.
  - $\tau(0)(failed) \wedge \tau$ *is any constant trajectory.*

In this definition, a TIOA $\mathcal{A}$ is transformed into *Fail*$(\mathcal{A})$. The new automaton has one additional state variable, *failed*, indicating whether or not the machine is failed, and two additional input actions, fail and restart. The variable *failed* is a discrete variable (defined in Section 2.2), meaning it does not change over the course of a trajectory. The states of the new automaton are states of the old automaton, together with a valuation of the Boolean flag *failed*. The start states of the new machine are defined to be ones where *failed* is arbitrary, but if *failed* is false then the rest of the variables are set to values consistent with a start state of $\mathcal{A}$.

The definition of $\mathcal{D}'$ describes the new set of valid transitions (x, a, x'). First, the set includes the transitions (x, a, x) where the *failed* flag is set in x and a is an input action of $\mathcal{A}$. This basically addresses input-enabling in *Fail*$(\mathcal{A})$ by saying that if a machine is failed, then an input action that occurs results in no change to the state. Second, the set includes "normal" transitions of $\mathcal{A}$ when the machine is not failed– if the machine is not failed in state x and a is in the set of actions of $\mathcal{A}$, then the resulting state x' is still nonfailed, and

$(x\lceil X, a, x'\lceil X)$ is a valid transition of $\mathcal{D}$. Third, we describe the failing of a machine– if $a =$fail, then $x'(failed)$ is true and the rest of the state can be changed arbitrarily. Fourth, we describe the restarting of a failed machine– if $x(failed)$ is true and $a =$restart, then $x'(failed)$ is false and the rest of the variables are initialized to an initial state of $\mathcal{A}$. Fifth, we describe the no-op that results if we restart a non-failed machine– if $a =$ restart and $\neg x(failed)$, then state x equals x'.

The set of trajectories of $\mathcal{T}'$ can be divided into two sets of trajectories based on the value of the *failed* variable. In both sets, the value of the *failed* variable is constant. If *failed* is false over the course of the trajectory τ, then τ is such that $\tau \downarrow X$ is a trajectory of the machine $\mathcal{A}$. In other words, while the machine is not failed its trajectories basically look like those of the original machine. If *failed* is true over the trajectory τ, then all variables are constant in τ. This means that if the machine is failed, then its state variables are frozen. This does not constrain time from passing– any constant trajectory is allowed.

Results about the *Fail* transform

Here we present several results about the failure transformation that will prove useful later in the thesis. The first two results show a relationship between the failure transformation applied to a composition of two TIOAs and the failure transformation of the individual component TIOAs. Then we describe a relationship between the $Fail$ and U operators (useful when considering self-stabilizing algorithms).

The following theorem is an execution projection result that says that performing a *Fail*-transform on the composition $\mathcal{A}_1 \| \mathcal{A}_2$ of two automata results in a machine whose executions constrained to actions and variables of $Fail(\mathcal{A}_1)$ or $Fail(\mathcal{A}_2)$ are executions of $Fail(\mathcal{A}_1)$ or $Fail(\mathcal{A}_2)$ respectively. It follows immediately from the definition of *Fail* and Lemma 2.14.

Theorem 5.3 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be compatible TIOAs that are each $Fail$-transformable, and let α be an execution fragment of $Fail(\mathcal{A}_1 \| \mathcal{A}_2)$. Then $\alpha\lceil(A_1 \cup \{\mathsf{fail}, \mathsf{restart}\}, X_1 \cup \{failed\})$ is an execution fragment of $Fail(\mathcal{A}_1)$, and $\alpha\lceil(A_2 \cup \{\mathsf{fail}, \mathsf{restart}\}, X_2 \cup \{failed\})$*

is an execution fragment of $Fail(\mathcal{A}_2)$*. Also,* α *is an execution of* $Fail(\mathcal{A}_1 \| \mathcal{A}_2)$ *iff* α *restricted in the manner above is an execution of* $Fail(\mathcal{A}_i)$ *for each* $i \in \{1, 2\}$*.*

The following theorem is an execution pasting result similar to Lemma 2.16. Say that we are given two compatible *Fail*-transformable TIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$, and executions α_1 and α_2 of $Fail(\mathcal{A}_1)$ and $Fail(\mathcal{A}_2)$ respectively, where each execution starts with the same value for $failed$. The result says that if there is a hybrid sequence β with the same type as a trace of $Fail(\mathcal{A}_1 \| \mathcal{A}_2)$ and such that β is consistent with the traces of executions α_1 and α_2 in that β restricted to external actions of $Fail(\mathcal{A}_1)$ is equal to the trace of α_1 (and similarly for $\mathcal{A}_2$), then we can paste together the executions α_1 and α_2 to get an execution of $Fail(\mathcal{A}_1 \| \mathcal{A}_2)$ whose trace is equal to β.

It follows immediately from the definition of *Fail* and Lemma 2.16.

Theorem 5.4 *Let* $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$*, and let* α_1 *and* α_2 *be executions of* $Fail(\mathcal{A}_1)$ *and* $Fail(\mathcal{A}_2)$ *respectively such that* $\alpha_1.fstate(failed) = \alpha_2.fstate(failed)$*. Let* β *be an* $(E_{Fail(\mathcal{A})}, \emptyset)$ *sequence such that* $\beta \lceil (E_{Fail(\mathcal{A}_1)}, \emptyset) = trace(\alpha_1)$ *and* $\beta \lceil (E_{Fail(\mathcal{A}_2)}, \emptyset) = trace(\alpha_2)$*.*

Then there exists an execution α *of* $Fail(\mathcal{A})$ *such that* $\alpha_1 = \alpha \lceil (A_{Fail(\mathcal{A}_1)}, X_{Fail(\mathcal{A}_1)})$*,* $\alpha_2 = \alpha \lceil (A_{Fail(\mathcal{A}_2)}, X_{Fail(\mathcal{A}_2)})$*, and* $trace(\alpha) = \beta$*. (Notice that this implies that the* $failed$ *flag in the first state of* α_1 *is equal to the* $failed$ *flag in the first state of* α*, and similarly for the first state of* α_2*.)*

The last result is the following, stating that we can interchange the $Fail$ and U operators on an automaton and get the same result:

Theorem 5.5 *Let* $\mathcal{A}$ *be a TIOA such that* fail *and* restart *is not an action of* $\mathcal{A}$ *and* $failed$ *is not a variable. Then* $Fail(U(\mathcal{A})) = U(Fail(\mathcal{A}))$*.*

Proof: This follows immediately from the definitions of $Fail$ and U. In both cases, the resulting automaton is $\mathcal{A}$ started in an arbitrary state, only with new fail and restart actions and with a new *failed* variable started with an arbitrary value. ■

Chapter 6

Layers: Physical layer model

Here we describe the formal theoretical system model for a mobile network that we will be working with in this thesis.

The physical layer consists of a bounded, tiled region of the plane, where mobile physical (real) nodes are deployed. These nodes are TIOAs susceptible to crash failures and restarts, and with access to local clocks. They also have access to a local broadcast service *Pbcast*, which models broadcasts and receives of messages, and reliable real world automaton, *RW*, which models movement of the physical nodes and real-time. We will use this layer to emulate the VSA layer (we define emulation in Chapter 4).

6.1 Network tiling

The network tiling describes the geography of the network:

- R is the deployment space of the network. It is a fixed, closed, bounded connected portion of the two-dimensional plane.
- $dist : R^2 \rightarrow \mathbb{R}^{\geq 0}$ is the Euclidean distance between two points in R.
- U is the finite totally ordered set of region identifiers.

A *region* is a set of connected points in R, with a unique identifier from U. R is divided into closed regions. The only overlap of points permitted at distinct regions is at the shared boundaries.

- $points : U \rightarrow 2^R$ is a function mapping from region ids to points in R. $points(u)$ is defined to be the set of points in the region corresponding to identifier u.

- $region : R \rightarrow U$ is a function from points in R to region ids. For a point $l \in R$, $region(l)$ is defined to be $min(\{u \in U \mid l \in points(u)\})$, that is, the minimum id of any region containing l.

- $nbrs : 2^{U \times U}$ is a neighbor relation on ids from U. $nbrs$ holds for any two distinct region ids u and v whose regions share any points. More formally, $(u, v) \in nbrs \Leftrightarrow (u \neq v \wedge points(u) \cap points(v) \neq \emptyset)$. Recall that if two distinct regions share any points, these must be boundary points of both regions. This definition implies that diagonally adjacent neighbors in a grid are neighbors, for example.

- $nbrs^+(u) : U \rightarrow 2^U$ refers to the set $\{u\} \cup nbrs(u)$.

- $regDist : U^2 \rightarrow \mathbb{N}$ is the region distance between two regions. For regions u and v, $regDist(u, v)$ is defined to be the hop count of the shortest path between u and v in the neighbor graph induced by the nbr relation. For example, if $u = v$ then $regDist(u, v) = 0$, and if $(u, v) \in nbrs$ then $regDist(u, v) = 1$.

- D, a natural number, is the network diameter in terms of region distances. It is defined as $max_{u,v \in U}\ regDist(u, v)$.

- r, a non-negative real, is an upper bound on the Euclidean distance between two points in the same or neighboring regions. More formally, we require that for every u, v such that $v \in nbrs^+(u)$ and for every $l_1 \in points(u)$ and $l_2 \in points(v)$, $dist(l_1, l_2) \leq r$.

Example: Grid tiling: Tilings are not required to be regular, though this is often useful. One example of a regular tiling is a base b, $b \in \mathbb{R}^{>0}$, grid where R is divided into square $b \times b$ regions. Squares that share edges or are diagonal from one another, sharing a single border point, are neighbors. This means that each non-border square in R has eight neighbors. In such a base b grid, r could be any value greater than or equal to $2\sqrt{2}\, b$.

```
1  Signature:
     Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^{≥0}
3    Input brcv(m)_p, m ∈ Msg
     Output bcast(m)_p, m ∈ Msg
5    Arbitrary additional non-fail, non-restart actions.

7  State:
     analog clock: ℝ^{≥0} ∪ {⊥}, initially ⊥
9    Finite set of additional non-failed variables, each initially
       set to a unique initial value.

   Trajectories:                                              12
     if clock ≠ ⊥ then
       d(clock) = 1                                           14
     else constant clock
       Additional variables develop as specified.             16
```

Figure 6-1: P_p.

Example: Hexagonal tiling: Another example of a regular tiling is a hexagonal, or honeycomb, tiling with edges of length b, $b \in \mathbb{R}^{>0}$. In this case, each interior hexagonal region has six neighbors, one for each edge. r could be any value greater than or equal to $\sqrt{13}\,b$.

6.2 Mobile physical nodes

Here we describe our model of the mobile physical nodes. This model describes the software aspects of the physical nodes, but does not address the actual mobility of the nodes; mobility is modelled by the "Real World" automaton RW, described in Section 6.3.

- P is the set of mobile node ids.
- For each $p \in P$, we assume a mobile physical node P_p from a set of TIOAs, $PProgram_p$. Each physical node P_p is modeled as a timed I/O automaton.
- Msg is the set of messages that a process may broadcast.

We provide an outline of the allowable structure of P_p in Figure 6-1. Each mobile physical node P_p has a local clock variable $clock$. We assume that each node's $clock$ progresses at the rate of real-time, and is initially $\perp$.

We assume that a physical node P_p has at least the following external interface, which includes the ability to broadcast and receive messages and to receive RW updates.

- **Output** $\mathsf{bcast}(m)_p, m \in Msg$:

  A node p may broadcast a message using *Pbcast* through $\mathsf{bcast}(m)_p$.

- **Input** brcv$(m)_p, m \in Msg$:

  A node p receives a message m from *Pbcast* through brcv$(m)_p$.

- **Input** GPSupdate$(l, t)_p, l \in R, t \in \mathbb{R}^{\geq 0}$:

  Such an update from *RW* indicates P_p is currently at location l and the current time is t. If the node adopts the value t as its local clock value $clock$, then since the local $clock$'s value progresses at the rate of real-time, the value of $clock$ will generally be equal to that of real time. However, $clock$ might not be equal to real-time if the node has just recovered from a failure or started in an arbitrary state. In these cases, the periodic GPSupdate can correct the clock value.

We allow additional arbitrary non-fail and non-restart actions and local non-*failed* state. Our restrictions on fail and restart actions and *failed* variables makes P_p *Fail*-transformable, which allows us to use the *Fail* transformation described in Chapter 5 to get *Fail*(P_p).

6.3 *RW*: Real World

RW models system time and mobile node locations. It is an external source of reliable time and location knowledge for physical nodes. The *RW* TIOA in Figure 6-2 maintains location/ time information and updates physical nodes with that information. Its outputs are also inputs to the *Pbcast* service, allowing the broadcast service to guarantee delivery of messages sent between nodes that are located geographically close to each other.

RW is parameterized by the following constants:

- $v_{max} : \mathbb{R}^{\geq 0}$, a maximum velocity magnitude for mobile node motion.

- $\epsilon_{sample} : \mathbb{R}^{\geq 0}$, a maximum amount of time between updates for each node.

RW maintains a variable, now, that is considered the true system time, and two mappings from the set of physical node ids P, loc and $updates$:

- now is a non-negative real representing the current true system time.

```
   Signature:                                              Trajectories:
2    Output GPSupdate(l, t)_p, l ∈ R, p ∈ P, t ∈ ℝ^≥0        evolve                                         10
                                                              d(now) = 1
4  State:                                                     |d(loc(p))| ≤ v_max, for each p ∈ P           12
     analog now: ℝ^≥0, initially 0                          stop when
6    updates(p): 2^(R×ℝ^≥0), for each p ∈ P, initially ∅      ∃p ∈ P: ∀⟨l, t⟩ ∈ updates(p): now ≥ t + ε_sample  14
     loc(p): R, for each p ∈ P, initially arbitrary
                                                           Transitions:                                     16
                                                             Output GPSupdate(l, t)_p
                                                             Precondition:                                  18
                                                               l = loc(p) ∧ t = now ∧ ∀⟨u, t'⟩ ∈ updates(p): t ≠ t'
                                                             Effect:                                        20
                                                               updates(p) ← updates(p) ∪ {⟨l, t⟩}
```

Figure 6-2: $RW[v_{max}, \epsilon_{sample}]$.

```
   prevUpd (p: P, t: ℝ^≥0): ℝ^≥0 ∪ {⊥}                        reg+(p: P, t: ℝ^≥0): U ∪ {⊥}                                   12
2    if ∃⟨l, t'⟩ ∈ updates(p): t' < t then                       if ∃l ∈ R: ⟨l, t⟩ ∈ updates(p) then
       return max({t' ∈ ℝ^≥0 | ∃⟨l,t'⟩ ∈ updates(p): t' < t})     return min({v ∈ U | ∃l ∈ points(v): ⟨l,t⟩ ∈ updates(p)})  14
4    else return ⊥                                               else return reg−(p, t)
                                                                                                                            16
6  reg−(p: P, t: ℝ^≥0): U ∪ {⊥}                                reg(p: P): U ∪ {⊥}
     if ∃⟨l, t'⟩ ∈ updates(p): t' < t then                       return reg+(p, now)                                         18
8        return min({v ∈ U |
             ∃l ∈ points(v): ⟨l, prevUpd(p, t)⟩ ∈ updates(p)})
10   else return ⊥
```

Figure 6-3: *RW* derived variables.

- $loc : P \rightarrow R$ maps each physical node id to a point in the plane indicating the node's current location. Initially this is arbitrary. We assume that the magnitude of the change in loc for each $p \in P$ is bounded by speed v_{max}.

- $updates : P \rightarrow 2^{R \times \mathbb{R}^{\geq 0}}$, maps the set of physical node ids, P, to a set of pairs $\langle l, t \rangle$, indicating that a GPSupdate$(l, t)_p$ occurred. Initially, this set is empty.

When RW outputs a GPSupdate$(loc(p), now)_p$ at a mobile node P_p, informing the node of the node's location $loc(p)$ and the current time, the pair $\langle loc(p), now \rangle$ is stored in $updates(p)$ as a record of the update. A GPSupdate is required to occur at a mobile node P_p at time 0 (guaranteed by the stopping condition on line 14 and the fact that *updates(p)* starts out empty for each p in P) and at least every ϵ_{sample} time thereafter (guaranteed by the stopping condition expressed on line 14). A GPSupdate is allowed to occur only once at any particular time t and particular process P_p (guaranteed by the precondition that $\forall \langle l, t' \rangle \in updates(p) : t' \neq t$ on line 19). This precondition is useful later in preventing certain race conditions from occurring when a node restarts after a failure.

We also define several derived function variables that will be useful throughout this thesis (see Figure 6-3):

- $prevUpd : (P \times \mathbb{R}) \rightarrow \mathbb{R} \cup \{\perp\}$ maps a physical node id p and time t to the time t' of the last $\mathsf{GPSupdate}_p$ that occurred before time t. This is calculated to be $max(\{t' \mid \exists \langle l, t' \rangle \in updates(p) : t' < t\})$. If no such time exists, it returns $\perp$.

- $reg^- : (P \times \mathbb{R}) \rightarrow U \cup \{\perp\}$ maps a physical node id p and time t to the region indicated by the last $\mathsf{GPSupdate}_p$ before time t. It is defined as the $v \in U$ such that there is a location l in $region(v)$ such that $\langle l, prevUpd(p,t) \rangle \in updates(p)$. If no such region exists, it returns $\perp$.

  This function is useful for referring to the region that a process is in, as indicated by the history of GPSupdates stored in $RW.updates$, at the beginning of some time. A process P_p can be considered to be in two different regions at the same time t in an execution. For example, a process's region in some execution at time t might be a region v. Then a $\mathsf{GPSupdate}(l,t)_p$ could occur, changing the region to $u = region(l)$. This means that the variable $RW.reg(p)$ at time t is set to v at the beginning of time t, and set to u at the end of time t. The function reg^- returns the first region, v.

- $reg^+ : (P \times \mathbb{R}) \rightarrow U \cup \{\perp\}$ maps a physical node id p and time t to the region indicated by the $\mathsf{GPSupdate}_p$ that occurs at time t if it exists, and to the result of $reg^-(p,t)$ if it does not.

  Similarly to reg^-, this function is useful for referring to the region that a process is in at some time t, though in this case it refers to the region at the end of that time. For example, in the scenario described in reg^-, $reg^+(p,t)$ would return u.

- $reg : P \rightarrow U \cup \{\perp\}$ maps a physical node id p to the region of the node as indicated by the last $\mathsf{GPSupdate}_p$. This is the last reported region of the node, and is defined to be $reg^+(p, now)$.

Reachable states of *RW*

Here we characterize the reachable states of RW by providing a list of properties exactly describing those states. We show that (1) the list of properties is an invariant for RW and (2) any state satisfying the list of properties is indeed a reachable state of RW.

First we describe the reachable states of RW.

Definition 6.1 *Define Inv_{RW} to be the set of states x of RW such that the following properties hold:*

1. $\forall t \in (0, now], \forall p \in P, \exists \langle l, t' \rangle \in updates(p) : |t - t'| \leq \epsilon_{sample}$.
   This means that for any time t after 0 and up to the clock time in x, for each $p \in P$ there is some $\langle l, t' \rangle$ pair in $updates(p)$ where t' is within ϵ_{sample} time of t.

2. $\forall p \in P, \forall \langle l, t \rangle \in updates(p) : t \in [0, now]$.
   This means that there are no update records that indicate an update occurred before time 0 or after the current time.

3. $\forall p \in P, \forall \langle l, t \rangle, \langle l', t' \rangle \in updates(p) : [t = t' \Rightarrow l = l']$.
   This means that there is at most one update record for a particular time and process.

4. *There exists a function $locAt : (P \times \mathbb{R}^{\geq 0}) \rightarrow R$ such that for all $p \in P$:*

   (a) $\forall \langle l, t \rangle \in updates(p) : locAt(p, t) = l$.

   (b) $locAt(p, now) = loc(p)$.

   (c) $\forall t_1, t_2 : 0 \leq t_1 < t_2 \leq now : \frac{|locAt(p,t_1) - locAt(p,t_2)|}{|t_1 - t_2|} \leq v_{max}$.
   This means that there is a function that can describe for each $p \in P$ a location at any time between 0 and the current time that is consistent with the update histories stored in $updates(p)$, the current location, and the maximum speed restriction of v_{max}.

We now show that the set of properties describing Inv_{RW} is an invariant for RW. We do this by showing that every reachable state of RW is in Inv_{RW}.

Lemma 6.2 $reachable_{RW} \subseteq Inv_{RW}$.

Proof: Consider a state in $reachable_{RW}$. We must show that it satisfies the properties of a state in Inv_{RW}. This is the same as showing that the last state of any closed execution of RW is in Inv_{RW}. We proceed by induction on closed executions of RW.

First, we check that the initial state of RW satisfies the list of properties above. Since $updates(p)$ are empty for each $p \in P$, the properties are trivially satisfied.

Next we check that if the properties hold in some state x and an action is performed that leads to state x', then the properties hold in state x'. We break this down by action:

- GPSupdate$(l, t)_p$: It is easy to see that Properties 1 and 2 still hold. Property 3 can only be violated in there exists some pair $\langle l', t \rangle \in updates(p)$ where $l \neq l'$. However, by Property 4 in state x, $l = locAt(p, now)$. Since $loc(p)$ does not change in 0 time, then $loc(p) = locAt(p, now)$, meaning that $l = l'$. For Property 4, the $locAt$ function that exists for state x would still satisfy our requirements in state x'.

Finally we check that for any closed trajectory τ starting with a state x where the properties hold and ending in a state x', the properties hold in state x'. The only interesting properties to check are 1 and 4. Property 1 will still hold due to the stopping condition expressed in line 14. For Property 4, simply adopt the function $locAt$ that must exist at the beginning of the trajectory and extend the mapping for $t \geq x(now)$ for each $p \in P$ to be $loc(p)$ at time t in τ. The resulting function will satisfy 4(a) and 4(b). 4(c) will hold due to the trajectory restriction described on line 12. ■

Now we show the opposite direction, namely that any state in Inv_{RW} is a reachable state of RW. We do this by showing how, given a state x in Inv_{RW}, we can construct an execution of RW that ends in x.

Lemma 6.3 $Inv_{RW} \subseteq reachable_{RW}$.

Proof: Consider a state x in Inv_{RW}. We must show that x is a reachable state of RW. We do this by constructing an execution α of RW such that $\alpha.lstate = x$. This execution describes the motion of the physical nodes and contains only GPSupdate events.

By property 4 in the description of Inv_{RW}, there exists some function $locAt$ to describe the location of each process from time 0 to $x(now)$. We use this to describe an execution

```
   Signature:
2    Input GPSupdate(l, t)_p, l ∈ R, p ∈ P, t ∈ ℝ^≥0
     Input bcast(m)_q, m ∈ Msg, q ∈ P
4    Output brcv(m)_p, m ∈ Msg, p ∈ P
     Internal drop(m, t, q, p), m ∈ Msg, t ∈ ℝ^≥0, p, q ∈ P
6
   State:
8    analog now: ℝ^≥0, initially 0
     updates(p): 2^(R×ℝ^≥0), for each p ∈ P, initially ∅
10   pbcastq(p): 2^(Msg×ℝ^≥0×2^P), for each p ∈ P, initially ∅

12 Trajectories:
     evolve
14     d(now) = 1
     stop when
16     ∃p∈ P:∃⟨m,t,P′⟩∈ pbcastq(p):[t= now-d_phys ∧ P′≠ ∅ ]

18 Transitions:
     Input GPSupdate(l, t)_p
20   Effect:
       updates(p) ← updates(p) ∪ {⟨l, t⟩}

   Input bcast(m)_p
   Effect:                                                          24
     if ∀⟨m′, t, P′⟩ ∈ pbcastq(p): [m′ ≠ m ∨ t ≠ now ] then
       pbcastq(p) ← pbcastq(p) ∪ {⟨m, now, P⟩}                      26

   Output brcv(m)_q                                                 28
   Local:
     p: P, t: ℝ^≥0, P′: 2^P                                         30
   Precondition:
     ⟨m, t, P′⟩ ∈ pbcastq(p) ∧ q ∈ P′ ∧ t ≠ now                     32
   Effect:
     pbcastq(p) ← pbcastq(p) -{⟨m, t, P′⟩} ∪ {⟨m, t, P′-{q}⟩}       34

   Internal drop(m, t, q, p)                                        36
   Local:
     l, l′: R, t′ ≤ t, t′′: ℝ^≥0, P′: 2^P                           38
   Precondition:
     ⟨l, t′⟩∈ updates(p)∧∀ ⟨l*, t*⟩∈ updates(p):[t*≤ t′∨ t*> t]     40
     ⟨l′, t′′⟩∈ updates(q)∧∀ ⟨l*, t*⟩ ∈ updates(q): t* ≤ t′′
     ⟨m, t, P′⟩∈ pbcastq(p) ∧q∈ P′ ∧t≠ now ∧dist(l, l′)> r_real     42
   Effect:
     pbcastq(p) ← pbcastq(p)-{⟨m, t, P′⟩}∪ {⟨m, t, P′-{q}⟩}         44
```

Figure 6-4: $Pbcast[d_{phys}, r_{real}]$.

α, where the evolution of the variable loc in α from time 0 to now is defined as follows: for each $p \in P$ and time t, $loc(p)$ at time t in α is equal to $locAt(p, t)$. In addition, for each $p \in P$ and $\langle l, t \rangle \in updates(p)$, we add a GPSupdate$(l, t)_p$ action at time t in α. If more than one GPSupdate occurs at any time t, order the GPSupdates by the process for which the update is occurring (recall that by property 3 there is at most one GPSupdate per process at a particular time). It is easy to see that α is an execution of RW: by Property 4 and our construction of the evolution of loc, the change in location of processes satisfies the requirements for an execution of RW. By Properties 2, 3, and 4, each GPSupdate is enabled. By Property 1, GPSupdates occur often enough to satisfy the stopping conditions of RW in line 14. It is also easy to see that $\alpha.lstate$ is equal to x. ■

The preceding two lemmas directly imply the following characterization theorem:

Theorem 6.4 $Inv_{RW} = reachable_{RW}$.

6.4 *Pbcast*: Local broadcast service

Each node has access to the local broadcast communication service *Pbcast*, modelled in Figure 6-4. The service is parameterized with the following:

- r_{real}, a non-negative real representing the minimum broadcast radius of the nodes. We require that $r_{real} \geq r + \epsilon_{sample} v_{max}$.

- d_{phys}, a non-negative real representing the message delay upper bound.

The service described in Figure 6-4 allows each client P_p to broadcast a message to all nearby clients through $\mathsf{bcast}(m)_p$ and receive messages broadcast by other clients through $\mathsf{brcv}(m)_p$.

The main variable of this service is $pbcastq(p)$ for each $p \in P$, storing information about broadcasts performed by P_p. When a $\mathsf{bcast}(m)_p$ input occurs at some time t, if no $\mathsf{bcast}(m)_p$ already occurred at time t, *Pbcast* adds a $\langle m, t, P \rangle$ tuple to $pbcastq(p)$. The set of process ids in the third component of the tuple represents the set of processes that might still potentially receive the message. Some positive amount of time after the broadcast (guaranteed by the precondition that $t \neq now$ on line 32 and 42), a process P_q in the set can either receive the message (lines 28-34), or if P_q's last reported location l' (as described on line 41) is farther than r_{real} from the last reported location l of the sender at time t (as described on line 40), the transmission may fail to P_q (lines 36-44). In either case, the id q is removed from the set of ids of processes that might still receive the message. We require that once a message is broadcast, for every node the message is received or the transmission fails by at most d_{phys} time later (guaranteed by the stopping condition expressed on line 16). Our requirement that a non-0 amount of time pass between broadcast and the possible receiving or dropping of the message is utilized later to prevent race conditions that can result when a process changes regions or failure modes.

6.4.1 Properties of *Pbcast*

The service guarantees that in each execution α of *Pbcast*, there exists a function mapping each $\mathsf{brcv}(m)_q$ event to the $\mathsf{bcast}(m)_p$ event that caused it such that the following hold:

- *Integrity*: If a $\mathsf{brcv}(m)_q$ event π is mapped to a $\mathsf{bcast}(m)_p$ event π', then π' occurs before π.

- *Non-duplicative delivery*: If a $\mathsf{brcv}(m)_q$ event π is mapped to a $\mathsf{bcast}(m)_p$ event π' which occurs at a time t, then there do not exist any other $\mathsf{brcv}(m)_q$ events that map to a $\mathsf{bcast}(m)_p$ event at time t. (Notice that this is slightly stronger than the normal non-duplicative delivery property. Here, if some process sends the same message more than once at some time t, this property implies that at most one copy of the message is received by any process. This is enforced through the check on line 25 for whether the sender has already sent a copy of the message at this time.)

- *Bounded-time delivery*: If a $\mathsf{brcv}(m)_q$ event π is mapped to a $\mathsf{bcast}(m)_p$ event π' where π' occurs at time t, then event π occurs in the interval $(t, t + d_{phys}]$.

- *Reliable local delivery*: This guarantees that a transmission will be received by nearby nodes: If a $\mathsf{bcast}(m)_p$ event π' occurs at time t where the last recorded location of p by the end of time t is l and $\alpha.ltime > t + d_{phys}$, and for each last recorded location l' of q in the entire interval $[t, t + d_{phys}]$, $dist(l, l') \leq r_{real}$, then there exists a $\mathsf{brcv}(m)_q$ event π such that π is mapped to some $\mathsf{bcast}(m)_p$ event (not necessarily π') at time t. (This property is enforced through the preconditions for the drop action in lines 38-42. A process fails to receive a message transmitted at time t only if at some point during the transmission interval it is too far, as reported by GPSupdates, from the last reported location of the transmitter at time t.)

Notice that we are not concerned with the failure status of physical nodes in our model of *Pbcast*. Messages are delivered entirely based on the locations of the nodes. If a *Fail*-transformed physical node is failed when a brcv event occurs for it, then by our definition of the *Fail* transform, the brcv event is a no-op.

Clearly, this is a theoretical abstract model of broadcast communication available to mobile nodes. In real mobile network deployments, reliable local delivery can be difficult to achieve. While the abstract model assumed here does accommodate the possibility of bounded-time retransmission to tackle wireless broadcast issues such as message collisions, it does not handle the reality of having only *high probability* bounded-time retransmission. There is ongoing work towards providing reliable communication in wireless networks with collision failures [16, 17], but coping with such settings is beyond the scope of this

thesis.

6.4.2 Reachable states of *Pbcast*

Here we characterize the reachable states of *Pbcast* by providing a list of properties exactly describing those states. We show that (1) the list of properties is an invariant for *Pbcast* and (2) any state satisfying the list of properties is indeed a reachable state of *Pbcast*.

First we describe the reachable states of *Pbcast*.

Definition 6.5 *Define* Inv_{Pbcast} *to be the set of states* x *of* Pbcast *such that the following properties hold:*

1. $\forall p \in P, \forall \langle m, t, P' \rangle \in pbcastq(p) : t \in [0, now]$.
   This means that the timestamp attached to a message broadcast record is not for a time before 0 or after the current time.

2. $\forall p \in P, \forall \langle m, t, P' \rangle \in pbcastq(p) : [t < now - d_{phys} \Rightarrow P' = \emptyset]$.
   This means that for any record of a message broadcast from more than d_{phys} *time ago, the set of processes yet to either receive the message or drop it is empty.*

3. $\forall p \in P, \forall \langle m, t, P' \rangle \in pbcastq(p) : [t = now \Rightarrow P = P']$.
   This means that for any record of a message broadcast that occurred at the current time, no process has yet received or dropped the message.

4. $\forall p \in P, \forall \langle m, t, P' \rangle, \langle m', t', P'' \rangle \in pbcastq(p) : [\langle m, t \rangle = \langle m', t' \rangle \Rightarrow P' = P'']$.
   This means that for any two distinct records of message broadcasts from the same time t *in* $pbcastq(p)$ *for some* $p \in P$*, the messages must be different.*

We now show that the set of properties describing Inv_{Pbcast} is an invariant for *Pbcast*. We do this by showing that every reachable state of *Pbcast* is in Inv_{Pbcast}.

Lemma 6.6 $reachable_{Pbcast} \subseteq Inv_{Pbcast}$.

Proof: Consider a state in $reachable_{Pbcast}$. We must show that it satisfies the properties of a state in Inv_{Pbcast}. This is the same as showing that the last state of any closed execution of $Pbcast$ is in Inv_{Pbcast}. We proceed by induction on closed executions of *Pbcast*.

First, we check that the initial state of *Pbcast* satisfies the list of properties above. Since $pbcastq(p)$ is empty for each $p \in P$, the properties are trivially satisfied.

Next we check that if the properties hold in some state x and an action is performed that leads to state x', then the properties hold in state x'. We break this down by action:

- GPSupdate$(l, t)_p$: It is easy to see that the properties still hold.

- bcast$(m)_p$: It is easy to see that all properties except 1 are not affected. Properties 1 and 3 are satisfied by the structure of the tuple added to $pbcastq(p)$ in line 26. Property 4 will still hold because of the test on line 25.

- brcv$(m)_q$: The only possibly nontrivial property verification to be done is for Property 3. However, the precondition for a brcv on line 32 states that $t \neq now$. Hence, Property 3 will continue to hold.

- drop(m, t, q, p): The only possibly nontrivial property verification for this action is for Property 3. By the precondition on line 42, we know that $t \neq now$. Hence, the property still holds.

Finally we check that for any closed trajectory τ starting with a state x where the properties hold and ending in a state x', the properties hold in state x'. The only interesting property to check is 2. Property 2 will still hold due to the stopping condition expressed in line 16. ■

Now we show the opposite direction, namely that any state in Inv_{Pbcast} is a reachable state of *Pbcast*. We do this by showing how, given a state x in Inv_{Pbcast}, we can construct an execution of *Pbcast* that ends in x.

Lemma 6.7 $Inv_{Pbcast} \subseteq reachable_{Pbcast}$.

Proof: Consider a state x in Inv_{Pbcast}. We must show that x is a reachable state of *Pbcast*. We do this by constructing an execution α of *Pbcast* such that $\alpha.lstate = x$.

The construction is done in two phases. First, we construct an execution α_1 which describes the GPSupdates that occurred. Then we construct α by adding a bcast event

to α_1 for each message tuple in $pbcastq(p), p \in P$, together with brcv events for processes whose ids do not appear in the tuple's set of process ids. We describe this construction in more detail below.

For execution α_1, for each $p \in P$ and $\langle l, t \rangle \in updates(p)$, we add a $\mathsf{GPSupdate}(l, t)_p$ action at time t in α_1. If more than one GPSupdate occurs at any time t, we order the updates by the process for which the update occurs. It is easy to see that α_1 is an execution of *Pbcast*: since GPSupdate is an input it is always enabled. It is also easy to see that $\alpha_1.lstate$ restricted to the value of the now and $updates$ variables is equal to the value of x restricted in a similar manner. $\alpha_1.lstate$, however, has an empty $pbcastq(p)$ for each $p \in P$.

We then create α by adding bcast and brcv events to α_1 in the following way: For each $p \in P$ and $\langle m, t, P' \rangle \in pbcastq(p)$, add a $\mathsf{bcast}(m)_p$ event at time t, and for every q not in P', add a $\mathsf{brcv}(m)_q$ after the bcast action at time $min(t + d_{phys}, x(now))$. Since bcast is an input action, it is always enabled. Since the time t is not $x(now)$ in any of the records and properties 3 and 4 hold, any of the brcv events is enabled. We also need to check that the stopping conditions in line 16 are not violated; a violation can only occur in our construction if some bcast event is added to α more than d_{phys} time before $x(now)$ and there is some process for which a corresponding brcv event does not occur. By property 2, any tuple from more than d_{phys} time before $x(now)$ has an empty set P', meaning that our construction added an associated brcv for each process, and the stopping condition was not violated. Hence, α is an execution of *Pbcast*.

The only thing remaining to be checked is that the value of $\alpha.lstate(pbcastq)$ is equal to that of x. This is easy to see by the construction of α and property 1. ■

The preceding two lemmas directly imply the following characterization theorem:

Theorem 6.8 $Inv_{Pbcast} = reachable_{Pbcast}$.

6.4.3 Reachable states of *RW*||*Pbcast*

Here we characterize the reachable states of *RW*||*Pbcast* by providing a list of properties exactly describing those states. We show that (1) the list of properties is an invariant for

RW||*Pbcast* and (2) any state satisfying the list of properties is indeed a reachable state of *RW*||*Pbcast*. We then show a useful result about the relationship between broadcast and receive events and the regions of nodes.

First we describe the reachable states of *RW*||*Pbcast*.

Definition 6.9 *Define $Inv_{RW\|Pbcast}$ to be the set of states x of $RW\|Pbcast$ such that the following properties hold:*

1. $x\lceil X_{RW} \in Inv_{RW}$.

   This means that the RW-related elements of state satisfy the properties of Inv_{RW}.

2. $x\lceil X_{Pbcast} \in Inv_{Pbcast}$.

   This means that the $Pbcast$-related elements of state satisfy the properties of Inv_{Pbcast}.

3. $Pbcast.now = RW.now \wedge Pbcast.updates = RW.updates$.

   This means that the clock values and update records are the same between $Pbcast$ and RW.

We now show that the set of properties describing $Inv_{RW\|Pbcast}$ is an invariant for *RW*||*Pbcast*. We do this by showing that every reachable state of *RW*||*Pbcast* is in $Inv_{RW\|Pbcast}$.

Lemma 6.10 $reachable_{RW\|Pbcast} \subseteq Inv_{RW\|Pbcast}$.

Proof: Consider a state in $reachable_{RW\|Pbcast}$. We must show that it satisfies the properties of a state in $Inv_{RW\|Pbcast}$. This is the same as showing that the last state of any closed execution of *RW*||*Pbcast* is in $Inv_{RW\|Pbcast}$. By Lemma 6.2, property 1 of $Inv_{RW\|Pbcast}$ holds throughout such an execution. By Lemma 6.6, property 2 of $Inv_{RW\|Pbcast}$ holds throughout such an execution. That leaves only property 3 to show. We proceed by induction on closed executions of *RW*||*Pbcast*.

First, we check that the initial state of *RW*||*Pbcast* satisfies property 3. Since $pbcastq(p)$ and $updates(p)$ are empty for each $p \in P$ and both start with $now = 0$, property 3 is trivially satisfied.

Next we check that if property 3 holds in some state x and an action is performed that leads to state x', then property 3 holds in state x'. We break this down by action:

- GPSupdate$(l,t)_p$: The pair $\langle l,t\rangle$ is added to $updates(p)$ in both RW and *Pbcast*, so since property 3 holds in state x, it still holds in x'.

- bcast$(m)_p$, brcv$(m)_q$, drop(m,t,q,p):
  These do not impact $updates(p)$ or now, so property 3 still trivially holds.

Finally we check that for any closed trajectory τ starting with a state x where property 3 holds and ending in a state x', property 3 holds in state x'. The $updates$ variable does not change over a trajectory and the now variables develop at the same rate. Hence, property 3 holds in state x'. ■

Now we show the opposite direction, namely that any state in $Inv_{RW\|Pbcast}$ is a reachable state of *RW*‖*Pbcast*. We do this by showing how, given a state x in $Inv_{RW\|Pbcast}$, we can construct an execution of *RW*‖*Pbcast* that ends in x.

Lemma 6.11 $Inv_{RW\|Pbcast} \subseteq reachable_{RW\|Pbcast}$.

Proof: Consider a state x in $Inv_{RW\|Pbcast}$. We must show that x is a reachable state of *RW*‖*Pbcast*. We do this by taking an execution α_{RW} of RW and an execution α_{Pbcast} of $Pbcast$ and pasting them together to get an execution α of *RW*‖*Pbcast* where $\alpha.lstate = x$.

Let α_{RW} be the execution of RW with $\alpha_{RW}.lstate = x\lceil X_{RW}$ constructed in Lemma 6.3, which exists because x satisfies property 1. Let α_{Pbcast} be the execution of $Pbcast$ with $\alpha_{Pbcast}.lstate = x\lceil X_{Pbcast}$ constructed in Lemma 6.7, which exists because x satisfies property 2. Let β be $trace(\alpha_{Pbcast})$. Obviously, $\beta\lceil(E_{Pbcast},\emptyset) = trace(\alpha_{Pbcast})$. Because of property 3 and the construction of the two executions, it is obvious that $trace(\alpha_{RW}) = trace(\beta)\lceil(E_{RW},\emptyset)$. Hence, by Lemma 2.16, there exists an execution α of *RW*‖*Pbcast* such that $\alpha_{RW} = \alpha\lceil(A_{RW}, X_{RW})$ and $\alpha_{Pbcast} = \alpha\lceil(A_{Pbcast}, X_{Pbcast})$. By construction, $\alpha.lstate$ must equal x. ■

The preceding two lemmas directly imply the following characterization theorem:

Theorem 6.12 $Inv_{RW\|Pbcast} = reachable_{RW\|Pbcast}$.

We now present a result that will be used later in the thesis. Using Theorem 6.12, our upper bounds on region sizes allow us to conclude that after a broadcast at time t from a process p whose GPSupdates indicate it starts in a region u (equal to $reg^-(p,t)$) and ends in a region v (equal to $reg^+(p,t)$) at time t, a brcv for the message will be output by *Pbcast* for each process whose GPSupdates indicate it is in u, v, or neighboring regions (equal to $nbrs^+(u)$ and $nbrs^+(v)$) for the entire duration of the message broadcast interval:

Lemma 6.13 *Let α be an execution of $RW \| Pbcast$ and let map be a function mapping from each* brcv$(m)_q$ *event to a* bcast$(m)_p$ *event such that the Integrity, Non-duplicative delivery, Bounded-time delivery, and Reliable local delivery properties hold.*
Suppose a bcast$(m)_p$ *event π' occurs at time t and $\alpha.ltime > t + d_{phys}$. Consider any q such that for all t^* in the interval $[t, t + d_{phys}]$, $reg^+(q, t^*) \in nbrs^+(reg^-(p,t)) \cup nbrs^+(reg^+(p,t))$. Then there exists a* brcv$(m)_q$ *event π such that π is mapped to π'.*

Proof: If a GPSupdate$_p$ event occurs at time t, then let l be the associated location, else let l be the associated location of the last GPSupdate$_p$ event before time t. Let q be an id such that for all t^* in the interval $[t, t + d_{phys}]$, $reg^+(q, t^*) \in nbrs^+(reg^-(p,t)) \cup nbrs^+(reg^+(p,t))$. We must show that there exists a brcv$(m)_q$ event π such that π is mapped to π'. By the Reliable local delivery property of *Pbcast* (Section 6.4), this result would be implied if we could just show that for each time t^* and location l' such that l' is the most recent location record of q in *Pbcast.updates(q)* at time t^*, $dist(l, l') \leq r_{real}$.

We consider cases for the region of l'. If its region is in $nbrs^+(reg^-(p,t))$, then by our upper bound on region size, the distance between l' and any point in $nbrs^+(reg^-(p,t))$ is at most r. If point l is in $reg^-(p,t)$, then this implies that $dist(l, l') \leq r$. If point l is not in $reg^-(p,t)$, then a GPSupdate$_p$ occurred at time t. By property 1 of Inv_{RW}, the last update before time t occurred for some point l'' in $reg^-(p,t)$ no more than ϵ_{sample} before t, and by property 4 of Inv_{RW}, the maximum distance that could have been travelled in that time is $\epsilon_{sample} v_{max}$, meaning point l is no more than $\epsilon_{sample} v_{max}$ from the point l''. Hence, $dist(l, l') \leq dist(l, l'') + dist(l'', l') \leq r + \epsilon_{sample} v_{max} \leq r_{real}$.

If the region of l' is in $nbrs^+(reg^+(p,t))$, then by our upper bound on region size, the distance between l' and any point in $nbrs^+(reg^+(p,t))$ is at most r. Since point l is located

in $reg^+(p, t)$, $dist(l, l') \leq r \leq r_{real}$. ■

6.5 P-algorithms and *PLayers*

Here we define a physical layer algorithm and the complete physical layer.

First, we define a physical layer algorithm. A physical layer algorithm is just an assignment of a TIOA program to each physical node.

Definition 6.14 *A P-algorithm, $palg : P \rightarrow PProgram_p$, is a mapping from each mobile node id $p \in P$ to some TIOA $P_p \in PProgram_p$. The set of all P-algorithms is referred to as $PAlgs$.*

Since we are interested in considering failure-prone physical nodes, given a physical layer algorithm, the physical layer is then the composition of RW and *Pbcast* with *Fail*-transformed programs for all the physical nodes, as indicated by the physical layer algorithm.

Definition 6.15 *Let $palg$ be an element of $PAlgs$.*

- $PLNodes[palg]$*, the $Fail$-*transformed physical nodes parameterized by $palg$*, is the composition of $Fail(palg(p))$ for all $p \in P$.*
- $PLayer[palg]$*, the* physical layer parameterized by $palg$*, is the composition of $PLNodes[palg]$ with $RW \| Pbcast$.*

Chapter 7

Layers: Virtual Stationary Automata layer model

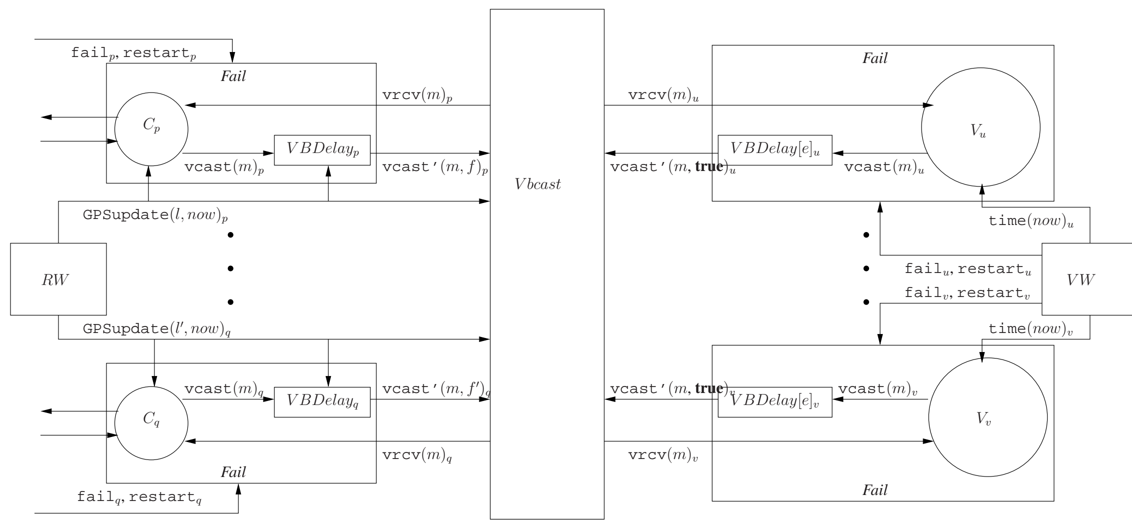


Figure 7-1: Virtual Stationary Automata layer. VSAs and clients communicate locally using *Vbcast*. VSA and client outputs may be delayed in *VBDelay* buffers. *VW* provides timing and failure information to VSAs, and *RW* provides timing and mobile node location information.

Here we describe our formal theoretical model for the virtual node abstraction layer.

The *Virtual Stationary Automata* abstraction layer [29] includes the network tiling and RW of the physical layer, client nodes that correspond to physical nodes, virtual stationary automata (VSAs) at predefined regions of the network, a *VW* time and failer service for

```
  Signature:                                                        Transitions:                                      16
2   Output time(t)_u, t ∈ ℝ^{≥0}, u ∈ U                                Output time(t)_u
    Output fail_u, u ∈ U                                               Precondition:                                  18
4   Output restart_u, u ∈ U                                              t = now
                                                                     Effect:                                        20
6 State:                                                                 last(u) ← t
    analog now: ℝ^{≥0}, initially 0                                                                                 22
8   last(u): ℝ^{≥0} ∪ {⊥}, for each u ∈ U, initially ⊥                Output fail_u
                                                                     Precondition:                                  24
10 Trajectories:                                                         None
    evolve                                                           Effect:                                        26
12    d(now) = 1                                                         None
    stop when                                                                                                       28
14    ∃u ∈ U: last(u) ∈ {⊥, now -ε_sample}                            Output restart_u
                                                                     Precondition:                                  30
                                                                         None
                                                                     Effect:                                        32
                                                                         None
```

Figure 7-2: *VW[ϵ_{sample}]*, Virtual time and failer service.

VSAs, and a local broadcast service between client nodes and VSAs, called *Vbcast*, similar to the *Pbcast* of the physical layer (see Figure 7-1). In addition, the abstraction layer contains *VBDelay* buffers that delay the broadcasts of clients and VSAs.

In the rest of this chapter, we describe all of the above components in more detail. The entire VSA layer is then defined to be just the composition of RW, VW, and $Vbcast$ together with the *Fail*-transform for each client and VSA of the composition of that process's machine with its corresponding $VBDelay$ buffer.

7.1 Network tiling and *RW*

The network tiling, describing the geography of the network, is the same as in Section 6.1.

The reliable location and time oracle RW is the same as in Section 6.3.

7.2 Virtual time and failer service *VW*

The virtual time and failer service, VW serves both as a time oracle for VSAs and a fail and restart service for VSAs. A TIOA description of VW is in Figure 7-2. Similar to RW for clients, VW performs a $\mathsf{time}(t)_u$ output at time 0 and at least every ϵ_{sample} time for each $u \in U$. Also, VW nondeterministically issues fail_u and $\mathsf{restart}_u$ outputs for each $u \in U$.

Reachable states of *VW*

Here we characterize the reachable states of *VW* by providing a list of properties exactly describing those states.

Definition 7.1 *Define Inv_{VW} to be the set of states x of VW such that the following properties hold:*

1. $\forall u \in U, now \neq 0 \Rightarrow last(u) \neq \bot$.
   This means that after time 0, there must be a non-$\bot$ time stored for each $u \in U$.

2. $\forall u \in U, last(u) \neq \bot \Rightarrow last(u) \in [now - \epsilon_{sample}, now]$.
   This means that for any $u \in U$, any non-$\bot$ $last(u)$ stores a time at most ϵ_{sample} old and no larger than now.

We do not show that Inv_{VW} describes the set of reachable states of VW since it is trivial.

7.3 Mobile client nodes

Here we describe our model of the mobile client nodes; this model is very similar to the model for P_p in Section 6.2.

- For each p in the set of physical node ids P, we assume a mobile client node C_p from a set of TIOAs, $CProgram_p$.

As for P_p, C_p has a local clock variable, $clock$ that progresses at the rate of real-time, and is initially $\bot$. As before, additional arbitrary local non-*failed* variables are allowed. Its external interface is also assumed to at least include the GPSupdate inputs of P_p, as well as vcast$(m)_p$ outputs and vrcv$(m)_p$ inputs, corresponding to bcast and brcv actions at P_p. Additional arbitrary non-fail and non-restart actions are again allowed.

```
1  Signature:
     Input time(t)_u, t ∈ ℝ^≥0
3    Input vrcv(m)_u, m ∈ Msg
     Output vcast(m)_u, m ∈ Msg
5    Arbitrary additional non-fail, non-restart internal actions.

7  State:
     analog clock: ℝ^≥0 ∪ {⊥}, initially ⊥
9    Finite set of additional non-failed variables, each initially
       set to a unique initial value.

   Trajectories:                                                  12
     if clock ≠ ⊥ then
       d(clock) = 1                                               14
     else constant clock
     Additional variables develop as specified.                   16

   Transitions:                                                   18
     Input time(t)_u
     Effect:                                                      20
       if clock ≠ t then
         Optional state changes may occur.                        22
       clock ← t
                                                                  24
     Additional transitions as allowed by the signature.
     Changes to clock are not permitted in non-time transitions.  26
```

Figure 7-3: V_u.

7.4 Virtual Stationary Automata (VSAs)

Here we describe VSAs. A VSA is a clock-equipped abstract virtual machine V_u associated with a region u in the network.

- For each u in the set of region identifiers U, we assume an abstract virtual machine V_u from a set of TIOAs, $VProgram_u$.

We provide an outline of the allowable structure of V_u in Figure 7-3. Each abstract virtual machine V_u has a local clock variable $clock$. We assume that each node's $clock$ progresses at the rate of real-time, and is initially $\perp$ before being updated with a time input.

We assume that an abstract virtual machine V_u has only the following external interface, which consists of the ability to receive time updates and broadcast and receive messages.

- **Input** $\mathsf{time}(t)_u, t \in \mathbb{R}^{\geq 0}$:

  This input reports the current time t. We require that in the state that results from this input, node u's $clock$ equals t. Also, we require that no other state changes occur unless $clock$ was not equal to t when the action occurred.

- **Output** $\mathsf{vcast}(m)_u, m \in Msg$:

  A node u may broadcast a message through $\mathsf{vcast}(m)_u$.

- **Input** $\mathsf{vrcv}(m)_u, m \in Msg$:

  A node u receives a message m through $\mathsf{vrcv}(m)_u$.

```
   Signature:                                               Transitions:                                        14
2    Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^≥0                  Input GPSupdate(l, t)_p
     Input vcast(m)_p, m ∈ Msg                                 Effect:                                          16
4    Output vcast′(m, f)_p, m ∈ Msg, f ∈ Bool                    to_send⁻ ← to_send⁺
                                                                 to_send⁺ ← λ                                   18
6  State:                                                        updated ← true
     to_send⁺, to_send⁻: Msg*, initially λ                                                                      20
8    updated: Bool, initially false                            Input vcast(m)_p
                                                               Effect:                                          22
10 Trajectories:                                                 if updated then
     stop when                                                     to_send⁺ ← append(to_send⁺, m)              24
12     to_send⁺ ≠ λ ∨ to_send⁻ ≠ λ
                                                               Output vcast′(m, f)_p                            26
                                                               Precondition:
                                                                 m = head(to_send⁻ to_send⁺) ∧ (f ⇔ to_send⁻ = λ) 28
                                                               Effect:
                                                                 if f then                                      30
                                                                   to_send⁺ ← tail(to_send⁺)
                                                                 else to_send⁻ ← tail(to_send⁻)                 32
```

Figure 7-4: $\mathit{VBDelay}_p$, Message delay service for clients.

We allow additional arbitrary non-*failed* variables and non-fail and non-restart internal actions. We also require that each action be deterministic, in that for each state s and each action a of V_u, there exists at most one state s' such that (s, a, s') is a transition of V_u.

7.5 *VBDelay* delay buffers

As mentioned previously, there are outbound delay buffers from clients and VSAs to the broadcast service *Vbcast*. For each client or VSA node, its associated *VBDelay* buffer takes as input the vcast(m) outputs of the node, tags each message m with a Boolean that is later used by the *Vbcast* service to help determine what region the node was in when the message was produced, and passes the tagged message on to the *Vbcast* service. In this section we first describe the client delay buffer, and then the VSA delay buffer.

7.5.1 Client *VBDelay*

The delay buffer for a client, $\mathit{VBDelay}_p, p \in P$ (see Figure 7-4), tags vcast messages from the client with a Boolean indicating if the message was submitted before the latest GPSupdate at the client, and submits the tagged message to the *Vbcast* service before any time passes. (Hence, the delay buffer has a delay of time 0.) Its state consists of the

following variables:

- $to_send^+ \in Msg^*$: This is a queue of messages to be passed on to *Vbcast*. It is initially empty.
- $to_send^- \in Msg^*$: This is also a queue of messages to be passed on to *Vbcast*. It contains messages that were submitted before the latest GPSupdate at the client. It is initially empty.
- $updated : Bool$: This is a Boolean indicating whether the node has experienced a GPSupdate since starting. It is initially false.

Its interface consists of the following three kinds of actions:

- **Input** GPSupdate$(l, t)_p, l \in R, t \in \mathbb{R}^{\geq 0}, p \in P$: This input indicates that process p is at location l, and results in the process moving its to_send^+ messages to to_send^- and then clearing to_send^+. It also updates $updated$ to true.
- **Input** vcast$(m)_p, m \in Msg, p \in P$: This input is a broadcast of a message m, resulting in the addition of m to to_send^+.
- **Output** vcast$'(m, f)_p, m \in Msg, f \in Bool, p \in P$: This output is the passing on of a vcast message to *Vbcast*. The Boolean f indicates whether the message was submitted to the process after its last GPSupdate.

When a vcast of a message occurs at a process that has received at least one GPSupdate, the message is appended to a local queue to_send^+ of messages the sender wants to communicate to other processes (lines 21-24). If no GPSupdate has occurred, the message is dropped. Whenever a GPSupdate occurs at the client, the queue to_send^- is overwritten with to_send^+, to_send^+ is erased, and $updated$ is set to true (lines 15-19). (Notice that, with our Section 6.3 restriction that a GPSupdate occurs at most once per time per particular client, the queue to_send^- will generally be empty at the time of a GPSupdate.) Whenever $to_send^- \cdot to_send^+$ is not empty (line 12), the first message in $to_send^- \cdot to_send^+$ is removed, tagged with a Boolean f equal to true if to_send^- is empty and false otherwise (line 28), and output via vcast$'$, passing the tagged message on to the *Vbcast* service (lines 26-32).

```
  Signature:                                              Transitions:
2   Input vcast(m)_u, m ∈ Msg                               Input vcast(m)_u                                  16
    Output vcast′(m, true)_u, m ∈ Msg                       Effect:
4                                                             to_send ← append(to_send, ⟨m, rtimer⟩)          18
  State:
6   analog rtimer: ℝ≥0, initially 0                         Output vcast′(m, true)_u                          20
    to_send: (Msg ×ℝ≥0)*, initially λ                       Precondition:
8                                                             ∃t ∈ ℝ≥0: ⟨m, t⟩ = head(to_send)                22
  Trajectories:                                             Effect:
10  evolve                                                    to_send ← tail(to_send)                         24
      d(rtimer) = 1
12  stop when
      ∃⟨m, t⟩ ∈ to_send: rtimer -t ∉ [0, e)
```

Figure 7-5: *VBDelay[e]*$_u$, Message delay service for VSAs.

7.5.2 VSA *VBDelay*

For each VSA, the delay buffer is slightly simpler than that of a client in that the VSA always knows its region (it does not receive GPSupdate inputs), and is slightly more complicated in that it does not have to immediately forward outgoing messages. Instead, *VBDelay*$_u$ is parameterized by the following constant:

- $e : \mathbb{R}^{\geq 0}$, a maximum output delay time.

VBDelay for a VSA is almost the same as *VBDelay* for a client, except that the Boolean attached to each message is always set to true, and when *VBDelay*$_u$ receives a vcast$(m)_u$ input, it saves the message and the local time in the local to_send queue (lines 16-18) for some nondeterministically-chosen time in $[0, e]$ (enforced by the stopping condition on line 13), and then broadcasts the message through vcast′(m, true) (lines 20-24).

Any program written for the VSA layer must take into account e, as it would message delay.

7.6 *Vbcast*: Virtual local broadcast service

Each client and virtual node has access to the virtual local broadcast communication service *Vbcast*, modelled in Figure 7-6. The service is parameterized with the following:

- $d : \mathbb{R}^{\geq 0}$, the message delay upper bound. We require that $d > d_{phys}$.

```
1  Signature:
     Input GPSupdate(l, t)_p, l ∈ R, p ∈ P, t ∈ ℝ^≥0
3    Input vcast'(m, f)_i, m ∈ Msg, f ∈ Bool, i ∈ P ∪ U
     Output vrcv(m)_j, m ∈ Msg, j ∈ P ∪ U
5    Internal drop(n, j), n ∈ ℕat, j ∈ P ∪ U

7  State:
     analog now: ℝ^≥0, initially 0
9    reg(p), oldreg(p):U∪ {⊥}, for each p ∈ P, initially ⊥
     vbcastq: (Msg ×U ×ℝ^≥0 ×2^(P∪U))*, initially λ
11
   Trajectories:
13   evolve
       d(now) = 1
15   stop when
       ∃⟨m, u, t, P'⟩ ∈ vbcastq: [t = now -d ∧ P' ≠ ∅ ]
17
   Transitions:
19   Input GPSupdate(l, t)_p
     Effect:
21     oldreg(p) ← reg(p)
       reg(p) ← region(l)

   Input vcast'(m, f)_i                                          24
   Effect:
     if i ∈ U then                                               26
       vbcastq ← append(vbcastq, ⟨m, i, now, P ∪ U⟩)
     else if (f ∧ reg(p) ≠ ⊥) then                               28
       vbcastq ← append(vbcastq, ⟨m, reg(p), now, P ∪ U⟩)
     else if (¬ f ∧ oldreg(p) ≠ ⊥) then                          30
       vbcastq ← append(vbcastq, ⟨m, oldreg(p), now, P ∪ U⟩)
                                                                 32
   Output vrcv(m)_j
   Local:                                                        34
     n ∈ [1, . . ., |vbcastq| ], u: U, t: ℝ^≥0, P': 2^(P∪U)
   Precondition:                                                 36
     vbcastq[n ]= ⟨m, u, t, P'⟩ ∧ j ∈ P' ∧ t ≠ now
   Effect:                                                       38
     vbcastq[n ] ← ⟨m, u, t, P' -{j}⟩
                                                                 40
   Internal drop(n, j)
   Local:                                                        42
     m: Msg, u: U, t: ℝ^≥0, P': 2^(P∪U)
   Precondition:                                                 44
     vbcastq[n ] = ⟨m, u, t, P'⟩ ∧ j ∈ P' ∧ t ≠ now
     (j ∈ P ∧ reg(j) ∉ nbrs+(u)) ∨ (j ∈ U ∧ j ∉ nbrs+(u))         46
   Effect:
     vbcastq[n ] ← ⟨m, u, t, P' -{j}⟩                            48
```

Figure 7-6: *Vbcast[d]*.

The service described in Figure 7-6 takes each $\mathsf{vcast}'(m, f)_i$ input from client and virtual node delay buffers and delivers the message m via $\mathsf{vrcv}(m)$ at each client or virtual node that is in some region u or a neighboring region for the d time after broadcast of the message. If the vcast′ was from a VSA at region i, then the region u is equal to i. Otherwise, if the vcast′ was from a client, we use the Boolean tag f to determine the region u; if f is true then region u is the region of i when the vcast′ occurs, and if f is false then region u is the region of i before the last GPSupdate at i occurred.

Vbcast's interface consists of the following three kinds of actions:

- **Input** $\mathsf{GPSupdate}(l, t)_p, l \in R, t \in \mathbb{R}^{\geq 0}, p \in P$: This input indicates that process p is at location l, and results in the update of records storing a client's last two regions.

- **Input** $\mathsf{vcast}'(m, f)_i, m \in Msg, f \in Bool, i \in P \cup U$: This input is a broadcast of a message m by some node i where i is either the id of a client or a VSA. The Boolean f indicates for clients whether the client's last GPSupdate occurred before the client vcast the message.

- **Output** $\mathsf{vrcv}(m)_j, m \in Msg, j \in P \cup U$: This output represents the delivery of a message m at process j.

The state variables are:

- $now : \mathbb{R}^{\geq 0}$: This variable is a real-time clock. It is initially 0.
- $reg(p) : U \cup \{\perp\}$ for $p \in P$: This is the region of each client. It is initially $\perp$ for each $p \in P$, and is set whenever a $\mathsf{GPSupdate}_p$ occurs.
- $oldreg(p) : U \cup \{\perp\}$ for $p \in P$: This is the region of each client before the client's last GPSupdate. It is initially $\perp$ for each $p \in P$, and is updated to the old value of $reg(p)$ whenever a $\mathsf{GPSupdate}_p$ occurs.
- $vbcastq : (Msg \times U \times \mathbb{R}^{\geq 0} \times 2^{P \cup U})^*$: This is the record of all outstanding vcast′ events, structured as an initially empty queue of tuples. Each tuple consists of a vcast′ message and its attached region, the time at which the message was input, and a set of ids of nodes (clients and VSAs) for which the message has not yet been delivered or lost.

The main variable of this service is $vbcastq$, storing information about all previous virtual broadcasts. When a $\mathsf{vcast}'(m, f)_i$ input occurs at some time t, the action first calculates a region u to associate with the message. Region u is equal to i if i is a region id (lines 26-27). If i is a client id and f is true, then u is set to $reg(p)$. If i is a client id and f is false, then u is set to $oldreg(p)$. If u is $\perp$ the message is dropped, otherwise the tuple $\langle m, u, now, P \cup U\rangle$ is then appended to $vbcastq$. The set of ids in the tuple represents the set of processes that might still potentially receive the message. This set starts as all mobile node and region ids. Some positive amount of time after the broadcast (guaranteed by the precondition that $t \neq now$ on line 37 and 45), a process j in the set can either receive the message (lines 33-39), or if j is a mobile node id with a region not equal to or neighboring u or if j is a region id not equal to or neighboring u, the transmission may fail to j (lines 41-48). In either case, the id j is removed from the set of ids of processes that might still receive the message. We require that once a message is broadcast, for every node the message is received or the transmission fails by at most d time later (guaranteed by the stopping

condition expressed on line 16). Our requirement that a non-0 amount of time pass between broadcast and the possible receiving or dropping of the message is utilized later to prevent race conditions that can result when a process changes regions or failure modes.

Properties of *Vbcast*

The service guarantees that in each execution α of *Vbcast*, there exists a function mapping each $\mathsf{vrcv}(m)_j$ event to a $\mathsf{vcast}'(m, f)_i$ event such that the following hold:

- *Integrity*: If a vrcv event π is mapped to a vcast' event π', then π' occurs before π.

- *Non-duplicative delivery*: If a $\mathsf{vrcv}(m)_j$ event π is mapped to a vcast' event π', then there do not exist any other $\mathsf{vrcv}(m)_j$ events that map to π'.

- *Bounded-time delivery*: If a vrcv event π is mapped to a vcast' event π' where π' occurs at time t, then event π occurs in the interval $(t, t + d]$.

- *Reliable local delivery*: This guarantees that a transmission will be received by nearby nodes: Say a $\mathsf{vcast}'(m, f)_i$ event π' occurs at time t and $\alpha.ltime > t + d$. Let u be i if $i \in U$, otherwise be $reg^-(i, t)$ if f is false or $reg(i)$ at the time of π' if f is true. If u is not $\perp$, then for each $j \in P \cup U$ such that either $j \in P$ and $reg^+(j, t') \in nbrs^+(u)$ for all t' in the interval $[t, t+d]$ or $j \in nbrs^+(u)$, there exists a $\mathsf{vrcv}(m)_j$ event π such that π is mapped to π'.

The *Vbcast* service is very similar to the *Pbcast* service described in Section 6.4. The most obvious difference is that the *Vbcast* service is extended to a larger id set, consisting of region ids as well as physical node ids. Comparing the guarantees for both services, we also note that the *Non-duplicative delivery* property and the *Reliable local delivery* property are both slightly different.

The *Non-duplicative delivery* property of *Vbcast* says that at most one vrcv event at a particular process is mapped to a single vcast' event. The property in *Pbcast* says something more stringent, namely that if a $\mathsf{brcv}(m)_q$ event is mapped to a $\mathsf{bcast}(m)_p$ event at time t, then there are no other $\mathsf{brcv}(m)_q$ events that map to any $\mathsf{bcast}(m)_p$ event at time t

for the same m and p. With the more restrictive non-duplication property of *Pbcast* we can easily build a *Vbcast* service with this more common definition of non-duplicative delivery.

The *Reliable local delivery* property of *Vbcast* differs in that it is expressed in terms of regions, unlike in *Pbcast* where it is expressed in terms of locations. Here, we require that messages that originate from some region u be received by all nodes that are in region u or neighboring regions for the transmission period. For *Pbcast*, we require that messages that originate from some location l be received by all nodes within some distance of l for the transmission period.

Reachable states of *Vbcast[d]*

Here we characterize the reachable states of *Vbcast[d]* by providing a list of properties exactly describing those states.

Definition 7.2 *Define Inv_{Vbcast} to be the set of states x of $Vbcast$ such that the following properties hold:*

1. $\forall p \in P, oldreg(p) \neq \bot \Rightarrow reg(p) \neq \bot$.
   This means that for each $p \in P$, the value of $reg(p)$ can only be $\bot$ if $oldreg(p)$ is $\bot$.

2. $\forall \langle m, u, t, P' \rangle \in vbcastq, t \leq now \wedge (P' \neq \emptyset \Rightarrow t \geq now - d) \wedge (t = now \Rightarrow P' = P \cup U)$.
   This means that for each message tuple in $vbcastq$, the timestamp of the message is not after now, if there are still processes that have not either lost or delivered the message then the message is no older than d, and if the message was sent at this time then P' is full.

3. *The tuples in $vbcastq$ are in order of their timestamp.*

We do not show that Inv_{Vbcast} describes the set of reachable states of $Vbcast$ since it is trivial.

7.7 V-algorithms and *VLayers*

Here we provide definitions for a VSA layer algorithm and a complete VSA layer.

A VSA layer algorithm is just an assignment of a TIOA program to each client and VSA.

Definition 7.3 *A V-algorithm,* $alg : P \cup U \rightarrow CProgram \cup VProgram$*, is a mapping such that for each* $p \in P$*,* $alg(p) \in CProgram_p$ *and for each* $u \in U$*,* $alg(u) \in VProgram_u$*. The set of all V-algorithms is referred to as* $VAlgs$*.*

Since we are interested in providing this layer using failure-prone physical nodes, we then define a *VLayer*, a VSA layer with failure-prone clients and VSAs. Given a VSA layer algorithm alg, a fail-transformed node (either a client or a VSA) of the VSA layer is the *Fail*-transformed version of the composition of the TIOA for the node as indicated by alg with the node's delay buffer. The *VLayer* is then the composition of $RW\|VW\|Vbcast$ with all the fail-transformed nodes of the VSA layer.

Definition 7.4 *Let* alg *be an element of* $Valgs$*.*

- $VLNodes[alg]$*, the* fail-transformed nodes of the VSA layer parameterized by alg*, is the composition of* $Fail(VBDelay_i\|alg(i))$ *for all* $i \in P \cup U$*.*
- $VLayer[alg]$*, the* VSA layer parameterized by alg*, is the composition of* $VLNodes[alg]$ *with* $RW\|VW\|Vbcast$*.*

Reachable states of *RW*||*VW*||*Vbcast*

Here we characterize the reachable states of *RW*||*VW*||*Vbcast* by providing a list of properties exactly describing those states.

Definition 7.5 *Define* $Inv_{\mathrm{RW\|VW\|Vbcast}}$ *to be the set of states* x *of* $RW\|VW\|Vbcast$ *such that the following properties hold:*

1. $x \lceil X_{Vbcast} \in Inv_{Vbcast} \wedge x \lceil X_{RW} \in Inv_{RW} \wedge x \lceil X_{VW} \in Inv_{VW}$.
   This says that a state of the composition restricted to the individual components is in the corresponding set of reachable states for that component.

2. $RW.now = VW.now = Vbcast.now$.
   This says that the clock values of the components are the same.

3. $\forall p \in P$, $RW.reg(p) = Vbcast.reg(p)$.
   This says that the region for a process matches between $Vbcast$ *and* RW.

4. $\forall p \in P$, *if* $|RW.updates(p)| > 1$ *then let* $\langle u_p, t_p \rangle$ *be the tuple with second highest* t_p *in* $RW.updates(p)$*, else let* u_p *be* $\perp$*. Then* $Vbcast.oldreg(p) = u_p$.
   This says that the $oldreg(p)$ *for any* $p \in P$ *matches the region associated with the next-to-last* GPSupdate *at process* p.

We do not show that $Inv_{RW\|VW\|Vbcast}$ describes the set of reachable states of *RW*$\|$*VW*$\|$*Vbcast* since it is trivial.

Chapter 8

VSA layer emulations

Here we describe what it means for a mapping from V-algorithms to P-algorithms to be an emulation algorithm for the VSA layer, using the language and theory of Chapter 4. If such a mapping is an emulation algorithm for the VSA layer, then an application programmer could write programs for the VSA layer and then run those programs on the physical layer.

First we define the concepts of an emulation and a stabilizing emulation of a VSA layer. Then we conclude that a stabilizing emulation of a self-stabilizing VSA layer program has traces that eventually look like those of the VSA layer program starting in some legal state. This separates the reasoning about stabilization properties of a VSA layer emulation algorithm from those of the VSA layer program.

We define an emulation algorithm $amap$ of the VSA layer to be a function mapping V-algorithms to P-algorithms, where for any alg in $VAlgs$, a trace of $PLNodes[amap[alg]]$ composed with *RW*$\|$*Pbcast* is related to some trace of $VLNodes[alg]$ composed with $RW\|VW\|Vbcast$. For a particular alg, $amap[alg]$ could be defined so that each physical node's program is a composition of the client program in the VSA layer for that node, and a VSA emulator portion where the physical node helps emulate its current region's VSA.

First, for use throughout this thesis, we introduce two pieces of notation that describe actions to be hidden in the physical layer and the virtual layer:

Definition 8.1 *Define* H_{PL} *to be* $\{\mathsf{bcast}(m)_p, \mathsf{brcv}(m)_p \mid m \in Msg, p \in P\}$.

Definition 8.2 *Define* H_{VL} *to be* $\{\mathsf{vcast}(m)_i, \mathsf{vrcv}(m)_i, \mathsf{vcast}'(m, f)_i, \mathsf{time}(t)_u, \mathsf{fail}_u, \mathsf{restart}_u \mid m \in Msg, f \in Bool, t \in \mathbb{R}^{\geq 0}, i \in P \cup U, u \in U\}$.

Now we can define our concepts of VSA layer emulation.

Definition 8.3

- *Let* $amap$ *be a function of type* $VAlgs \rightarrow PAlgs$*, and let* t *be in* $\mathbb{R}^{\geq 0}$.
- *Let* $\mathcal{PL}$ *be* $\{PLNodes[amap[alg]] \mid alg \in VAlgs\}$.
- *Let* $\mathcal{VL}$ *be* $\{VLNodes[alg] \mid alg \in VAlgs\}$.
- *Let* emu *be the function of type* $\mathcal{VL} \rightarrow \mathcal{PL}$ *such that for each* $alg \in VAlgs$*,* $emu(VLNodes[alg]) = PLNodes[amap[alg]]$.
- *Let* S *be a function that maps each element* V *of* $\mathcal{PL}$ *to a suffix-closed subset of* $frags_{\mathsf{ActHide}(H_{VL}, V \| RW \| VW \| Vbcast)}$.

Then we define the following two terms:

1. $amap$ *is an* S-constrained VSA layer emulation algorithm *if* $(\mathcal{PL}, RW \| Pbcast, H_{PL})$ *emulates* $(\mathcal{VL}, RW \| VW \| Vbcast, H_{VL})$ *constrained to* S *with* emu.

   Recall from Definition 4.1 that this means that emu *maps each element* VL *of* $\mathcal{VL}$ *to an element* PL *of* $\mathcal{PL}$ *such that each trace of* $PL \| RW \| Pbcast$ *with actions in* H_{PL} *hidden is a trace of an execution of* $VL \| RW \| VW \| Vbcast$ *with actions in* H_{VL} *hidden that also happens to be in* $S(VL)$.

2. $amap$ *is an* S-constrained t-stabilizing VSA layer emulation algorithm *if* $(\mathcal{PL}, RW \| Pbcast, H_{PL})$ *emulation stabilizes in time* t *to* $(\mathcal{VL}, RW \| VW \| Vbcast, H_{VL})$ *constrained to* S *with* emu.

We can now combine a stabilizing VSA layer emulation with a self-stabilizing VSA layer algorithm and conclude that the appropriately restricted traces of the result stabilize to appropriately restricted trace fragments of the VSA layer algorithm started from legal states of that algorithm. This is a simple corollary of Theorem 4.7.

Corollary 8.4 *1. Let $amap$ be an S-constrained t_1-stabilizing VSA layer emulation algorithm.*

2. Let $alg \in VAlgs, t_2 \in \mathbb{R}^{\geq 0}$, and legal set L for $VLayer[alg]$ be chosen so that $VLNodes[alg]$ self-stabilizes to L relative to $R(RW\|VW\|Vbcast)$ in time t_2.

Then $traces_{\mathsf{ActHide}(H_{PL}, U(PLNodes[amap[alg]])\|R(RW\|Pbcast))}$ stabilizes in time $t_1 + t_2$ to $\{trace(\alpha) \mid \alpha \in execs_{\mathsf{ActHide}(H_{VL}, Start(VLayer[alg], L))} \cap S(VLNodes[alg])\}$.
In other words, consider the composition of $RW\|Pbcast$ started in a reachable state with $PLNodes[amap[alg]]$ (the physical nodes running an emulation of the virtual layer program alg) started in an arbitrary state. Hide the actions in H_{PL}. The set of traces of the resulting machine stabilizes in time $t_1 + t_2$ (the time for the VSA layer emulation to stabilize, followed by the time for the virtual layer program to stabilize to legal set L) to the set of traces of executions allowed by S of the virtual layer started in legal set L, after hiding actions in H_{VL}.

Part II

VSA layer emulation algorithm

Part II describes an implementation of the VSA layer using the underlying mobile ad-hoc system, and proves that the implementation provides a stabilizing emulation of the VSA programming layer. This implementation is in three parts: totally ordered broadcast, leader election, and a main emulation component.

Chapter 9 is where I describe the totally ordered broadcast service. It is useful to have access to a totally ordered broadcast service that allows nodes in the same region to receive the same sets of messages in the same order. The totally ordered broadcast service is intended to allow a non-failed node p that knows it is in some region u to broadcast a message m, via tocast$(m)_p$, and to have the message be received exactly $d, d > d_{phys}$, time later via torcv$(m)_q$, by nodes that are in region u or a neighboring region for at least d time.

In Chapter 10, I describe the leader election service that allows nodes in the same region to periodically compete to be named sole leader of the region for some time. Our leader election service is a round-based service that collects information from potential leaders at the beginning of each round, determines up to one leader per region, and performs leader outputs for those leaders that remain alive and in their region for long enough.

Finally, in Chapter 11, I describe a fault-tolerant implementation of each VSA by mobile nodes in its region of the network, and prove that the implementation gives us a stabilizing emulation of the VSA layer. At a high level, the individual mobile nodes in a region share emulation of the virtual machine through a deterministic state replication algorithm while also being coordinated by a leader. Each mobile node runs its portion of the totally ordered broadcast service, leader election service, and a Virtual Node Emulation ($VSAE$) algorithm, for each virtual node.

Chapter 9

Totally ordered broadcast service

In order to simplify later algorithms, it is useful to have access to a totally ordered broadcast service that allows nodes in the same region to receive the same sets of messages in the same order. The totally ordered broadcast service is intended to allow a non-failed node p that knows it is in some region u to broadcast a message m, via $\mathsf{tocast}(m)_p$, and to have the message be received exactly $d, d > d_{phys}$, time later via $\mathsf{torcv}(m)_q$, by nodes that are in region u or a neighboring region for at least d time. In this chapter, we start by introducing a specification for the service. We then show how to implement this service using the physical layer. Finally, we show that our implementation is correct and that it is self-stabilizing.

9.1 *TOBspec*: Specification of totally ordered broadcast

We describe the specification of totally ordered broadcast (see Figure 9-1) in three parts: *TObcast*, *TOBDelay*$_p$, and *TOBFilter*$_p$, for each $p \in P$. The specification of the totally ordered broadcast service is then *TOBspec*, which is equal to *TObcast*$\|RW$ composed with $Fail(TOBDelay_p \| TOBFilter_p)$ for all $p \in P$, with certain actions hidden.

TObcast is the main message ordering and delivery service, taking inputs of message and Boolean pairs, tagging the message with a region u calculated based on the Boolean and the GPSupdate history of the sender of the message, and holding the region-tagged messages for exactly d time before delivering $\langle m, u\rangle$ at each process that has been in region

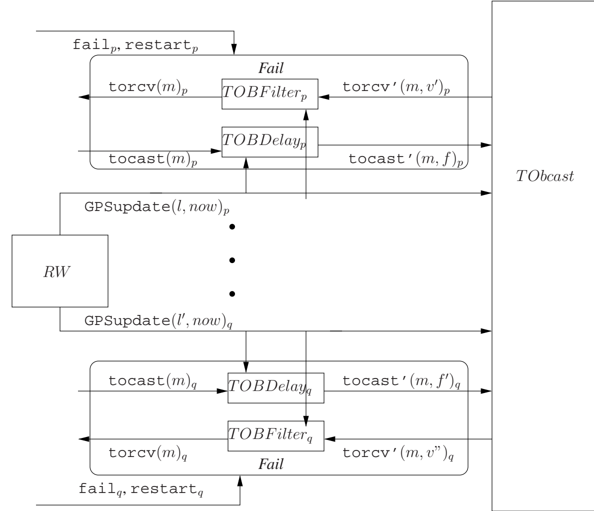


Figure 9-1: Totally ordered broadcast service. Client outputs may be delayed in *TOBDelay* buffers, and messages are filtered out based on region and time alive information in *TOBFilter* buffers. *RW* provides timing and mobile node location information.

u or a neighboring region for the appropriate time. The order of these deliveries at each process is consistent with a global ordering of all broadcast inputs to *TObcast*.

TOBDelay$_p$ is an outgoing delay buffer that sits between process p and *TObcast*, taking inputs of messages to be sent via the totally ordered broadcast service from the process, tagging each with a Boolean indicating if the message was submitted to the automaton since the client's last GPSupdate, and submitting the tagged messages to *TObcast*. This mechanism is similar to the one used in $VBDelay$ (see Figure 7-4) for the virtual layer.

TOBFilter$_p$ also sits between *TObcast* and a user of the *TObcast* service, but in the opposite direction. When *TObcast* delivers a message tagged with some region u to *TOBFilter*$_p$, *TOBFilter*$_p$ determines whether or not process p received a GPSupdate after starting and at least d time ago, and if so passes the message along to be received at the user. This prevents p from receiving messages that it was not alive and in the region long enough to receive.

Notice that the *TOBDelay* and *TOBFilter* machines are for individual processes. In this thesis we are interested in considering *Fail*-transformed mobile nodes. In the presence of process failures, it is apparent that allowable traces of the totally ordered broadcast service will be dependent on the history of the fails and restarts of a mobile node. Separating

```
1  Signature:
     Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^≥0
3    Input tocast′(m, f)_p, m ∈ Msg, f ∈ Bool, p ∈ P
     Output torcv′(m, u)_p, m ∈ Msg, u ∈ U, p ∈ P
5    Internal drop(p), p ∈ P

7  State:
     analog now: ℝ^≥0, initially 0
9    updates(p): 2^(U×ℝ^≥0), for each p ∈ P, initially ∅
     procs: 2^P, initially P
11   sent, oldsent: (Msg × U × P × ℝ^≥0)*, initially λ

13 Derived variables:
     reg⁻(p: P): U ∪ {⊥}
15     if ∃⟨u, t⟩ ∈ updates(p): t < now then
         return min({u ∈ U | ∃t′ < now: ⟨u, t′⟩ ∈ updates(p)
17           ∧ ∀⟨u*, t*⟩ ∈ updates(p): (t* ≤ t ∨ t* = now)})
       else return ⊥
19
     reg(p: P): U ∪ {⊥}
21     if ∃u ∈ U: ⟨u, now⟩ ∈ updates(p) then
         return min({u ∈ U | ⟨u, now⟩ ∈ updates(p)})
23     else return reg⁻(p)

25   regSpan(p: P, r: U, t: ℝ^≥0): Bool
       return ∃⟨u, t′⟩ ∈ updates(p): [t′ ≤ t
27         ∧ ∀⟨v, t′′⟩ ∈ updates(p): (t′′ ≥ t′ ⇒ r ∈ nbrs⁺(v))]

29 Trajectories:
     evolve
31     d(now) = 1
     stop when
33     ∃⟨m, u, p, t⟩ = head(sent): t = now -d

   Transitions:
   Input GPSupdate(l, t)_p                                          36
   Effect:
     updates(p) ← updates(p) ∪ {⟨region(l), t⟩}                     38

   Input tocast′(m, f)_p                                            40
   Effect:
     choose i ∈ {j ∈ [0, |sent|] | ∀k ∈ (j, |sent|]:                42
           ∃⟨m′, u′, p′, t′⟩ = sent(k): (p′ ≠ p ∧ t′ = now)}
     if (f ∧ reg(p) ≠ ⊥) then                                       44
       sent ← insert(sent, ⟨m, reg(p), p, now⟩, i)
     if (¬ f ∧ reg⁻(p) ≠ ⊥) then                                    46
       sent ← insert(sent, ⟨m, reg⁻(p), p, now⟩, i)
                                                                    48
   Output torcv′(m, u)_p
   Precondition:                                                    50
     ∃q ∈ P: ⟨m, u, q, now -d⟩ = head(sent) ∧ p ∈ procs
     regSpan(p, u, now -d)                                          52
   Effect:
     procs ← procs -{p}                                             54
     if procs = ∅ then
       oldsent ← append(oldsent, head(sent))                        56
       sent ← tail(sent)
       procs ← P                                                    58

   Internal drop(p)                                                 60
   Local:
     m: Msg, u: U                                                   62
   Precondition:
     ∃q ∈ P: ⟨m, u, q, now -d⟩ = head(sent) ∧ p ∈ procs             64
     ¬ regSpan(p, u, now -d)
   Effect:                                                          66
     procs ← procs -{p}
     if procs = ∅ then                                              68
       oldsent ← append(oldsent, head(sent))
       sent ← tail(sent)                                            70
       procs ← P
```

Figure 9-2: *TObcast[d]*, Message ordering service.

the *TOBDelay* and *TOBFilter* machines from *TObcast* allows us to *Fail*-transform portions of *TOBspec*. This separation makes it easier to describe a main service component that is $Fail$-oblivious, making it easier to use $Fail$-transform related theory from Chapter 5. If a component not corresponding to a particular mobile node were to not be oblivious to mobile node failures, it would introduce complications later when we use totally ordered broadcast in conjunction with other services (in Chapter 11).

We describe the *TOBDelay, TOBFilter*, and *TObcast* pieces in more detail below.

9.1.1 *TObcast*

Here we provide a description of *TObcast* (Figure 9-2), the message ordering and region-based delivery service. The interface of *TObcast* consists of three kinds of actions:

- **Input** $\mathsf{GPSupdate}(l,t)_p, l \in R, t \in \mathbb{R}^{\geq 0}, p \in P$: This input indicates that a process p is currently located at position l.

- **Input** $\mathsf{tocast}'(m,f)_p, m \in Msg, f \in Bool, p \in P$: This input is a broadcast of a message m from a process. The Boolean f indicates whether the message was submitted to p's delay buffer after p's last GPSupdate, and is used by *TObcast* to determine the appropriate source region for the message.

- **Output** $\mathsf{torcv}'(m,u)_q, m \in Msg, u \in U, q \in P$: This output represents the delivery of a message m at process q. The message m corresponds to an earlier tocast′ message. The region u is the region of sender of the message at the time the message was tocast.

The state variables are:

- $now : \mathbb{R}^{\geq 0}$: This variable is the real-time. It is initially 0.

- $updates(p) : 2^{U \times \mathbb{R}^{\geq 0}}, p \in P$: This variable is a history of the reported regions for each process. For each $\mathsf{GPSupdate}(l,t)_p$ input, the pair $\langle region(l), t\rangle$ is stored in $updates(p)$.

- $procs : 2^P$: This is a bookkeeping variable, used to keep track of which processes have not had the first message in $sent$ delivered or dropped. It is initially P.

- $sent : (Msg \times U \times P \times \mathbb{R}^{\geq 0})^*$: This is the queue of all outstanding tocast′ events, initially empty. For each $\mathsf{tocast}'(m,f)_p$ input, a tuple $\langle m,u,p,now\rangle$ is stored in $sent$, where u is calculated based on the value of f and $updates(p)$.

- $oldsent : (Msg \times U \times P \times \mathbb{R}^{\geq 0})^*$: This is the queue of all processed tocast′ events, initially empty. Each entry in this queue was previously an entry in $sent$.

The code also uses three derived variables:

- $reg : P \to U \cup \{\bot\}$ maps a physical node id p to the region indicated by the last $\mathsf{GPSupdate}_p$. If no such region exists, the function returns $\bot$. The function is calculated in a similar manner to the reg function in Section 6.3.

- $reg^- : P \rightarrow U \cup \{\perp\}$ maps a physical node id p to the region indicated by the last $\mathsf{GPSupdate}_p$ before the current time. If no such region exists, the function returns $\perp$. The function is calculated in a similar manner to the reg^- function in Section 6.3.

- $regSpan : (P \times U \times \mathbb{R}^{\geq 0}) \rightarrow Bool$ maps a physical node id p, region id u, and time t to a Boolean indicating whether the process p was in region u or a neighboring region from the end of time t up to the current time. This is calculated by examining all the pairs in $updates(p)$ and seeing if there exists some pair with a timestamp t' no larger than t such that for each pair with a timestamp at least as large as t', the region in that pair is either u or a neighbor of u.

Whenever a $\mathsf{tocast}'(m, f)_p$ input occurs (line 40), the action calculates a region u to associate with the message. If f is true, then u is set to $reg(p)$, else u is set to $reg^-(p)$. If u is $\perp$, then the message is dropped, else *TObcast* inserts the tuple $\langle m, u, p, now \rangle$ into $sent$ (lines 44-47) at some position such that all tuples after it in $sent$ were also sent at time now and not sent by p (lines 42-43). This means that the tuples are ordered in $sent$ with respect to the real-time at which they arrived, and that tuples that originate from the same process are ordered with respect to the order in which the process submitted them.

Whenever the head tuple $\langle m, u, p, t \rangle$ of the $sent$ queue has a timestamp t equal to $now - d$, meaning the tuple was added d time ago, a $\mathsf{torcv}'(m, u)_q$ or $\mathsf{drop}(q)$ output is performed (ensured by the stopping condition on line 33) for each q in P, and the tuple is moved from $sent$ to $oldsent$. The action is $\mathsf{torcv}'(m, u)_q$ if q was in region u or a neighboring region from the end of time t until the current time (expressed in line 52 as the condition that $regSpan(q, u, t)$). The action is $\mathsf{drop}(q)$ otherwise (line 65). This prevents q from receiving the message.

Properties of *TObcast*

In each execution α of *TObcast*, there exists a function mapping each $\mathsf{torcv}'(m, u)_q$ event to a $\mathsf{tocast}'(m, f)_p$ event such that the following hold:

- *Region-based integrity*: If a $\mathsf{torcv}'(m, u)_q$ event π is mapped to a $\mathsf{tocast}'(m, f)_p$ event π' at some time t, then $(f \wedge u = reg(p)) \vee (\neg f \wedge u = reg^-(p))$ when event π'

occurs and $regSpan(q, u, t)$ is true when π occurs.

- *Non-duplicative delivery*: If a torcv'_q event π is mapped to a tocast' event π', then there do not exist any other $\mathsf{torcv}'(m)_q$ events that map to π'.

- *Exact-time delivery*: If a torcv' event π is mapped to a tocast' event π' where π' occurs at time t, then event π occurs at time $t + d$.

- *Reliable local delivery*: This guarantees that a transmission will be received by nearby nodes: Say a $\mathsf{tocast}'(m, f)_p$ event π' occurs at time t and $\alpha.ltime > t + d$. Let u be $reg(p)$ when π' occurs if f is true or $reg^-(p)$ when π' occurs otherwise. If u is not $\perp$, then for each $q \in P$ such that $regSpan(q, u, t)$ in all states of α at time $t + d$, there exists a torcv'_q event π such that π is mapped to π'.

- There exists a total order on tocast' events such that the following hold:

  - *Sender-order preservation*: For any tocast'_p events π'_1 and π'_2, if π'_1 occurs before π'_2 then $\pi'_1 < \pi'_2$.

  - *Consistent delivery order*: For any tocast' events π'_1 and π'_2 where $\pi'_1 < \pi'_2$, and any torcv' events π_1 and π_2 where π_1 maps to π'_1 and π_2 maps to π'_2, we have that π_1 occurs before π_2.

  - *No gap delivery*: Let π'_1 be a $\mathsf{tocast}'(m, f)_p$ event at time t, u be $reg(p)$ when π'_1 occurs if f is true or $reg^-(p)$ when π'_1 occurs otherwise, and π'_2 be a tocast' event such that $\pi'_1 < \pi'_2$. Let π_2 be some torcv'_q event such that π_2 maps to π'_2. If u is not $\perp$ and $regSpan(q, u, t)$ when π_2 occurs, then there exists a torcv'_q event π_1 such that π_1 maps to π'_1.

It is easy to define the mapping and total ordering referred to in the properties above. The mapping from torcv' events to tocast' events is the one that matches each torcv' event that occurs when some tuple $\langle m, u, p, t \rangle$ is at the head of $sent$ to the tocast' event that added that tuple to $sent$. The tocast' events are ordered by the order of their respective tuples in *oldsent sent*.

```
   Signature:                                              Transitions:                                          14
 2   Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^≥0                 Input GPSupdate(l, t)_p
     Input tocast(m)_p, m ∈ Msg                               Effect:                                            16
 4   Output tocast'(m, f)_p, m ∈ Msg, f ∈ Bool                  to_send⁻ ← to_send⁺
                                                                to_send⁺ ← λ                                     18
 6 State:                                                       updated ← true
     to_send⁺, to_send⁻ : Msg*, initially λ                                                                       20
 8   updated: Bool, initially false                           Input tocast(m)_p
                                                              Effect:                                            22
10 Trajectories:                                                if updated then
     stop when                                                    to_send⁺ ← append(to_send⁺, m)                 24
12     to_send⁺ ≠ λ ∨ to_send⁻ ≠ λ
                                                              Output tocast'(m, f)_p                             26
                                                              Precondition:
                                                                m = head(to_send⁻ to_send⁺) ∧ (f ⇔ to_send⁻ = λ) 28
                                                              Effect:
                                                                if f then                                        30
                                                                  to_send⁺ ← tail(to_send⁺)
                                                                else to_send⁻ ← tail(to_send⁻)                   32
```

Figure 9-3: *TOBDelay*$_p$, Message delay service.

9.1.2 *TOBDelay*

Figure 9-3 describes the TIOA for *TOBDelay*$_p$, which tags tocast messages from process p with Booleans indicating if the message was submitted since the last GPSupdate, and then passes the pair to *TObcast* to handle. This TIOA is identical, except for the names of the broadcast actions, to *VBDelay*$_p$ (see Figure 7-4) for the virtual layer.

9.1.3 *TOBFilter*

Figure 9-4 gives a TIOA, *TOBFilter*, that acts as an intermediary between *TObcast* and a user of the service, filtering torcv messages based on the amount of time since the first GPSupdate received by the process after the process was started; we only want a process p to receive a message sent from a region u d time ago if we know that process p was alive and knew it was in u or a neighboring region of u from d time ago until it receives the message. This certainty is useful later (in Section 11.3) to simplify our reasoning that all emulators of a virtual node receive exactly the same sequences of messages.

$TOBFilter$'s state consists of the following variables:

- $rtimer : [0, d] \in \mathbb{R}^{\geq 0} \cup \{\bot\}$: This variable is a timer. It is initially $\bot$, but it is set to 0 at the first GPSupdate the process receives, after which it progresses at the rate of real-time until it hits d.

```
  Signature:                                              Transitions:
2   Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ≥0               Input GPSupdate(l, t)_p                          18
    Input torcv′(m, u)_p, m ∈ Msg, u ∈ U                  Effect:
4   Output torcv(m)_p, m ∈ Msg                              if rtimer = ⊥ then                             20
                                                              rtimer ← 0
6 State:                                                    for each ⟨m, u⟩ ∈ to_rcv: region(l) ∉ nbrs+(u)  22
    analog rtimer: [0, d] ∈ ℝ≥0 ∪ {⊥}, initially ⊥            to_rcv ← to_rcv -{⟨m, u⟩}
8   to_rcv: (Msg × U)*, initially λ                                                                        24
                                                          Input torcv′(m, u)_p
10 Trajectories:                                          Effect:                                          26
    if rtimer ∉ {⊥, d} then                                 if rtimer = d then
12    d(rtimer) = 1                                           to_rcv ← append(to_rcv, ⟨m, u⟩)              28
    else constant rtimer
14  stop when                                             Output torcv(m)_p                                30
      to_rcv ≠ λ                                          Precondition:
                                                            ∃u ∈ U: ⟨m, u⟩ = head(to_rcv)                  32
                                                          Effect:
                                                            to_rcv ← tail(to_rcv)                          34
```

Figure 9-4: *TOBFilter[d]*$_p$, Message filtering service.

- $to_rcv : (Msg \times U)^*$: This is the queue of message and region pairs from *TObcast* of messages to be torcved. It is initially empty.

Its interface consists of the following three kinds of actions:

- **Input** GPSupdate$(l, t)_p, l \in R, t \in \mathbb{R}^{\geq 0}, p \in P$: This input indicates that process p is at location l.

- **Input** torcv$'(m, u)_q, m \in Msg, u \in U, q \in P$: This input is the passing on of a message from *TObcast*. The region u indicates the region of the sender at the time it tocast the message.

- **Output** torcv$(m)_q, m \in Msg, q \in P$: This output represents the delivery of a message m at process q. The message m is the message from some pair received through torcv$'$.

When a GPSupdate(l, t) occurs at the process, if the process's $rtimer$ is $\perp$ (meaning this is the first GPSupdate since it started), then $rtimer$ is set to 0 (lines 20-21) so that the process can keep track of how long it has been since it first started receiving updates. For each pair $\langle m, u \rangle$ in its to_rcv queue such that u is not equal to or neighboring $region(l)$, the pair is removed from to_rcv (lines 22-23); this prevents the process from receiving a message that originated from a region that the process has not been in or neigboring for the past d time.

When a $\mathsf{torcv}'(m, u)$ input occurs, if the process's first GPSupdate after it was started was at least d time ago (line 27), then the pair $\langle m, u \rangle$ is appended to the to_rcv queue (line 28). If to_rcv is not empty (line 15) then the head $\langle m, u \rangle$ of the queue is removed and the message m is torcved (lines 30-34).

9.1.4 *TOBspec*

As mentioned earlier, the full specification, *TOBspec*, for the totally ordered broadcast service is equal to the composition of the message ordering service and RW, *TObcast*$\|RW$, composed with the fail transformed filter and delay service for each process, *Fail(TOBFilter*$_p\|$*TOBDelay*$_p$*)* for all $p \in P$, with certain actions hidden. (Remember, the *Fail* transform from Chapter 5 takes an automaton and adds a mechanism for modeling crash failures and restarts of the automaton.) In particular, the hidden actions are the set $H_{TOspec} = \{\mathsf{tocast}'(m, f)_p, \mathsf{torcv}'(m, u)_p \mid m \in Msg, f \in Bool, u \in U, p \in P\}$. This means that *TOBspec* is equal to $\mathsf{ActHide}(H_{TOspec}$, *TObcast*$\|RW\| \prod_{p \in P}$*Fail(TOBFilter*$_p\|$ *TOBDelay*$_p$)).

Reachable states of *TOBspec*

Here we characterize the reachable states of *TOBspec* by providing a list of properties exactly describing those states. We show that (1) the list of properties is an invariant for *TOBspec* and (2) any state satisfying the list of properties is indeed a reachable state of *TOBspec*.

Definition 9.1 *Define $Inv_{TOBspec}$ to be the set of states x such that the following properties hold:*

1. $x \lceil X_{RW} \in Inv_{RW}$.

   This says that the RW component state is a reachable state of RW.

2. $\forall p \in P : TObcast.updates(p) = \{\langle region(l), t\rangle \mid \langle l, t\rangle \in RW.updates(p)\} \wedge TObcast.now = RW.now$.

   This says that real-time and $updates$ should correspond beween RW and $TObcast$.

3. $procs \neq P \Rightarrow \exists\langle m, u, p, t\rangle = head(sent) : t = now - d$.

   This says that if the bookkeeping variable $procs$ is not full, then there must be some exactly d old message at the head of $sent$.

4. $\forall\langle m, u, p, t\rangle \in oldsent : t \leq now - d$, *and tuples are in order of* t.

   This says that tuples in $oldsent$ are at least d old and are ordered by their timestamps.

5. $\forall\langle m, u, p, t\rangle \in sent : t \in [now - d, now]$, *and tuples are in order of* t.

   This says that tuples in $sent$ are at most d old, not sent from a future time, and are ordered by their timestamps.

6. $\forall p \in P, \forall t \in \mathbb{R}^{\geq 0}$, *consider the subsequence* $\langle m_1, u_1, p, t\rangle, \cdots, \langle m_n, u_n, p, t\rangle$ *of* $oldsent\ sent$ *(the concatenation of* $oldsent$ *and* $sent$*). Then* $u_1, \cdots, u_n \in RW.reg^-(p, t)^* RW.reg^+(p, t)^*$.

   This says that the regions attached to messages in $oldsent\ sent$ are consistent with the GPSupdate*s for the senders.*

7. $\forall p \in P : \neg failed_p :$

   (a) $\neg updated_p \Rightarrow rtimer_p = \bot \wedge to_send^-_p = to_send^+_p = to_rcv_p = \lambda$.

   This says that if $updated_p$ does not hold, then the rest of the state of $TOBDelay_p$ and $TOBFilter_p$ is set to initial values.

   (b) $updated_p \Rightarrow \exists\langle l, t\rangle \in RW.updates(p) : t + rtimer_p = now \vee d = rtimer_p < now - t$.

   This says that $updated_p$ implies there was a GPSupdate$_p$ *either $rtimer_p$ ago if $rtimer_p < d$, else at least d time ago.*

   (c) $to_send^-_p \neq \lambda \Rightarrow [rtimer_p > 0 \wedge \exists\langle l, t\rangle \in RW.updates(p) : t = now \wedge \forall\langle m, u, p, now\rangle \in sent : u = RW.reg^-(p, now)]$.

   This says that a non-empty $to_send^-_p$ indicates that p was first updated before now, and updated at now. Also, any messages in $sent$ from p at the current time are from the p's region before its last GPSupdate.

(d) *Let* $proced_p$ *be* $append(oldsent, head(sent))$ *if* $p \notin procs$ *and* $oldsent$ *otherwise. Let* $\langle m_1, u_1, p_1, now - d\rangle, \cdots, \langle m_n, u_n, p_n, now - d\rangle$ *be the subsequence of* $proced_p$ *such that* $\forall i \in [1, n] : regSpan(p, u_i, now - d)$. *Then* $\exists i \in [0, n] : [to_rcv_p = \langle m_{i+1}, u_{i+1}\rangle, \cdots, \langle m_n, u_n\rangle \wedge (rtimer_p < d \Rightarrow i = n)]$. *This says that if* $rtimer_p < d$*, then* to_rcv_p *is empty, else* to_rcv_p *is the (message, region) restriction of a suffix of the sequence of tuples from* d *time ago, tagged with regions* u *that pass* $regSpan(p, u, now - d)$*, and processed by* $TObcast$ *for* p.

We now show that the set of properties describing $Inv_{TOBspec}$ is an invariant for ***TOBspec***. We do this by showing that every reachable state of ***TOBspec*** is in $Inv_{TOBspec}$.

Lemma 9.2 $reachable_{TOBspec} \subseteq Inv_{TOBspec}$.

Proof: Consider a state in $reachable_{TOBspec}$. We must show that it satisfies the properties of a state in $Inv_{TOBspec}$. This is the same as showing that the last state of any closed execution of $TOBspec$ is in $Inv_{TOBspec}$. By Lemma 6.2, property 1 is true throughout such an execution. This leaves properties 2-7 to check. We proceed by induction on closed executions of $TOBspec$.

First, we check that the initial state of $TOBspec$ satisfies the list of properties above. This is easy to see.

Next we check that if the properties hold in some state x and an action is performed that leads to state x', then the properties hold in state x'. We break this down by action:

- GPSupdate$(l, t)_p$: The only relevant properties are 2, 6, and 7. Of these, the only interesting case is for property 7(c).

  For property 7(c), if p is non-failed and $to_send_p^-$ is non-empty in state x', it must be that $to_send_p^+$ was non-empty in state x. By the fact that properties 1, 7(a), and 7(b) held in state x, we know that $rtimer_p > 0$ in state x, and hence in state x'. An update for now is added to $RW.updates(p)$ as a result of this action, so we know that $\exists \langle l, t\rangle \in RW.updates(p) : t = now$. Finally, by properties 1 and 6, we know that in state x, all messages sent by p at the current time in $oldsent\ sent$ must have been tagged with a region equal to $RW.reg^-(p, now)$.

- torcv$'(m,u)_p$: The relevant properties are 3-5 and 7. The only interesting one to check is property 7(d). Consider the case where p is not failed (the only case we have to consider). Since x satisfied property 7(d) in state x, by the precondition for this action to occur, it must have been the case that p was in $procs$ in state x. If $rtimer_p < d$, then the action results in no addition of a tuple to to_rcv_p and we are done. If not, then the action results in an addition of the tuple $\langle m, u \rangle$ to the end of to_rcv_p. Since $x'(proced_p) = append(x(proced_p), head(x(sent_p)))$, the result follows.

- torcv$(m)_p, m \in Msg$: The only relevant property is 7, 7(d) in particular. It is trivial to check.

- tocast$(m)_p$: The only relevant property is 7, but it is trivial to check.

- tocast$'(m,f)_p$: The only relevant properties are 5-7. The only interesting one to check is property 6. Let u be $RW.reg(p)$ if f is true, and $RW.reg^{-}(p, now)$ otherwise. If u is $\perp$, then nothing happens to $sent$, and property 6 still is true. Otherwise, in state x', we know that a tuple $\langle m, u, p, now \rangle$ is added to $sent$ after any other messages sent by p and not before any messages sent before time now. We must show that if the region u is not $RW.reg(p)$, then there is no tuple $\langle m', RW.reg(p), p, now \rangle$ in $x(sent)$. If u is not $RW.reg(p)$, then it must be the case that f is false, meaning that $to_send_p^-$ was non-empty in state x. By property 7(c), this implies that all tuples in $sent$ from p at time now are labelled with a region equal to $RW.reg^{-}(p, now)$, and we are done.

- drop(p): The only relevant properties are 3-5 and 7. They are trivial to check.

Finally we check that for any closed trajectory τ starting with a state x where the properties hold and ending in a state x', the properties hold in state x'. The only continuous variables are now and $rtimer_p$, and it is easy to check that all properties will hold in state x' due to trajectory stopping conditions. ■

Now we show the opposite direction, namely that any state in $Inv_{TOBspec}$ is a reachable state of $TOBspec$. We do this by showing how, given a state x in $Inv_{TOBspec}$, we can

construct an execution of $TOBspec$ that ends in x.

Lemma 9.3 $Inv_{TOBspec} \subseteq reachable_{TOBspec}$.

Proof: Consider a state x in $Inv_{TOBspec}$. We must show that x is a reachable state of $TOBspec$. We do this by constructing an execution α of $TOBspec$ such that $\alpha.lstate = x$.

This construction is done in phases. Each phase is constructed by modifying the execution constructed in the prior phase to produce a new valid execution of ***TOBspec***. After the first four phases, the constructed execution leads to the fail status, region setting, and $rtimer$ for each process that is consistent with that of state x. The fifth phase adds tocast and tocast′ events for $oldsent$ $sent$ message tuples. It then adds torcv′ and drop events for each tuple in $oldsent$. The phase finally adds torcv′ events for messages sent more than d time ago. The sixth phase adds torcv′ and drop events for processes not in $x(proc)$. The seventh phase adds torcv events for messages sent d time ago, but not in a process's to_rcv queue. The final phase adds tocast events for $outgoing$ queue messages in state x; these are messages that were tocast but not yet successfully propagated via a tocast′.

1. *Construction of* α_1: By Theorem 6.12 and the fact that x satisfies property 1 of $Inv_{TOBspec}$, it is possible to construct an execution α_{RW} of RW ending in a state of RW consistent with that of x. α_1 is the execution of ***TOBspec*** such that $\alpha.fstate$'s non-*failed* ***TOBspec*** state is the unique initial one, *failed*$_p$ is false for each $p \in P$, and α_1 restricted to the actions and variables of RW is equal to α_{RW} restricted in a similar manner.

   Validity of execution: It is easy to observe that α_1 is an execution of ***TOBspec***.

   Relation to x: Let y be $\alpha_1.lstate$. Let X_1 be $X_{RW} \cup \{TObcast.updates, TObcast.now\}$. It is obvious that since x satisfies properties 1 and 2 of $Inv_{TOBspec}$, $x \lceil X_1 = y \lceil X_1$. Also, for each $p \in P$ such that $\neg x(failed_p)$, we have $\neg y(failed_p)$.

2. *Construction of* α_2: To construct α_2, for each $p \in P$ if $x(failed_p)$ then we add a fail$_p$ event at time $x(now)$ in α_1, after any other events at time $x(now)$.

Validity of execution: Since fail events are input actions, it is easy to observe that α_2 is an execution of *TOBspec*.

Relation to x: Let y be $\alpha_2.lstate$. Let X_2 be $X_1 \cup \{failed_p \mid p \in P\}$. The relationship from step 1 still is true. In addition, we now have that for all $p \in P$, $x(failed_p) = y(failed_p)$, meaning that $x\lceil X_2 = y\lceil X_2$.

3. *Construction of α_3:* To construct α_3, for each $p \in P$ if $\neg x(failed_p)$ and $\neg x(updated_p)$, then we add a fail_p immediately followed by a $\mathsf{restart}_p$ at time $x(now)$ in α_2, after any other events.

   Validity of execution: Since these are input actions, α_3 is an execution of *TOBspec*.

   Relation to x: Let y be $\alpha_3.lstate$. The relationship from step 2 still is true. Also, for each $p \in P$ that is non-failed in x and has $\neg updated_p$, we have that $y(rtimer_p) = \bot$ since a $\mathsf{restart}_p$ event resets the $rtimer_p$ variable to $\bot$. Together with the fact that x satisfies property 7(a) of $Inv_{TOBspec}$ and that properties of step 2, and hence of step 1, still hold for y, we have that for all $p \in P$ such that $\neg x(updated_p)$, $x(TOBDelay_p) = y(TOBDelay_p)$ and $x(TOBFilter_p) = y(TOBFilter_p)$.

4. *Construction of α_4:* To construct α_4, for each non-failed $p \in P$ with $now \neq rtimer_p < d$, we add a fail_p followed immediately by a $\mathsf{restart}_p$ immediately before the $\mathsf{GPSupdate}_p$ at time $now - rtimer_p$ in α_3.

   Validity of execution: Since these are input actions, α_4 is an execution of *TOBspec*.

   Relation to x: Let y be $\alpha_4.lstate$. The relationship from step 3 still is true. Also, it is easy to see that the construction forces $rtimer_p$ to be equal to $x(now) - rtimer_p$ for those non-failed p for which $rtimer_p$ is less than d and not equal to now. Hence, by the fact that x satsifies property 7(b) of $Inv_{TOBspec}$, in addition to the relationship in step 3, we have that for all non-failed $p \in P$, $x(updated_p) = y(updated_p)$ and $x(rtimer_p) = y(rtimer_p)$.

5. *Construction of α_5:* To construct α_5, there are three substeps.

(a) First, for each $p \in P, t \leq x(now)$, and $u \in U$, consider the subsequence $\langle m_1, u, p, t \rangle, \cdots \in x(oldsent)\ x(sent)$. We construct an alternating sequence s of events $\mathsf{tocast}(m_1)_p$, $\mathsf{tocast}'(m_1, true)_p$, $\mathsf{tocast}(m_2)_p$, $\mathsf{tocast}'(m_2, true)_p, \cdots$. We add events in s in order and immediately after each other at time t in α_4 such that the following hold:

- The addition of a tuple to $sent$ in the tocast' action inserts the tuple so that the ordering is the same as in $x(oldsent)\ x(sent)$.
- If $u = x(RW.reg^-(p,t))$, then events in s are added before any $\mathsf{GPSupdate}_p$ or fail_p event at time t.
- If $u \neq x(RW.reg^-(p,t))$, then events in s are added immediately after any $\mathsf{GPSupdate}_p$ event at time t.

(b) Then, for each $t \leq x(now) - d$, consider the subsequence $\langle m_1, v_1, p_1, t \rangle$, $\langle m_2, v_2, p_2, t \rangle, \cdots \langle m_n, v_n, p_n, t \rangle$ of $x(oldsent)$. We construct a sequence s' of torcv' and drop events to add to α_5, consisting of exactly one $\mathsf{torcv}'(m_1, v_1)_p$ or $\mathsf{drop}(p)$ event for each $p \in P$, followed by exactly one $\mathsf{torcv}'(m_2, v_2)_p$ or $\mathsf{drop}(p)$ event for each $p \in P$, etc. We add this sequence s' of events in order and immediately after each other in α_5 at time $t+d$, after all other events at that time. We select torcv' or drop based on $updates(p)$.

(c) Finally, for each $\mathsf{torcv}'(m_1, v_1)_p$ event that occurs at some time $t' < x(now)$, if p is non-failed with $rtimer_p = d$ in our constructed execution at the time of the torcv' event, then insert a $\mathsf{torcv}(m)_p$ event immediately after the torcv' event in the execution.

Validity of execution: To check that α_5 is an execution, we consider each substep.

(a) Since tocast is an input and hence always enabled, we just need to check that the tocast' events are enabled. What we need to check is that the associated Booleans paired with the messages in the tocast' actions are "correct" and that each tocast' occurs while the process is alive. To see that the Boolean value of true is always appropriate, notice that the construction does not allow there to

be any carryover of messages when a GPSupdate occurs. Hence, all messages that are passed along are from the $to_send_p^+$ queue, meaning the Boolean is always true.

Next we note that for any $t < now$, if a fail_p occurs at time t, only one can occur and it occurs before a $\mathsf{GPSupdate}_p$ (by our construction in steps 1-4). For $t = now$, a fail_p occurs at most once and occurs after a $\mathsf{GPSupdate}_p$.

We consider cases in this step of our construction. The first case places messages sent from the first region of the process at time t before any $\mathsf{GPSupdate}_p$ or fail_p event at time t. Since it is ordered before any fail_p, the process is alive. We conclude that the tocast′ event is enabled.

The second case places messages sent from the ending region of the process at time t after any GPSupdate for the region. The associated region on the message would obviously be for the ending region. If $t < now$ and a fail_p occurs at time t, it is immediately followed by a restart and then the GPSupdate, implying the process would be alive for these actions. If $t = now$, then any fail event is after the GPSupdate, and since s is squeezed in between the GPSupdate and the fail, then the process again must be alive. In either case, we conclude that the tocast' event is enabled.

(b) Exactly one of a $\mathsf{torcv}'(m, v)_p$ or $\mathsf{drop}(p)$ action is enabled for a message sent at some time $t \leq x(now) - d$ if the head of $sent$ is d old and its message has not yet been delivered or dropped to p. Since this is our precondition for adding one of the actions in our construction, and because of the way in which we select which of the actions to perform based on $updates(p)$, we can can conclude that these actions were enabled for each of their corresponding tuples in $x(oldsent)$.

(c) It is obvious that the torcv actions are enabled.

It is easy to check that no trajectory stopping conditions are violated in α_5 since messages in to_send and to_rcv queues are immediately processed, and messages added to $sent$ are removed exactly d time after their addition.

Relation to x: It is easy to see that this construction preserves the properties of

step 4. Let y be $\alpha_5.lstate$. It is clear, by the fact that x satisfies property 6 of $Inv_{TOBspec}$ and our condition in step (a) that tuples be added to $sent$ in a way that reflects the ordering of tuples in $x(oldsent)\ x(sent)$, that $y(oldsent)\ y(sent)$ is equal to $x(oldsent)\ x(sent)$. It is also clear by the fact that x satisfies properties 4 and 5 of $Inv_{TOBspec}$ and by step (b) that $x(oldsent) = y(oldsent)$ and $x(sent) = y(sent)$. By step (c) we can see that in both state x and state y, for all non-failed $p \in P$, to_rcv_p can only contain pairs corresponding to $x(oldsent)$ tuples with timestamps equal to $now - d$.

6. *Construction of* α_6: To construct α_6, let $\langle m, u, q, t\rangle$ be $head(x(sent))$. For each $p \notin x(procs)$, we add a $\mathsf{torcv}'(m, u)_p$ or $\mathsf{drop}(p)$ action based on the $regStart$ test at time $x(now)$ in α_5, after all other events.

   Validity of execution: To check that this is an execution, note that since $x \in Inv_{TOBspec}$, property 3 of $Inv_{TOBspec}$ means that if $x(procs)$ is not equal to P, then it must be the case that $t = now - d$, meaning one of either torcv' or drop is enabled for each $p \in P$.

   Relation to x: It is easy to see that the construction preserves the properties of step 5. Let y be $\alpha_6.lstate$. It is clear that $x(TObcast) = y(TObcast)$. Since x satisfies property 7(d) of $Inv_{TOBspec}$, it should also be clear that for every non-failed $p \in P$, $x(to_rcv_p)$ is a suffix of $y(to_rcv_p)$.

7. *Construction of* α_7: For each non-failed $p \in P$, let i be $|\alpha_6.lstate(to_rcv_p)| - |x(to_rcv_p)|$. Let $\langle m_1, u_1\rangle, \cdots \langle m_n, u_n\rangle$ be $\alpha_6.lstate(to_rcv_p)$, and let $\langle m_{i+1}, u_{i+1}\rangle, \cdots, \langle m_n, u_n\rangle$ be $x(to_rcv_p)$. We construct a sequence s of actions $\mathsf{torcv}(m_1)_p, \cdots, \mathsf{torcv}(m_i)_p$. We then add this sequence s of actions in order and immediately after each other in α_7 at time $x(now)$, after all other events.

   Validity of execution: Note that since $x(to_rcv_p)$ is a suffix of $\alpha_6.lstate(to_rcv_p)$, there must be some prefix of pairs in $\alpha_6.lstate(to_rcv_p)$. Since removal of these pairs via tocast is always enabled at a non-failed process, this is a valid execution.

   Relation to x*:* Let y be $\alpha_7.lstate$. It is easy to see the construction preserves the

properties of step 6. It is also that for every non-failed $p \in P$, $x(TOBFilter_p) = y(TOBFilter_p)$.

8. *Construction of α:* For each non-failed $p \in P$, we modify α_7 by adding tocast events at time $x(now)$:

   - Let s^- be a sequence of events $\text{tocast}(m_1^-)_p$, $\text{tocast}(m_2^-)_p, \cdots$, where $m_1^-, m_2^-, \cdots$ is $x(to_send_p^-)$. The events in s^- are added in order and immediately after each other after any other tocast_p events and before any GPSupdate_p event at time $x(now)$.
   - Let s^+ be a sequence of events $\text{tocast}(m_1^+)_p$, $\text{tocast}(m_2^+)_p, \cdots$, where $m_1^+, m_2^+, \cdots$ is $x(to_send_p^+)$. The events in s^+ are added in order and immediately after each other immediately after any other events at time $x(now)$.

   Validity of execution: Since the added events are inputs, α is an execution.

   Relation to x: The properties of step 7 still hold. Let y be $\alpha.lstate$. It is easy to see that $x(to_send_p^-) = y(to_send_p^-)$ and $x(to_send_p^+) = y(to_send_p^+)$ if each tocast occurs while the process is alive. We check that now.

   By our construction, the only way for a fail_p event to occur at time $x(now)$ for a non-failed process with non-$\perp$ region is in step 4– it would be followed immediately by a restart_p and GPSupdate_p.

   We consider the two cases of s^- and s^+ in this step of our construction. The first case places s^- before any GPSupdate_p event at time $x(now)$. By our observation in the paragraph above and the fact that x satisfies property 7(b) and 7(c), the process would have to be alive.

   The second case places s^+ after any GPSupdate_p for the process. Again, by our observation about step 4, the process would be alive.

We can conclude that $x = \alpha.lstate$. ■

The preceding two lemmas directly imply the following characterization theorem:

Theorem 9.4 $Inv_{TOBspec} = reachable_{TOBspec}$.

```
1  Signature: (Mtup = Msg × P × ℝ≥0 × Bool × ℕ × U)
     Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ≥0
3    Input tocast(m)_p, m ∈ Msg
     Input brcv(mtup)_p, mtup ∈ Mtup
5    Output torcv(m)_p, m ∈ Msg
     Output bcast(mtup)_p, mtup ∈ Mtup
7
   State:
9    analog clock: ℝ≥0 ∪ {⊥}, initially ⊥
     updates: 2^(U×ℝ≥0), initially ∅
11   btime: ℝ≥0, initially 0
     bseq: ℕ, initially 0
13   outgoing+, outgoing−: Msg*, initially λ
     incoming: 2^Mtup, initially ∅
15
   Derived variables:
17   reg−: U ∪ {⊥}
       if ∃⟨u, t⟩ ∈ updates: t < clock then
19       return min({u ∈ U | ∃t′ < clock: ⟨u, t′⟩ ∈ updates
             ∧ ∀⟨u*, t*⟩ ∈ updates: (t* ≤ t ∨ t* = clock)})
21     else return ⊥

23   reg: U ∪ {⊥}
       if ∃u ∈ U: ⟨u, clock⟩ ∈ updates then
25       return min({u ∈ U | ⟨u, clock⟩ ∈ updates})
       else return reg−
27
     regSpan(r: U, t: ℝ≥0): Bool
29     return ∃⟨u, t′⟩ ∈ updates: [t′ ≤ t ∧
           ∀⟨v, t′′⟩ ∈ updates: (t′′ ≥ t′ ⇒ r ∈ nbrs+(v)) ]
31
   Trajectories:
33   d(clock) = 1
     stop when
35     Any precondition is satisfied.

37 Transitions:
     Input tocast(m)_p
39   Effect:
       outgoing+ ← append(outgoing+, m)

   Input GPSupdate(l, t)_p                                               42
   Effect:
     if (clock ≠ t ∨ updates = ∅ ∨ ∃⟨u, t′⟩ ∈ updates: [t′ ≥ t] ∨ btime > t   44
         ∨ ∃⟨m, s, t′, f, b, r⟩ ∈ incoming: t′ ∉ [t -d, t)) then
       clock, btime ← t                                                   46
       bseq ← 0
       updates, incoming ← ∅                                              48
       outgoing+ ← λ
     updates ← updates ∪ {⟨region(l), t⟩}                                 50
     outgoing− ← outgoing+
     outgoing+ ← λ                                                        52
     for each ⟨m, s, t′, f, b, r⟩ ∈ incoming: ¬ regSpan(r, t′)
       incoming ← incoming -{⟨m, s, t′, f, b, r⟩}                         54

   Input brcv(⟨m, s, t, f, b, r⟩)_p                                       56
   Effect:
     if (t ∈ [clock -d_phys, clock) ∧ regSpan(r, t)) then                 58
       incoming ← incoming ∪ {⟨m, s, t, f, b, r⟩}
                                                                          60
   Output bcast(⟨m, p, t, f, b, r⟩)_p
   Precondition:                                                          62
     m = head(outgoing− outgoing+) ∧ [f ⇔ ∃u ∈ U: {⟨u, t⟩} = updates]
     r ≠ ⊥ ∧ (outgoing− = λ ⇒ r = reg) ∧ (outgoing− ≠ λ ⇒ r = reg−)      64
     t = clock ≠ ⊥ ∧ [(btime ≠ t ∧ b = 1) ∨ (btime = t ∧ b = bseq+1)]
   Effect:                                                                66
     if outgoing− ≠ λ then
       outgoing− ← tail(outgoing−)                                        68
     else outgoing+ ← tail(outgoing+)
     btime ← clock                                                        70
     bseq ← b
                                                                          72
   Output torcv(m)_p
   Local:                                                                 74
     s: P, t: ℝ≥0, f: Bool, b: ℕ, r: U
   Precondition:                                                          76
     updates ≠ ∅ ∧ ⟨m, s, t, f, b, r⟩ ∈ incoming ∧ t ≤ clock -d
     ∀⟨m′, s′, t′, f′, b′, r′⟩ ∈ incoming: ⟨t, s, f, b⟩ ≤ ⟨t′, s′, f′, b′⟩   78
   Effect:
     incoming ← incoming -{⟨m, s, t, f, b, r⟩}                            80
```

Figure 9-5: *TOBimpler*$_p$, providing ordered broadcast.

9.2 *TOBimpl*: Implementation

Here we present a self-stabilizing implemention of *TOBspec* using the physical layer. For each physical node id $p \in P$, the corresponding physical node has a TIOA called *TOBimpler*$_p$, which we describe in this section. The implementation of the entire totally ordered broadcast service, *TOBimpl*, is then the composition of *Fail(TOBimpler*$_p$) for all the $p \in P$ and *Pbcast*$\|RW$, with the bcast and brcv actions of *Pbcast* hidden. Recall that the *Fail*-transform of an automaton takes an automaton and adds a mechanism for allowing crash failures and restarts.

Our technique is loosely based on one originally suggested by Lamport [61]. In that work, Lamport presented an ordering technique to ensure total ordering of messages. We extend that technique here to accommodate both multiple transmissions of the same message by the same process at the same time (allowing us to use this service to help emulate the virtual layer broadcast service where such multiple transmissions are allowed) and process failures. Each tocast message is tagged by the sender with the time of transmission, the id and region of the sender, and a Boolean and sequence number, and then sent using *Pbcast*. Received messages from nearby regions are stored until exactly d time has passed since the message was sent. They are then torcved in lexicographic order of sender id, Boolean flag, and sequence number, in that order. In the lexicographic order, a false value is ordered before a true value, according to the convention that false is equal to 0 and true is equal to 1.

The sequence number allows us to order messages sent by a process at the same time. The Boolean value is an indication of whether or not the sender has received its first GPSupdate since starting at the time of the broadcast. This is important to ensure that, when we allow failures and restarts of the physical nodes, if a process broadcasts a message, fails, restarts, and broadcasts a new message, all at some time t, the message sent after the fail and restart is ordered after the one sent before the fail and restart: Any message sent before the failure would be tagged with a false Boolean flag. After a restart, a process's tocast is only sent out if a GPSupdate occurs before the tocast. Hence, any message sent after a process restarts would have a true Boolean flag, ordering it after the pre-failure messages. Now we describe *TOBimpler* in more detail.

The state variables of *TOBimpler*$_p$ are as follows:

- $clock : \mathbb{R}^{\geq 0} \cup \{\perp\}$: This is the local clock time. It is initially $\perp$, but after the first GPSupdate$_p$ after initialization, it should be equal to the current real system time.

- $updates : 2^{U \times \mathbb{R}^{\geq 0}}$: This is the set of region and time pairs that correspond with the GPSupdates received at the process. It is initially $\emptyset$.

- $btime : \mathbb{R}^{\geq 0}$: This is a time at least as large as the broadcast timestamp of the last message sent by the process but no larger than the current time. It is initially 0.

- $bseq : \mathbb{N}$: This is a message sequence number, initially 0. It is used to help order messages sent at the same time by the process.

- $outgoing^+ : Msg^*$: This is a queue of tocast messages yet to be broadcast via *Pbcast*, initially empty.

- $outgoing^- : Msg^*$: This is also a queue of tocast messages yet to be broadcast via *Pbcast*, initially empty. It contains messages that were submitted before the latest GPSupdate at the process.

- $incoming \in 2^{Msg \times P \times \mathbb{R}^{\geq 0} \times Bool \times N \times U}$: This is an initially empty set of messages, each tagged by sender, broadcast time, a Boolean, a sequence number, and a broadcast region. It is the set of messages received by the process through *Pbcast*, but not yet processed in a torcv event.

We also define two derived variables, both calculated in manner similar to that of their counterparts in Section 9.1.1:

- $reg : U \cup \{\bot\}$ maps to the region indicated by the last $\mathsf{GPSupdate}_p$. If no such region exists, the function returns $\bot$.

- $reg^- : U \cup \{\bot\}$ maps to the region indicated by the last $\mathsf{GPSupdate}_p$ before the current time. If no such region exists, the function returns $\bot$.

- $regSpan : (U \times \mathbb{R}^{\geq 0}) \rightarrow Bool$: This function takes a region r and a time t, and returns a Boolean indicating whether or not the process has entries in $updates$ consistent with the process having been in or neighboring region r from some time before or equal to time t and through the present time.

When a node receives a GPSupdate (line 42) when its $updates$ is $\emptyset$, indicating that the GPSupdate is the first since it started, or when there is some local inconsistency in state (lines 44-45), then it initializes its non-$clock$ and non-$btime$ variables (lines 47-49), and sets $clock$ and $btime$ to the time indicated by GPSupdate (line 46). Otherwise, and after the above initialization, the current region and time is added to $updates$ (line 50), $outgoing^-$ is replaced with $outgoing^+$ (line 51), $outgoing^+$ is cleared (line 52), and each

entry in $incoming$ that is tagged with a region and time that does not pass the $regSpan$ test is removed from $incoming$ (lines 53-54).

When a node receives a $\mathsf{tocast}(m)_p$ input (line 38), it appends m to its local $outgoing^+$ sequence (line 40). Whenever $outgoing^- outgoing^+$ is nonempty for a process with a non-$\bot$ $clock$ and non-$\bot$ $r = reg$ if $outgoing^-$ is empty or $r = reg^-$ otherwise, a bcast_p action occurs (lines 35 and 63-65). In this action, the m at the head of $outgoing^- outgoing^+$ is expanded into a larger message tuple $\langle m, p, clock, f, b, r\rangle$, which includes the process id, current time, values f and b to help order its messages sent at a particular time, and the region r of the message. The tuple is broadcast using *Pbcast*. f is true exactly when the process's $updates = \{\langle r, clock\rangle\}$ (line 63), indicating whether the process had received its first GPSupdate since initialization at this time. b is a message sequence number, either equal to $bseq{+}1$ if $btime = clock$ (incrementing the sequence number if $btime$ was already updated to the current time, either through a message having been sent at the current time or a GPSupdate having updated the process's state), or 1 (resetting the sequence number) if this is the first message sent at this time since $btime$ was last updated (line 65). As a result of the action, if $outgoing^-$ is nonempty, the head of $outgoing^-$ is removed, else the head of $outgoing^+$ is removed (lines 67-69). Then the $btime$ and $bseq$ numbers are updated to match the timestamp and number b of the message tuple that was sent (lines 70-71).

When a node receives such a message tuple (line 56) from its own or a neighboring region r such that the message was sent at a time t that is not too soon or too late by the broadcast service requirements and such that $regSpan(r, t)$ is true (line 58), it adds the the message tuple to $incoming$ (line 59). Message tuples in $incoming$ with timestamps that are exactly d old are removed from $incoming$ and torcved in order of sender id and sequence number (lines 35 and 73-80).

As mentioned in the beginning of the section, the complete implementation of the totally ordered broadcast service is the composition of *Pbcast*$\|RW$ and $Fail(TOBimpler_p)$ for all $p \in P$. Hence, in addition to the variables and actions described above, for each $p \in P$, there is a $failed_p$ Boolean flag indicating whether or not the process is failed, as well as a fail_p and $\mathsf{restart}_p$ input action for each $p \in P$. Since brcv and bcast actions do not exist in the *TOBspec*, we also hide those actions in the implementation.

9.3 Correctness of the implementation

In this section we describe aspects of the correctness of our implementation of the totally ordered broadcast service. Define ***TOBimpler*** to be the composition of ***Fail(TOBimpler***$_p$) for all $p \in P$, and let H_{TOimpl} be $\{\mathsf{bcast}(m)_p, \mathsf{brcv}(m)_p \mid m \in (Msg \times P \times \mathbb{R}^{\geq 0} \times Bool \times \mathbb{N} \times U), p \in P\}$. The implementation of the service is then ***TOBimpl*** = $\mathsf{ActHide}(H_{TOimpl}, Pbcast\|RW\|TOBimpler)$, the composition of ***Pbcast***$\|RW$ and $TOBimpler$ with the bcast and brcv actions for implementation messages hidden.

To show correctness, we first describe a legal set $L_{TOBimpl}$ of $TOBimpl$ (Section 9.3.1). Then, we show that $Start(TOBimpl, L_{TOBimpl})$ implements $Start(TOBspec, Inv_{TOBspec)}$. We do this in the following way: using the legal set definition (Definition 3.12) and a simulation relation (Definition 2.20), we show in Section 9.3.2 that our implementation, ***TOBimpl***, implements ***TOBspec***, meaning that traces of the implementation are contained in traces of the specification. The simulation relation is defined only for states of $TOBimpl$ in the legal set $L_{TOBimpl}$; we then show a separate result that each of these states is related to some reachable state of $TOBspec$ (Lemma 9.18).

Next, we argue in Section 9.3.3 that ***TOBimpler*** is self-stabilizing to $L_{TOBimpl}$ relative to $R(RW\|Pbcast)$, which allows us to finally conclude in Theorem 9.25 that our implementation eventually reaches a state that is related to a reachable state of $TOBspec$.

We use this approach in future chapters to describe correctness and stabilization of an implementation of a system. To summarize the strategy:

1. Define a legal set L_I for the implementation I, and show that the set is a legal set.

2. Define a legal set L_S for the specification S, and show that the set is a legal set.

3. Show that $Start(I, L_I) \leq Start(S, L_S)$, meaning that traces of the implementation started in legal set L_I are traces of the specification started in legal set L_S. This can be shown in the following way:

    (a) Define a simulation relation between states of the implementation in L_I and states of the specification. Show the relation is a simulation relation.

(b) Show that for each state in L_I, there exists a state in L_S of states of the specification such that the simulation relation holds between the states. (In the case of totally ordered broadcast, we define the invariant set of the specification as the reachable states, which happens to be a set of invariant states. In general, this is not necessary. It is done simply for convenience here, since it is obvious that the trace of the service starting from a reachable state is a suffix of some trace of the specification that satisfies the properties described in Section 9.1.1.)

4. Show that the implementation self-stabilizes to L_I.

5. Conclude that the set of traces of the implementation stabilizes to the set of traces of execution fragments of the specification starting in L_S. (This follows immediately from points 3 and 4.)

For the rest of the section, we refer to a state variable v of *Fail(TOBimpl*$_p$) as v_p. We also refer to a state variable v of *RW*$\|$*Pbcast* simply as v.

9.3.1 Legal sets

Here we describe a legal set of $TOBimpl$ by describing four legal sets, each a subset of the prior one. Recall from Lemma 3.13 that a legal set of states for a TIOA is one where each closed execution fragment starting in a state in the set ends in a state in the set. We break the definition of the legal set up into four legal sets in order to simplify the proof reasoning and more easily prove stabilization later, in Section 9.3.3.

Legal state set $L^1_{TOBimpl}$:

The first set of legal states describes some properties that become true at an alive process at the time of the first GPSupdate for the process.

Definition 9.5 *$L^1_{TOBimpl}$ is the set of states x of $TOBimpl$ where all of the following hold:*

1. $x \lceil X_{RW\|Pbcast} \in Inv_{RW\|Pbcast}$.
This says that the state restricted to the variables of $RW\|Pbcast$ are reachable states of $RW\|Pbcast$ (Theorem 6.12 showed that $Inv_{RW\|Pbcast} = reachable_{RW\|Pbcast}$).

2. *For each* $p \in P : (\neg failed_p \wedge updates_p \neq \emptyset)$ *:*

   (a) $clock_p = now \wedge btime_p \leq clock_p$.

   This says that a non-failed process with a non-$\emptyset$ $updates$ *must have a local* clock *that matches* $RW \| Pbcast$*'s* now*, and a* $btime$ *variable that is not set in the future.*

   (b) $\forall \langle m, s, t, f, b, r \rangle \in incoming_p : [regSpan_p(r, t) \wedge t \in [clock_p - d, clock_p)]$.

   This says that the message tuples in the $incoming$ *set of a non-failed process with a non-*$\emptyset$ $updates$ *are labelled with timestamps that are not set in the future or before* d *before the current time. It also says that each such tuple was sent from a region at a time such that process* p *has been in range for the transmission period.*

   (c) $\exists \langle l, t \rangle \in updates(p) : [\langle region(l), t \rangle \in updates_p$

   $\wedge \forall \langle a, t' \rangle \in updates(p) \cup updates_p - \{\langle l, t \rangle, \langle region(l), t \rangle\} : t' < t]$.

   This says that the latest update for p *matches between* RW *and* $TOBimpler$*, and that the latest update is unique.*

   (d) $outgoing^- \neq \lambda \Rightarrow$

   $[\exists \langle v, t \rangle \in updates_p : t < clock_p \wedge \exists u \in U : \langle u, clock_p \rangle \in updates_p]$.

   This says that the $outgoing^-$ *queue of a non-failed process with a nonempty* $updates$ *is nonempty only if there is some recorded update that occurred before the current time and an update that occurred at the current time.*

Lemma 9.6 $L^1_{TOBimpl}$ *is a legal set for* $TOBimpl$*.*

Proof: Let x be any state in $L^1_{TOBimpl}$. By Definition 3.12 of a legal set, we must verify two things for state x:

- For each state x' of $TOBimpl$ and action a of $TOBimpl$ such that (x, a, x') is in the set of discrete transitions of $TOBimpl$, state x' is in $L^1_{TOBimpl}$.

- For each state x' and closed trajectory τ of $TOBimpl$ such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in $L^1_{TOBimpl}$.

By Theorem 6.4, we know that if x satisfies the first property of $L^1_{TOBimpl}$, then any discrete transition of $TOBimpl$ will lead to a state x' that still satisfies the first property, and any closed trajectory starting with state x will end in some state that satisfies the first property. This implies that we just need to check that in the two cases of the legal set definition, the state x' satisfies all parts of the second property of $L^1_{TOBimpl}$.

For the first case of the legal set definition, we consider each action:

- drop$(\langle m, s, t, f, b, u\rangle, t, q, p)$, tocast$(m)_p$: These don't impact property 2.

- fail$_p$: This action trivially preserves property 2.

- restart$_p$: Since this action sets $updates_p$ to $\emptyset$ if it makes any state changes at all, property 2 would still trivially hold.

- torcv$(m)_p$: This could impact property 2(b). However, since the only impact of the action is the removal of a tuple in $incoming_p$, then if property 2(b) holds in state x, it continues to hold in state x'.

- GPSupdate$(l, t)_p$: Let v be $region(l)$. If the conditional on lines 44-45 holds, then this action first sets $clock_p$ and $btime_p$ to now, $bseq_p$ to 0, $updates_p$ and $incoming_p$ to $\emptyset$, and $outgoing^+_p$ to λ. Then, regardless of whether the conditional holds, the action adds $\langle l, t\rangle$ to $updates(p)$ and $\langle v, t\rangle$ to $updates_p$, overwrites $outgoing^-_p$ with $outgoing^+_p$, clears $outgoing^+_p$, and removes any element of $incoming_p$ whose region and time does not pass $regSpan_p$. It is easy to see that the resulting state x' satisfies properties 2(a)-2(c).

  For property 2(d), it is obvious that the only thing to verify is that if $outgoing^-_p$ is not empty, then there exists some pre-$clock_p$ timestamped pair in $updates_p$. We consider the cases for whether the if conditional on lines 44-45 holds. If it held, then $outgoing^-_p$ is empty in state x', meaning property 2(d) holds. If the conditional did not hold, then by the fact that $updates_p$ must have contained a pre-$clock_p$ timestamped pair in x, property 2(d) still holds.

- brcv$(\langle m, s, t, f, b, r\rangle)_p$: The only property this might impact is 2(b). However, the

conditional on line 58 ensures that if a new tuple is added to $incoming_p$, then it satisfies the property.

- bcast$(\langle m, q, t, f, b, r\rangle)_p$: Properties 2(b) - 2(d) are obviously not impacted. Since this action sets $btime_p$ to $clock_p$, property 2(a) still holds.

For the second case of the legal set definition, we now consider any closed trajectory τ such that $x = \tau.fstate$. Let x' be $\tau.lstate$. We must show that $x' \in L^1_{TOBimpl}$. It is easy to see that because the only evolving variables referenced in the properties are $clock_p$ and now which evolve at the same rate, property 2(a) holds. With the trajectory stopping conditions of *TOBimpler* on line 35, if an entry in some $incoming_p$ has a timestamp from more than d time ago, then it is torcved. This means that property 2(b) remains true throughout a trajectory. Property 2(c) is not impacted in a trajectory. Property 2(d) holds throughout a trajectory because of the stopping conditions on line 35, enforcing that no time passes until any entries in $outgoing^-_p$ and $outgoing^+_p$ are cleared. ■

Legal state set $L^2_{TOBimpl}$:

The next legal set describes a subset of states of $L^1_{TOBimpl}$ that satisfy some additional properties with respect to the $pbcastq$, $outgoing$, $updates$, and $btime$ variables.

Definition 9.7 *$L^2_{TOBimpl}$ is the set of states x of $TOBimpl$ where all of the following hold:*

1. $x \in L^1_{TOBimpl}$.

2. $\forall p \in P, \forall \langle\langle m, s, t, f, b, r\rangle, t', P'\rangle$ *in* $pbcastq(p) : t' = now$:

   (a) $s = p \wedge t = t' \wedge (f \Rightarrow \exists l \in R : \langle l, t\rangle \in updates(p) \wedge region(l) = r)$
   $\wedge r \in \{reg^-(p, t), reg^+(p, t)\}$.
   This says that any message tuple in $pbcastq(p)$ for some $p \in P$ has a source tag equal to the process id, a timestamp equal to the time that the message was actually sent, and a region tag consistent with the updates at that time. It also says that if a message tuple has a true Boolean tag then its region is the ending region of the process at transmission time.

(b) $\forall\langle\langle m', s', t'', f', b', r'\rangle, t', P''\rangle \in pbcastq(p) - \{\langle\langle m, s, t, f, b, r\rangle, t', P'\rangle\}$:
$(f \neq f' \vee b \neq b') \wedge [(\langle f, b\rangle < \langle f', b'\rangle \wedge r \neq r') \Rightarrow r = reg^{-}(p, t)]$.
This says that any two message tuple records for messages that were sent by the same process at the same time and with the same Boolean tag and sequence number are actually the same tuple. It also says that if two message tuples with the same correct timestamp have different region tags, then the one whose Boolean tag paired with message sequence number is lower than the other's has a region tag equal to the sender's region at the beginning of time t. Remember that the Boolean value is an indication of whether or not the sender has received its first GPSupdate *since starting at the time the message was originally submitted; any message with a false Boolean is one that was originally submitted before any* GPSupdate *for the period occurred at the process, while any message with a true Boolean is one that was submitted after. Hence, the region associated with a false Boolean is the region for the process at the beginning of time t, while a region associated with a true Boolean is the region for the process after a* GPSupdate *occurred at the process at time t.*

(c) $(\neg failed_p \wedge r = reg_p \neq reg_p^{-}) \Rightarrow outgoing_p^{-} = \lambda$.
This says that if some message was sent by a non-failed process with a non-$\perp$ region at the current time and with a region tag equal to the current local region of the process which differs from the prior region, then $outgoing^{-}$ is empty.

(d) Let f_p be a Boolean such that $f_p \Leftrightarrow \exists u \in U : \{\langle u, clock_p\rangle\} = updates_p$. Let $seqnum_p$ be a natural such that $seqnum_p = 0$ if $btime_p \neq clock_p$ and $seqnum_p = bseq_p$ otherwise.
Then $(\neg failed_p \wedge updates_p \neq \emptyset) \Rightarrow \langle clock_p, f_p, seqnum_p\rangle \geq \langle t, f, b\rangle$.
This says that any message ordering tags that might be added to an outgoing message will be larger than any previously broadcast tags at this time.

Lemma 9.8 *$L^2_{TOBimpl}$ is a legal set for $TOBimpl$.*

Proof: Let x be any state in $L^2_{TOBimpl}$. By Definition 3.12 of a legal set, we must verify two things for state x:

- For each state x' of $TOBimpl$ and action a of $TOBimpl$ such that (x, a, x') is in the set of discrete transitions of $TOBimpl$, state x' is in $L^2_{TOBimpl}$.

- For each state x' and closed trajectory τ of $TOBimpl$ such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in $L^2_{TOBimpl}$.

By Lemma 9.6, we know that if x satisfies the first property of $L^2_{TOBimpl}$, then any discrete transition of $TOBimpl$ will lead to a state x' that still satisfies the first property, and any closed trajectory starting with state x will end in some state that satisfies the first property. This implies that we just need to check that in the two cases of the legal set definition, the state x' satisfies all parts of the second property of $L^2_{TOBimpl}$.

For the first case of the legal set definition, we consider each action:

- drop$(\langle m, s, t, f, b, u\rangle, t, q, p)$, tocast$(m)_p$, torcv$(m)_p$, brcv$(\langle m, s, t, f, b, r\rangle)_p$: These don't impact property 2.

- fail$_p$: This action doesn't affect properties 2(a) and 2(b). It trivially preserves properties 2(c) and 2(d).

- restart$_p$: This action doesn't affect properties 2(a) and 2(b). Since this action sets $updates_p$ to $\emptyset$ if it makes any state changes at all, properties 2(c) and 2(d) still trivially hold.

- GPSupdate$(l, t)_p$: Let v be $region(l)$. It is trivial to see that properties 2(a) and 2(b) are still satisfied in state x'.

  The only way for this action to change any state relevant to the other parts of property 2 is if $\neg failed_p$. If the conditional on lines 44-45 holds, then the resulting state trivially satisfies property 2(c). For property 2(d), we know that $\langle clock_p, f_p, seqnum_p\rangle$ is equal to $\langle clock_p, true, 0\rangle$ in state x'. This is at least as great as the corresponding tags of $pbcastq(p)$ messages sent at time $clock_p$ if we can show that any such $pbcastq(p)$ message tags have $false$ in their second field. This follows from the fact that state x' satisfies property 3 of Inv_{RW} (see Definition 6.1), meaning that no more than one update occurred at the current time, and because property 2(a) held in state x, implying that no messages with $true$ flags were sent at the current time by p.

If the conditional on lines 44-45 does not hold, then the resulting state is one for which property 2(d) obviously still holds. More interesting to show is property 2(c). By property 3 of Inv_{RW}, we know that no other update could have occurred at this time. Hence, since property 2(a) held in state x, all messages in $pbcastq(p)$ must be tagged with $reg^{-}(p, now)$, which is either equal to v, meaning we are done, or equal to some other region, also meaning we are done.

- bcast$(\langle m, q, t, f, b, r \rangle)_p$: Examination of the attached tags in lines 63-65 show us that property 2(a) still holds. If $outgoing_p^-$ was empty in state x, then this action sets $outgoing_p^+$ to its tail and broadcasts a message with the current region. Property 2(c) still holds. If $outgoing_p^-$ was not empty in state x, then this action sets $outgoing_p^-$ to its tail and broadcasts a message with a region corresponding to the node's prior update. Since property 2(c) held in state x, it must still hold in state x'. Since in x, $\langle clock_p, f_p, seqnum_p \rangle$ is at least as large as any corresponding tags in $pbcastq(p)$ for this time, then this new message's tuple is strictly larger by the precondition for the action, and $btime_p$ and $bseq_p$ are updated by the action to match this message's t and b, preserving property 2(d). Since the tags are strictly larger, examination of the tags attached to the message imply that property 2(b) still holds.

For the second case of the legal state definition, we consider any closed trajectory τ such that $x = \tau.fstate$. Let x' be $\tau.lstate$. We must show that $x' \in L^2_{TOBimpl}$. It is easy to see that because the only evolving variables referenced in property 2 are $clock_p$ and now, with the trajectory stopping conditions of ***TOBimpler*** in line 35, messages in $outgoing$ queues will be removed through a bcast, preserving properties 2(a), 2(b), and 2(c). Property 2(d) is easily seen to remain true throughout a trajectory since the only relevant variable is $clock_p$, and any messages in transit that previously satisfied 2(d) have tags that continue to satisfy 2(d) when time passes. ■

Legal state set $L^3_{TOBimpl}$:

The next legal set is a subset of states of $L^2_{TOBimpl}$ that satisfy some additional properties with respect to the set of messages in transit and the history stored in $updates$.

Definition 9.9 *$L^3_{TOBimpl}$ is the set of states x of $TOBimpl$ where all of the following hold:*

1. $x \in L^2_{TOBimpl}$.

2. $\forall t' \geq now - d, \forall p \in P, \forall \langle\langle m, s, t, f, b, r\rangle, t', P'\rangle$ *in* $pbcastq(p)$*:*

   (a) $s = p \wedge t = t' \wedge r \in \{reg^-(p, t), reg^+(p, t)\}$
   $\wedge (f \Rightarrow \exists l \in R : \langle l, t\rangle \in updates(p) \wedge region(l) = r)$.
   This is property 2(a) of $L^2_{TOBimpl}$, extended to all $t' \geq now - d$.

   (b) $\forall\langle\langle m', s', t'', f', b', r'\rangle, t', P''\rangle \in pbcastq(p) - \{\langle\langle m, s, t, f, b, r\rangle, t', P'\rangle\}$:
   $(f \neq f' \vee b \neq b') \wedge [(\langle f, b\rangle < \langle f', b'\rangle \wedge r \neq r') \Rightarrow r = reg^-(p, t)]$.
   This is property 2(b) of $L^2_{TOBimpl}$, extended to all $t' \geq now - d$.

   (c) $(r \in \{reg^-(p, t), reg^+(p, t)\} \wedge t = t') \Rightarrow \forall q \in P - P'$:
   $[(\langle m, s, t, f, b, r\rangle \notin incoming_q \wedge \neg failed_q \wedge \exists\langle l', t'\rangle \in updates(p) : [t' \leq t \wedge$
   $\forall\langle l, t''\rangle \in updates(p) : t'' \geq t' \Rightarrow region(l) \in nbrs^+(r)] \wedge regSpan_q(r, t)) \Rightarrow$
   $(t \leq now - d \wedge \forall\langle m', s', t, f', b', r'\rangle \in incoming_q : \langle s', f', b'\rangle \geq \langle s, f, b\rangle)]$.
   In other words, consider any message tuple in a process's $pbcastq$ such that the tuple's region tag r is a region of the process at broadcast time, and the attached timestamp t is the time when the message was broadcast. Now consider any non-failed process q where q has been in range of the broadcast and has local $updates$ that indicate this (meaning q should receive the message). This property says that $RW\|Pbcast$ has yet to deliver the message to q or, if it has delivered the message, the message tuple is either in $incoming_q$ (meaning q received the message from $Pbcast$ and has the tuple stored locally to process) or the timestamp is at least d old and all tuples in $incoming_q$ have larger timestamp/ source/ Boolean flag/ sequence number tags than the message tuple (meaning that q received the message from $Pbcast$ and processed the tuple locally and in order with respect to the other message tuples it was supposed to receive).

3. *For each* $p \in P : (\neg failed_p \wedge updates_p \neq \emptyset)$:
   $\exists\langle u, t\rangle \in updates_p : [(t \leq now - d \vee t = min(\{t' \mid \exists v \in U : \langle v, t'\rangle \in updates_p\})) \wedge$

$\forall t' \geq t : \{u \mid \langle u, t' \rangle \in updates_p\} = \{region(l) \mid \langle l, t' \rangle \in updates(p)\}]$.

This says that for any non-failed process p*, there is some time* t *such that* $updates_p$ *corresponds with* $updates(p)$ *for all entries with timestamps starting at* t*, and such that* t *is either the minimum timestamp in* $updates_p$ *or is at least* d *old.*

Lemma 9.10 $L^3_{TOBimpl}$ *is a legal set for* $TOBimpl$*.*

Proof: Let x be any state in $L^3_{TOBimpl}$. By Definition 3.12 of a legal set, we must verify two things for state x:

- For each state x' of $TOBimpl$ and action a of $TOBimpl$ such that (x, a, x') is in the set of discrete transitions of $TOBimpl$, state x' is in $L^3_{TOBimpl}$.
- For each state x' and closed trajectory τ of $TOBimpl$ such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in $L^3_{TOBimpl}$.

By Lemma 9.8, we know that if x satisfies the first property of $L^3_{TOBimpl}$, then any discrete transition of $TOBimpl$ will lead to a state x' that still satisfies the first property, and any closed trajectory starting with state x will end in some state that satisfies the first property. This implies that we just need to check that in the two cases of the legal set definition, the state x' satisfies all parts of the second and third property of $L^3_{TOBimpl}$. By simple extension of the reasoning in Lemma 9.8, we can also quickly see that properties 2(a) and 2(b) hold. It is also simple to see that property 3 can only be affected by the GPSupdate action. Hence, for each non-GPSupdate action we consider only property 2(c), and for GPSupdate we consider property 2(c) and 3.

For the first case of the legal set definition, we consider each action:

- drop$(\langle m, s, t, f, b, u \rangle, t, q, p)$: This action is only enabled in state x if there is some set of ids P' such that P' contains q, $\langle \langle m, s, t, f, b, u \rangle, t, P' \rangle \in pbcastq(p)$, $t \neq now$, and the distance between the last reported location of p at time t and the last reported location of q is greater than r_{real}. The action results in the removal of q from P'. However, by the precondition, we know that q is only removed from P' if the distance above is more than r_{real}. By Lemma 6.13, $reg(q)$ must not be in $nbrs^+(u)$, so the property remains true.

- fail$_p$, restart$_p$, bcast$(\langle m, q, t, f, b, r\rangle)_p$: These actions trivially preserve properties 2 and 3.

- tocast$(m)_p$: This doesn't impact properties 2 and 3.

- torcv$(m)_p$: For property 2(c), note that the precondition for the action guarantees that in state x there must be some $\langle m, s, t, f, b, r\rangle \in incoming_p$ such that $t \leq clock_p - d$ and $\langle t, s, f, b\rangle$ is ordered before all other similar tuple components in $incoming_p$. Since property 2(b) of $L^1_{TOBimpl}$ holds in state x, we know that $t \geq clock_p - d$. This implies that $t = clock_p - d$. Hence, the two conditions on the right of the last implication in property 2(c) both hold.

- GPSupdate$(l, t)_p$: Let v be $region(l)$. For this action, we must consider both property 2(c) and 3.

  For 2(c), consider what happens if p is not failed. If the conditional on lines 44-45 holds, then state x' will have $updates_p = \{\langle v, t\rangle\}$. This means that $regSpan_p$ will only be true for messages with $t = now$. By property 3 in the description of Inv_{Pbcast}, the attached P' in the $pbcastq$ record contains q, satisfying property 2(c).

  If the conditional on lines 44-45 does not hold, then we just need to be sure that no message tuples that previously should not be in $incoming_q$ suddenly should be. However it is obvious that the addition of a pair to $updates_p$ does not suddenly allow prior disallowed tuples. Property 2(c) is still satisfied.

  For property 3, we are only interested in the case where p is not failed in state x. If the conditional on lines 44-45 holds, then it is obvious that property 3 holds in state x', since $updates_p = \{\langle v, t\rangle\}$ in state x'. If the conditional does not hold, then we know that $x(updates_p) \neq \emptyset$ and $updates_p$ in x' equals $updates_p$ in x, with an additional $\langle v, t\rangle$ element. Since state x satisfied property 3 and $updates_p$ was not empty, there was some pair in $updates_p$ such that the property held relative to the pair. If we select the same pair, it is obvious that the property still holds in state x'.

- brcv$(\langle m, s, t, f, b, r\rangle)_p$: Property 2(c) could only be a problem if this action does not add this tuple to $incoming_p$ or if it adds the tuple but $t = clock_p - d$ and $\langle s, f, b\rangle$ is

smaller than that of other entries with the same timestamp. The second can't happen by property 2 of Inv_{Pbcast} and the if condition on line 58. We examine the first. By the if condition on line 58 in the action, if the tuple is not added it must mean that either $\neg regSpan_p(r,t)$ or $t \geq clock_p$ or $t < clock_p - d_{phys}$. By properties 1-3 of Inv_{Pbcast} and since t is equal to the actual time the tuple is broadcast, then $t < clock_p$ and $t \geq clock_p - d_{phys}$. Hence, for the tuple not to be added, $\neg regStart_p(r,t)$. In either case, one of the conditions on the left of the last implication in property 2(c) fails, so property 2(c) still holds.

For the second case of the legal state definition, we consider any closed trajectory τ such that $x = \tau.fstate$. Let x' be $\tau.lstate$. We must show that $x' \in L^3_{TOBimpl}$. It is easy to see that because the only evolving variables referenced in property 2 are $clock_p$ and now, with the trajectory stopping conditions of both *RW*$\|$*Pbcast*, forcing updates at nodes and delivery of messages or drops of those messages within d_{phys} time, and ***TOBimpler*** on line 35, forcing processing of messages from $incoming$ whenever exactly d time has passed since broadcast, properties 2 and 3 will remain true throughout a trajectory. ∎

Legal state set $L_{TOBimpl}$:

The final legal set is a subset of $L^3_{TOBimpl}$ that satisfies an additional property about the entries of any $incoming$ set with respect to the state of *RW*$\|$*Pbcast*.

Definition 9.11 *$L_{TOBimpl}$ is the set of states x of $TOBimpl$ where all of the following hold:*

1. $x \in L^3_{TOBimpl}$.

2. *For each* $p \in P : (\neg failed_p \wedge updates_p \neq \emptyset) \Rightarrow \forall \langle m,s,t,f,b,r\rangle \in incoming_p : \exists P' \subseteq P - \{p\} : \langle\langle m,s,t,f,b,r\rangle, t, P'\rangle \in pbcastq(s)$.

   This says that any tuple in a process's $incoming$ must be a tuple that was actually handled for the process by $RW\|Pbcast$ and sent by the process whose id is the source tag in the message at the time indicated by the timestamp of the message.

Lemma 9.12 *$L_{TOBimpl}$ is a legal set for $TOBimpl$.*

Proof: Let x be any state in $L_{TOBimpl}$. By Definition 3.12 of a legal set, we must verify two things for state x:

- For each state x' of $TOBimpl$ and action a of $TOBimpl$ such that (x, a, x') is in the set of discrete transitions of $TOBimpl$, state x' is in $L_{TOBimpl}$.
- For each state x' and closed trajectory τ of $TOBimpl$ such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in $L_{TOBimpl}$.

By Lemma 9.10, we know that if x satisfies the first property of $L_{TOBimpl}$, then any discrete transition of $TOBimpl$ will lead to a state x' that still satisfies the first property, and any closed trajectory starting with state x will end in some state that satisfies the first property. This implies that we just need to check that in the two cases of the legal set definition, the state x' satisfies the second property of $L_{TOBimpl}$.

For the first case of the legal set definition, we could consider each action, but the only non-trivial one to examine is brcv:

- brcv$(\langle m, s, t, f, b, r\rangle)_p$: For this action to occur, by the precondition for this output in *RW*$\|$*Pbcast* and property 2(a) of $L^3_{TOBimpl}$, an appropriately tagged version of this tuple must have been in $pbcastq(s)$. Hence, if the tuple is added to $incoming_p$ in this action, then by the above observation, property 2 will hold.

For the second case of the legal state definition, we consider any closed trajectory τ such that $x = \tau.fstate$. It is easy to see that because the only evolving variables referenced in property 2 are $clock_p$ and now, property 2 will remain true throughout a trajectory. ■

A trivial observation is that an initial state of ***TOBimpl*** is in $L_{TOBimpl}$:

Lemma 9.13 *An initial state of $TOBimpl$ is in $L_{TOBimpl}$.*

9.3.2 Simulation relation

Here we show that $Start(TOBimpl, L_{TOBimpl})$ implements $Start(TOBspec, reachable_{TOBspec})$ (Lemma 9.19). We do this by first describing a simulation relation $\mathcal{R}_{TOB}$ from our implementation of the totally ordered broadcast

service, ***TOBimpl***, to the TIOA specification of the totally ordered broadcast service, ***TOBspec***. We prove that $\mathcal{R}_{TOB}$ is a simulation relation in Lemma 9.15, and then conclude that ***TOBimpl*** implements ***TOBspec*** (Theorem 9.16). In other words, we conclude that the traces of our implementation are traces of totally ordered broadcast. We then show in Lemma 9.18 that for each state in $L_{TOBimpl}$ there exists some reachable state of ***TOBspec*** that is related to it under $\mathcal{R}_{TOB}$.

You may notice in the definition below that for $x\mathcal{R}_{TOB}y$ to hold, state x must be a state in the legal set $L_{TOBimpl}$. This constrains the simulation relation to only be concerned with implementation states which we will later show are related to reachable states of $TOBspec$ (see Lemma 9.18).

Definition 9.14 *$\mathcal{R}_{TOB}$ is a relation between states of $TOBimpl$ and states of $TOBspec$ such that if x is a state of $TOBimpl$ and y is a state of $TOBspec$, then $x\mathcal{R}_{TOB}y$ exactly when the following conditions are satisfied:*

1. *$x \in L_{TOBimpl}$ and $x(RW) = y(RW)$.*

   This says that our relation only holds for state pairs where the state of our implementation is in the legal set $L_{TOBimpl}$ and the RW state is equal in x and y.

2. *$y \in Inv_{TOBspec} \wedge y(procs) = P$.*

   This says that y must be a reachable state of $TOBspec$, and that $y(procs)$ is full.

3. *For each $p \in P$, $x(failed_p) = y(failed_p)$.*

   This says that the failure status of each process is the same in the two states.

4. *For each $p \in P$, $\neg x(failed_p) \Rightarrow [(x(updates_p) = \emptyset \wedge y(rtimer_p) = \bot)$*
   $\vee \exists \langle u, t\rangle \in x(updates_p) : ([t \leq x(now) - d \wedge y(rtimer_p) = d]$
   $\vee [\forall \langle v, t'\rangle \in x(updates_p) : t' \geq t \wedge t = x(now) - y(rtimer_p)])]$.

   This says that the stored updates for each non-failed process corresponds to the $rtimer$. In particular, $rtimer$ is $\bot$ when $updates_p$ is empty, and either $rtimer$ is as old as the first pair in $updates$, or both are at least d old.

5. *For each $p \in P$, $(\neg x(failed_p) \wedge x(updates_p) \neq \emptyset)$*

$\Rightarrow [x(outgoing_p^-) = y(to_send_p^-) \wedge x(outgoing_p^+) = y(to_send_p^+)]$.

This says that corresponding $outgoing$ *and* to_send *queues are equal.*

6. *Let* $\langle\langle m_1, p_1, t_1, f_1, b_1, u_1\rangle, t_1, P_1\rangle, \cdots, \langle\langle m_n, p_n, t_n, f_n, b_n, u_n\rangle, t_n, P_n\rangle$ *be the subset of* $\bigcup_{p\in P} x(pbcastq(p))$ *with* $t_i > now - d$*, ordered by* $\langle t_i, p_i, f_i, b_i\rangle$*.*

   Then $y(sent) = \langle m_1, u_1, p_1, t_1\rangle, \cdots, \langle m_n, u_n, p_n, t_n\rangle$*.*

   This says that the sequence $sent$ *in* $TOBspec$ *is the same as the sequence of restricted message tuples in* $pbcastq$ *that are less than* d *old and then sorted by tags.*

7. *For each* $p \in P$*, let* $\langle m_1, p_1, t, f_1, b_1, u_1\rangle, \cdots, \langle m_n, p_n, t, f_n, b_n, u_n\rangle$ *be the subset of* $x(incoming_p)$ *with* $t = now - d$*, ordered by* $\langle p_i, f_i, b_i\rangle$*.*

   If $\neg x(failed_p) \wedge x(updates_p) \neq \emptyset$*, then* $y(to_rcv_p) = \langle m_1, u_1\rangle, \cdots, \langle m_n, u_n\rangle$*.*

   This says that for a non-failed process with a non-$\perp$ *region, the sequence of message and region pairs in* to_rcv *in* $TOBspec$ *is the same as the sequence of message and region pairs from tuples in* $incoming_p$ *that are exactly* d *old and then sorted by tags.*

Now we show that $\mathcal{R}_{TOB}$ is a simulation relation from ***TOBimpl*** to ***TOBspec***:

Lemma 9.15 *$\mathcal{R}_{TOB}$ is a simulation relation from $TOBimpl$ to $TOBspec$.*

Proof: By definition of a simulation relation we must show three things for all states of the two automata:

1. We must show that for any $x \in \Theta_{TOBimpl}$ there exists a state $y \in \Theta_{TOBspec}$ such that $x\mathcal{R}_{TOB}y$. There is one unique initial non-*failed* and non-*loc* state for both the first and the second automaton, and any values of *failed* and *loc* for each $p \in P$ is possible for either automaton. It is easy to check that $\mathcal{R}_{TOB}$ holds between any two such states.

2. Say that $x \in Q_{TOBimpl}$ and $y \in Q_{TOBspec}$, and that $x\mathcal{R}_{TOB}y$. Then for any action $a \in A_{TOBimpl}$, if ***TOBimpl*** performs action a and the state changes from x to x', we must show that there exists a closed execution fragment β of ***TOBspec*** with $\beta.fstate = y, trace(\beta) = trace(\wp(x)a\wp(x'))$, and $x'\mathcal{R}_{TOB}\beta.lstate$.

   By Lemma 9.12, Property 1 of $\mathcal{R}_{TOB}$ holds in x'.

For the other properties, we consider each action:

- drop: Let β be the point trajectory $\wp(y)$. It is trivial to see that $x'\mathcal{R}_{TOB}y$ and that the trace of both β and α are empty.
- fail_p and $\mathsf{restart}_p$: These are trivial.
- $\mathsf{tocast}(m)_p$: Let β be $\wp(y)$ $\mathsf{tocast}(m)_p$ $\wp(y')$. It is easy to see the trace is the same. If $x(updates_p) \neq \emptyset$ then since the same message is added to the end of $outgoing_p^+$ in ***TOBimpl*** and to $to_send_p^+$ in ***TOBspec***, then $x'\mathcal{R}_{TOB}y'$. Otherwise, $x(updates_p)$ is empty and we can trivially conclude that $x'\mathcal{R}_{TOB}y'$.
- $\mathsf{torcv}(m)_p$: Let β be $\wp(y)$ $\mathsf{torcv}(m)_p$ $\wp(y')$. We need to check that this action is enabled in y. If the action is enabled in the implementation, then there is an associated tuple in $incoming_p$ with timestamp $t \leq clock_p - d$ and with a tag which is smaller than the tags of all others in the set. By property 2(b) of $L^1_{TOBimpl}$, t is at least $clock_p - d$, implying it is equal to $clock_p - d$. Since xRy, this must mean that the tuple is the head of to_rcv_p. Hence, this action is enabled in the specification.

  We now note that $trace(\alpha) = trace(\beta)$ and that it is easy to see that $x'\mathcal{R}_{TOB}y'$: since $x\mathcal{R}_{TOB}y$ and the associated tuple is removed from $incoming_p$ and the corresponding tuple is removed from $y(to_rcv_p)$, $x'\mathcal{R}_{TOB}y'$ must hold.
- $\mathsf{GPSupdate}(l, t)_p$: Let β be $\wp(y)$ $\mathsf{GPSupdate}(l, t)_p$ $\wp(y')$. Let v be $region(l)$. It is easy to see that the traces of α and β are equal. To see that $x'\mathcal{R}_{TOB}y'$, we first note that properties 1-3 and 6 are easy to see hold. We consider several cases for the other properties: If $x(failed_p)$, then checking that the properties hold is trivial. So we consider where $\neg x(failed_p)$.

  Say the conditional on lines 44-45 holds. Since property 1 holds, we know that the only way for the conditional to hold is if $x(updates_p) = \emptyset$. Since $x\mathcal{R}_{TOB}y$, we know by property 4 of $\mathcal{R}_{TOB}$ that $y(rtimer_p) = \perp$, which by property 2 of $\mathcal{R}_{TOB}$ means that $TOBDelay_p.updated =$ false and $to_send_p^- = to_send_p^+ = \lambda = to_rcv_p$. In state x', $outgoing_p^- = outgoing_p^+ = \lambda$, $\emptyset = incoming_p$ and $updates_p = \{\langle v, t\rangle\}$. In state y', $to_send_p^- = to_send_p^+ = \lambda = to_rcv_p$ still,

satisfying properties 5 and 7. Also, in state y', $rtimer_p = 0$, satisfying property 4.

Now we check the other case, where the conditional on lines 44-45 does not hold. We know that in state x, $updates_p$ is not empty. In this case, the only changes between x and x' are that $updates_p$ in x' also contains the pair $\langle v, t\rangle$, and any tuples in $incoming_p$ that don't satisfy $regSpan$ are removed; it is easy to see that these will simply be those with region tags not equal to v or a neighboring region. It is trivial to see that property 5 still holds. For property 4, notice that in state y', $rtimer_p$ is not different from what it was in state y. Also, we can choose the same $\langle u, t'\rangle$ in $x(updates_p)$ to satisfy property 4 in state x'. Finally, to check property 7, notice that $TOBDelay_p$ removes any pair without a region that is the same or neighboring v. Hence, property 7 still holds.

- brcv$(\langle m, s, t, f, b, r\rangle)_p$: Let β be $\wp(y)$. It is easy to see that the traces are the same, and that the possible addition of an element to $incoming_p$ doesn't affect any properties since by property 2(a) of $L^3_{TOBimpl}$ and property 3 of Inv_{Pbcast}, $t > x(clock_p) - d$.

- bcast$(\langle m, q, t, f, b, r\rangle)_p$: Let β be $\wp(y)$ tocast$'(m, c)_p$ $\wp(y')$, where c is true iff $y(to_send^-_p)$ is empty, and the tuple is added to $sent$ so that any other tuples for time t from p or any smaller id process is before the point of addition, and any tuples for time t from a larger id process is after the point. We first check that tocast$'$ is enabled in y. Since $x\mathcal{R}_{TOB}y$ and bcast is enabled in x, $\neg y(failed_p)$.Also, $x(outgoing^-_p) = y(to_send^-_p)$ and $x(outgoing^+_p) = y(to_send^+_p)$, meaning the same message is transmitted. Hence, tocast$'$ is enabled.

  Now we check that $x'\mathcal{R}_{TOB}y'$ holds. This is easy for property 5 since the heads of two corresponding equal queues will be removed to leave new corresponding equal queues. The only other property to check is 6. We must check that the tags in the tuple added to $pbcastq(p)$ are the largest in the set, ordering the tuple after previously sent tuples by p. This is ensured through the fact that x satisfies

property 2(d) of $L^2_{TOBimpl}$. By our condition on the way in which the tuple is added to $sent$, we know that order is preserved between different senders.

3. Say that $x \in Q_{\mathit{TOBimpl}}, y \in Q_{TOBspec}$, and $x\mathcal{R}_{TOB}y$. Let α be an execution fragment of ***TOBimpl*** consisting of one closed trajectory, with $\alpha.fstate = x$.

   We must show that there is a closed execution fragment β of ***TOBspec*** with $\beta.fstate = y, trace(\beta) = trace(\alpha)$, and $\alpha.lstate\mathcal{R}_{TOB}\beta.lstate$.

   Let p_1 be the first id in P, p_2 be the second, etc. Let $\langle m_1, u_1, q_1, t_1\rangle$, $\langle m_2, u_2, q_2, t_2\rangle, \cdots \langle m_n, u_n, q_n, t_n\rangle$ be the $y(sent)$ prefix containing all tuples with $t_i \leq \alpha.lstate(now) - d$.

   Then β is the execution fragment $\tau_1\ a^1_{p_1}\ \tau_{1,1}\ a^1_{p_2}\ \tau_{1,2}\cdots\ a^1_{p_{|P|}}\ \tau_2\ a^2_{p_1}\ \cdots\ a^n_{p_{|P|}}\ \tau_{n+1}$, where $\beta.ltime = \alpha.ltime$, $t_i + d = \tau_i.lstate(now)$, and $a^i_{p_j} \in \{\mathsf{torcv}'(m_i, u_i)_{p_j}, \mathsf{drop(p_j)}\}$, for all i from 1 to n and j in 1 to $|P|$. We select $a^i_{p_j}$ to be torcv' if $\exists\langle v, t\rangle \in \tau_i.updates(p_j) : t \leq now - d \wedge \forall\langle v', t'\rangle \in updates(p) : (t' \geq t \Rightarrow v' \in nbrs^+(u))$, and drop otherwise.

   In other words, β is an execution fragment where torcv'_p and drop events are added in order of process id for each message in the $sent$ queue that is exactly d old. In order to satisfy properties 2, 6 and 7 of the relation $\mathcal{R}_{TOB}$, our construction ensures that in the last state of β no action torcv' or drop is enabled.

   It is obvious that the traces of α and β are the same. It is also easy to see that by construction, each torcv' and drop action will be enabled, and that $\alpha.lstate\mathcal{R}_{TOB}\beta.lstate$.

■

The following theorem concludes that our implementation of the totally ordered broadcast service implements $TOBspec$.

Theorem 9.16 $TOBimpl \leq TOBspec$.

Proof: This follows directly from the previous lemma and Corollary 2.23. ■

One useful observation about the proof that $\mathcal{R}_{TOB}$ is a simulation relation is the following, which says that, given any execution fragment α of $TOBimpl$ started in the legal set $L_{TOBimpl}$ and a state y of $TOBspec$ that is related to the first state of α, there is an execution fragment of $TOBspec$ starting in state y that not only has the same trace as α but also has the same RW and $Fail$-related projections as those of α (This is very useful later, when reasoning about the $Fail$-transformed composition of the totally ordered broadcast implementation pieces with pieces of other services):

Lemma 9.17 *Let α be in $frags_{TOBImpl}^{L_{TOBimpl}}$ and y be a state in $reachable_{TOBspec}$ such that $\alpha.fstate \mathcal{R}_{TOB} y$. Then there exists α' in $frags_{TOBspec}$ such that:*

1. $\alpha'.fstate = y$.
2. $trace(\alpha) = trace(\alpha')$.
3. *If α is a closed execution fragment, then $\alpha.lstate \mathcal{R}_{TOB} \alpha'.lstate$.*
4. $\alpha \lceil (A_{RW}, V_{RW}) = \alpha' \lceil (A_{RW}, V_{RW})$.
5. *For each $p \in P$, $\alpha \lceil (\{\mathsf{fail}_p, \mathsf{restart}_p\}, \{failed_p\}) = \alpha' \lceil (\{\mathsf{fail}_p, \mathsf{restart}_p\}, \{failed_p\})$.*

The first three properties of the lemma follow from the fact that $\mathcal{R}_{TOB}$ is a simulation relation, while the last two properties follow from the construction of the matching execution of $TOBspec$ in the proof that $\mathcal{R}_{TOB}$ is a simulation relation, which preserves the actions and variables of RW and each of the processes' $Fail$-transform variables and actions.

Now, as mentioned previously, we tie the legal states $L_{TOBimpl}$ to reachable states of ***TOBspec***. In particular, we show that each state in $L_{TOBimpl}$ is related to some reachable state of $TOBspec$.

Lemma 9.18 *For any state $x \in L_{TOBimpl}$, there exists a state $y \in reachable_{TOBspec}$ where $x \mathcal{R}_{TOB} y$.*

Proof: We prove this lemma by showing how, given a state $x \in L_{TOBimpl}$, we can construct a state y of ***TOBspec*** such that $x \mathcal{R}_{TOB} y$. We do this by describing the state of the components of state y. We then check that the constructed state y is one such that $y \in Inv_{TOBspec}$ and $x \mathcal{R}_{TOB} y$ holds.

1. $y(RW) = x(RW)$.

   This says that the RW component is the same in both x and y.

2. For each $p \in P$: $y(TObcast.updates(p)) = \{\langle region(l,t)\rangle \mid \langle l,t\rangle \in x(RW.updates(p))\}$.

   This says that the $updates$ should correspond between RW and $TObcast$.

3. $y(TObcast.now) = x(RW.now)$, and $y(procs) = P$.

   This says that the realtime should correspond between $TObcast$ and RW and that $procs$ should always be full.

4. For each $p \in P$, $x(failed_p) = y(failed_p)$.

   This says that the fail status of the processes should match between the states.

5. For each $p \in P$, if $x(failed_p)$ then $y(TOBFilter_p)$ and $y(TOBDelay_p)$ are arbitrary.

   This says that for failed processes the state of the $TOBFilter$ and $TOBDelay$ components are arbitrary.

6. For each $p \in P$, if $\neg x(failed_p)$ and $x(updates_p) = \emptyset$, then: $\neg y(updated_p)$, $y(to_send_p^-) = y(to_send_p^+) = y(to_rcv_p) = \lambda$, and $y(rtimer_p) = \bot$.

   This says that if a process is not failed and has an empty $updates$ in x, then in state y $updated$ is false, $rtimer$ is $\bot$, and the to_send^-, to_send^+, and to_rcv queues are empty.

7. For each $p \in P$, if $\neg x(failed_p)$ and $x(updates_p) \neq \emptyset$, then:

   - $y(updated_p)$.

     This says that if a process is not failed and $updates$ is not empty in x, then $updated$ is true for the process in y.

   - $y(to_send_p^-) = x(outgoing_p^-)$ and $y(to_send_p^+) = x(outgoing_p^+)$.

     This says that if a process is not failed and $updates$ is not empty in x, then the process's $outgoing$ queues correspond to their counterpart to_send queues in y.

- Let $t = min(\{t^* \in \mathbb{R}^{\geq 0} \mid \exists u \in U : \langle u, t^* \rangle \in x(updates_p)\})$.

  Then $y(rtimer_p) = min(d, x(now) - t)$.

  This says that if a process is not failed and $updates$ is not empty in x, then $rtimer$ in y is as old as the first pair in the process's $updates$ in state x, or both are at least d old.

- Let $\langle m_1, p_1, t, f_1, b_1, u_1 \rangle, \cdots, \langle m_n, p_n, t, f_n, b_n, u_n \rangle$ be the subset of $x(incoming_p)$ with $t = x(now) - d$, ordered by $\langle p_i, f_i, b_i \rangle$.

  Then $y(to_rcv_p) = \langle m_1, u_1 \rangle, \cdots, \langle m_n, u_n \rangle$.

  This is the same as property 7 of Definition 9.14.

8. Let $\langle \langle m_1, p_1, t_1, f_1, b_1, u_1 \rangle, t_1, P_1 \rangle, \cdots, \langle \langle m_n, p_n, t_n, f_n, b_n, u_n \rangle, t_n, P_n \rangle$ be the subset of $\bigcup_{p \in P} x(pbcastq(p))$ with $t_i > x(now) - d$, ordered by $\langle t_i, p_i, f_i, b_i \rangle$.

   Then $y(sent) = \langle m_1, u_1, p_1, t_1 \rangle, \cdots, \langle m_n, u_n, p_n, t_n \rangle$.

   This is the same as property 6 of Definition 9.14.

9. Let $\langle \langle m_1, p_1, t, f_1, b_1, u_1 \rangle, t, P_1 \rangle, \cdots, \langle \langle m_n, p_n, t, f_n, b_n, u_n \rangle, t_n, P_n \rangle$ be the subset of $\bigcup_{p \in P} x(pbcastq(p))$ with $t = x(now) - d$, ordered by $\langle p_i, f_i, b_i \rangle$.

   Then $y(oldsent) = \langle m_1, u_1, p_1, t \rangle, \cdots, \langle m_n, u_n, p_n, t \rangle$.

   This says that $oldsent$ in y is calculated from d-old $pbcastq$ messages.

Next we show that $y \in Inv_{TOBspec}$. We check each property of $Inv_{TOBspec}$ (Definition 9.1) in state y. Properties 1-5 and 7(a) of $Inv_{TOBspec}$ are trivial to check. Property 6 of $Inv_{TOBspec}$ holds in y because state x satisfies properties 2(a) and 2(b) of $L^2_{TOBimpl}$ and because of properties 8 and 9 in the construction above. To see this, notice that by properties 8 and 9 above, the concatenation of $oldsent$ and $sent$ in state y is the sequence of $pbcastq$ messages in state x with timestamps up to d old, in order of the timestamp, sender, attached Boolean, and sequence number of the message tuple. Properties 2(a) and 2(b) of $L^2_{TOBimpl}$ guarantee that those tuples in state x satisfy the region ordering property described in property 6 of $Inv_{TOBspec}$.

For the remainder of property 7 of $Inv_{TOBspec}$, we provide pointers to the properties of state x and the construction that imply the property. Property 7(b) of $Inv_{TOBspec}$ holds in y because state x satisfies property 3 of $L^3_{TOBimpl}$ and because of the third bullet in

property 7 in the construction above. Property 7(c) of $Inv_{TOBspec}$ holds in y because state x satisfies property 2(d) of $L^1_{TOBimpl}$ and because of property 7 in the construction above. Property 7(d) of $Inv_{TOBspec}$ holds in y because state x satisfies property 2(c) of $L^3_{TOBimpl}$ and because of properties 7-9 in the construction above.

All that remains is to show that $x\mathcal{R}_{TOB}y$. We check each property of $\mathcal{R}_{TOB}$. Properties 1-3, 6, and 7 are trivial to check. Property 4 of $\mathcal{R}_{TOB}$ holds because of property 6 and the third bullet of property 7 in the construction above. Property 5 of $\mathcal{R}_{TOB}$ holds because of the second bullet of property 7 in the construction above.

By Theorem 9.4, we know that $Inv_{TOBspec} = reachable_{TOBspec}$, and we conclude that for any state x in $L_{TOBimpl}$, there is some reachable state y of ***TOBspec*** such that $x\mathcal{R}_{TOB}y$.

■

Now we can pull together the results in this section to finally conclude that $Start(TOBimpl, L_{TOBimpl})$ implements $Start(TOBspec, reachable_{TOBspec})$.

Lemma 9.19 $Start(TOBimpl, L_{TOBimpl}) \leq Start(TOBspec, reachable_{TOBspec})$.

Proof: By Lemma 9.15, $\mathcal{R}_{TOB}$ is a simulation relation from ***TOBimpl*** to ***TOBspec***. By Lemma 9.18, we know that for each state $x \in L_{TOBimpl}$, there is some reachable state y of ***TOBspec*** such that $x\mathcal{R}_{TOB}y$. Hence, by Corollary 2.22, $tracefrags^{L_{TOBimpl}}_{\textbf{\textit{TOBimpl}}} \subseteq tracefrags^{reachable_{TOBspec}}_{TOBspec}$, which implies the result. ■

9.3.3 Self-stabilization

We've seen that $L_{TOBimpl}$ is a legal set for $TOBimpl$, and that each state in $L_{TOBimpl}$ is related to a reachable state of $TOBspec$. Here we show that ***TOBimpler*** self-stabilizes to $L_{TOBimpl}$ relative to $R(RW\|Pbcast)$ (Theorem 9.24), meaning that if certain program portions of the implementation are started in an arbitrary state and run with $R(RW\|Pbcast)$, the resulting execution eventually gets into a state in $L_{TOBimpl}$. This is done in phases, corresponding to each legal set $L^1_{TOBimpl}$, $L^2_{TOBimpl}$, $L^3_{TOBimpl}$, and finally $L_{TOBimpl}$.

After we show that ***TOBimpler*** self-stabilizes to $L_{TOBimpl}$ relative to $R(RW\|Pbcast)$, we use the fact that $\mathcal{R}_{TOB}$ (see Definition 9.14) is a simulation relation that relates states

in $L_{TOBimpl}$ with reachable states of $TOBspec$ to conclude that after an execution of *TOBimpl* has stabilized, the trace fragment from the point of stabilization with bcast and brcv actions hidden is the suffix of some trace of *TOBspec* (see Theorem 9.25).

The first lemma describes the first phase of stabilization, for legal set $L^1_{TOBimpl}$:

Lemma 9.20 *Let t^1_{tob} be any t such that $t > \epsilon_{sample}$.*
$TOBimpler$ self-stabilizes in time t^1_{tob} to $L^1_{TOBimpl}$ relative to $R(RW\|Pbcast)$.

Proof: By definition of self-stabilization, we must show that $execs_{U(TOBimpler)\|R(RW\|Pbcast)}$ stabilizes in time t^1_{tob} to $frags^{L^1_{TOBimpl}}_{TOBimpler\|R(RW\|Pbcast)}$. By Corollary 3.11, the set $frags^{L^1_{TOBimpl}}_{TOBimpler\|R(RW\|Pbcast)}$ is the same as $frags^{L^1_{TOBimpl}}_{TOBimpl}$. By Lemma 3.21, we just need to show that for any length-t^1_{tob} prefix α of an element of $execs_{U(TOBimpler)\|R(RW\|Pbcast)}$, $\alpha.lstate$ is in $L^1_{TOBimpl}$. We examine each property of $L^1_{TOBimpl}$.

By Theorem 6.4, since the state of *RW*$\|$*Pbcast* in the first state of α is a reachable state of *RW*$\|$*Pbcast*, we know that property 1 of $L^1_{TOBimpl}$ holds in each state of α.

By the proof of Lemma 9.6, we know that for each $p \in P$, if property 2 of $L^1_{TOBimpl}$ holds for p in some state, it continues to hold for p in subsequent states. Consider the first GPSupdate$_p$ in α for some p and the state x in α immediately after the event. It is easy to see that property 2 holds for p in state x. Since $\alpha.ltime = t^1_{tob}$ and $t^1_{tob} > \epsilon_{sample}$, we know that for each $p \in P$ at least one GPSupdate$_p$ action occurs in α. Hence, for each $p \in P$, property 2 of $L^1_{TOBimpl}$ holds at $\alpha.lstate$.

We conclude that $\alpha.lstate$ is in $L^1_{TOBimpl}$. ■

Lemma 9.21 *Let t^2_{tob} be any t such that $t > 0$.*
$frags^{L^1_{TOBimpl}}_{TOBimpl}$ stabilizes in time t^2_{tob} to $frags^{L^2_{TOBimpl}}_{TOBimpl}$.

Proof: By Lemma 3.21, we just need to show that for any length-t^2_{tob} prefix α of an element of $frags^{L^1_{TOBimpl}}_{TOBimpl}$, $\alpha.lstate$ is in $L^2_{TOBimpl}$. We examine each property of $L^2_{TOBimpl}$.

By Lemma 9.6, since the first state of α is in $L^1_{TOBimpl}$, we know that property 1 of $L^2_{TOBimpl}$ holds in each state of α.

Notice that there must be some state x of α such that $x(now) = \alpha.fstate(now)$ and all actions that occur after x in α occur at a state with $now > x(now)$. Consider any

state y in α such that y occurs a non-0 amount of time after $\alpha.fstate$ and no actions occur between x and y. This means that there are no tuples in $y(pbcastq(p))$ that were sent at the time $y(now)$ and no tuples in $y(outgoing^-_p)$ or $y(outgoing^+_p)$, meaning that property 2 is trivially satisfied. This allows us to conclude that property 2 of $L^2_{TOBimpl}$ holds at y and hence, by Lemma 9.8, at $\alpha.lstate$.

We conclude that $\alpha.lstate$ is in $L^2_{TOBimpl}$. ∎

Lemma 9.22 *Let t^3_{tob} be any t such that $t > d$. $frags^{L^2_{TOBimpl}}_{TOBimpl}$ stabilizes in time t^3_{tob} to $frags^{L^3_{TOBimpl}}_{TOBimpl}$.*

Proof: By Lemma 3.21, we just need to show that for any length-t^3_{tob} prefix α of an element of $frags^{L^2_{TOBimpl}}_{TOBimpl}$, $\alpha.lstate$ is in $L^3_{TOBimpl}$. We examine each property of $L^3_{TOBimpl}$.

By Lemma 9.8, since the first state of α is in $L^2_{TOBimpl}$, we know that property 1 of $L^3_{TOBimpl}$ holds in each state of α.

For property 2, based on the proof of Lemma 9.10, the property can be considered as a conjunction of separate statements, one for each possible time. It is also not difficult to see that for any state x in α and time t larger than $\alpha.fstate(now)$, property 2 holds for messages sent at time t. Hence, in order to ensure that property 2 as a whole holds at state $\alpha.lstate$, we need that property 2 holds at $\alpha.lstate$ for all times up to d time before $\alpha.lstate(now)$. This is satisfied because $\alpha.ltime > d$.

For property 3, we know that in $\alpha.fstate$, any non-failed process with non-empty $updates_p$ has its latest update in $updates_p$ correspond to its latest update at RW. After d time passes, that particular latest update satisfies the requirements of the $\langle u, t\rangle$ in property 3, if the process has not failed in the meantime. If the process has been failed in the meantime or was failed in $\alpha.fstate$, then it will have an $updates_p$ set consistent with the updates of RW starting from after it awakens.

We conclude that $\alpha.lstate$ is in $L^3_{TOBimpl}$. ∎

Lemma 9.23 *$frags^{L^3_{TOBimpl}}_{TOBimpl}$ stabilizes in time d to $frags^{L_{TOBimpl}}_{TOBimpl}$.*

Proof: By Lemma 3.21, we just need to show that for any length-d prefix α of an element of $frags^{L^3_{TOBimpl}}_{TOBimpl}$, $\alpha.lstate$ is in $L_{TOBimpl}$. We examine each property of $L_{TOBimpl}$.

By Lemma 9.10, since the first state of α is in $L^3_{TOBimpl}$, we know that property 1 of $L_{TOBimpl}$ holds in each state of α.

It is plain that for any state in α, any new tuple added to an $incoming$ queue for a process will satisfy property 2 of $L_{TOBimpl}$. Consider any $p \in P$ and tuple $\langle m, s, t, f, b, r\rangle \in incoming_p$ in $\alpha.fstate$. By property 2(b) of $L^1_{TOBimpl}$, we know that $t < \alpha.fstate(clock_p)$. By our stopping conditions on line 35, this tuple will be removed when $clock_p = t + d$. Hence, the tuple will be removed in less than d time. This holds for any process p and any tuple in $\alpha.fstate(incoming_p)$. This implies that in $\alpha.lstate$, property 2 will hold.

We conclude that $\alpha.lstate$ is in $L_{TOBimpl}$. ■

Theorem 9.24 *Let t_{tob} be any t such that $t > 2d + \epsilon_{sample}$.*
$TOBimpler$ self-stabilizes in time t_{tob} to $L_{TOBimpl}$ relative to $R(RW\|Pbcast)$.

Proof: We must show that $execs_{U(TOBimpler)\|R(RW\|Pbcast)}$ stabilizes in time t_{tob} to $frags^{L_{TOBimpl}}_{TOBimpler\|R(RW\|Pbcast)}$. By Corollary 3.11, $frags^{L_{TOBimpl}}_{TOBimpler\|R(RW\|Pbcast)}$ is the same as $frags^{L_{TOBimpl}}_{TOBimpl}$. The result follows from the application of Lemma 3.7 to the four lemmas (Lemmas 9.20-9.23) above. Let $t^1_{tob} = \epsilon_{sample} + (t_{tob} - 2d - \epsilon_{sample})/3$, $t^2_{tob} = (t_{tob} - 2d - \epsilon_{sample})/3$, and $t^3_{tob} = d + (t_{tob} - 2d - \epsilon_{sample})/3$. (These terms are chosen so as to satisfy the constraints that $t^1_{tob} > \epsilon_{sample}$, $t^2_{tob} > 0$, and $t^3_{tob} > d$, as well as the constraint that $t^1_{tob} + t^2_{tob} + t^3_{tob} + d = t_{tob}$.)

Let B_0 be $execs_{U(TOBimpler)\|R(RW\|Pbcast)}$, B_1 be $frags^{L^1_{TOBimpl}}_{TOBimpl}$, B_2 be $frags^{L^2_{TOBimpl}}_{TOBimpl}$, B_3 be $frags^{L^3_{TOBimpl}}_{TOBimpl}$, and B_4 be $frags^{L_{TOBimpl}}_{TOBimpl}$ in Lemma 3.7. Let t_1 be t^1_{tob}, t_2 be t^2_{tob}, t_3 be t^3_{tob}, and t_4 be d in Lemma 3.7. Then by Lemma 3.7 and Lemmas 9.20-9.23, $execs_{U(TOBimpler)\|R(RW\|Pbcast)}$ stabilizes in time $t^1_{tob}+t^2_{tob}+t^3_{tob}+d = t_{tob}$ to $frags^{L_{TOBimpl}}_{TOBimpl}$.

We conclude that $TOBimpler$ self-stabilizes in time t_{tob} to $L_{TOBimpl}$ relative to $R(RW\|Pbcast)$. ■

As promised, we can now conclude that an execution of *TOBimpl* eventually reaches a point such that the trace of the execution from that point on is the same as the suffix of some trace of the specification.

Theorem 9.25 *Let t_{tob} be any t such that $t > 2d + \epsilon_{sample}$.*
$traces_{\mathsf{ActHide}(H_{TOimpl}, U(TOBimpler)\|R(RW\|Pbcast))}$ stabilizes in time t_{tob} to $traces_{R(TOBspec)}$.

Proof: By Lemma 9.19, we know that $tracefrags_{\textit{\textbf{TOBimpl}}}^{L_{TOBimpl}} \subseteq tracefrags_{TOBspec}^{reachable_{TOBspec}}$. By Theorem 9.24, we know that $execs_{U(TOBimpler)\|R(RW\|Pbcast)}$ stabilizes in time t_{tob} to $frags_{TOBimpler\|R(RW\|Pbcast)}^{L_{TOBimpl}}$. By Lemma 3.10, $frags_{TOBimpler\|R(RW\|Pbcast)}^{L_{TOBimpl}}$ is the same as $frags_{\textit{\textbf{TOBimpl}}}^{L_{TOBimpl}}$. By Lemma 3.5, this implies that $traces_{\mathsf{ActHide}(H_{TOimpl},U(TOBimpler)\|R(RW\|Pbcast))}$ stabilizes in time t_{tob} to $tracefrags_{\textit{\textbf{TOBimpl}}}^{L_{TOBimpl}}$.

Since $tracefrags_{\textit{\textbf{TOBimpl}}}^{L_{TOBimpl}} \subseteq tracefrags_{TOBspec}^{reachable_{TOBspec}}$, we conclude that the set of traces of $\mathsf{ActHide}(H_{TOimpl}, U(TOBimpler)\|R(RW\|Pbcast))$ stabilizes in time t_{tob} to $tracefrags_{TOBspec}^{reachable_{TOBspec}}$, which is the same as $traces_{R(TOBspec)}$. ■

Chapter 10

Leader election service

In order to simplify the implementation of the VSA layer, it is useful to have access to a leader election service that allows nodes in the same region to periodically compete to be named sole leader of the region for some time. In this chapter, we describe the specification and implementation for a stabilizing round-based leader election service used in our emulator implementation. We then show that our implementation is correct and that it is self-stabilizing.

10.1 *LeadSpec*: Specification of the leader election service

We describe the specification of our leader election service as an algorithm in two parts: $LeadMain$ and $LeadCl_p, p \in P$ (see Figure 10-1). The specification of the leader election service is then $LeadSpec$, which is equal to $LeadMain \| RW$ composed with $Fail(LeadCl_p)$ for all $p \in P$, with certain actions hidden.

Notice that the $LeadCl$ machines are for individual processes. In this thesis we are interested in considering $Fail$ -transformed mobile nodes. Separating the $LeadCl$ machines from *LeadMain* allows us to *Fail*-transform portions of *LeadSpec*. As with $TOBDelay$ and $TOBFilter$ in Chapter 9, separating the leader election service into a *Fail*-oblivious central component and *Fail*-transforming individual components makes it easier to use *Fail*-transform theory from Chapter 5.

Our leader election service is a round-based service that collects information from po-

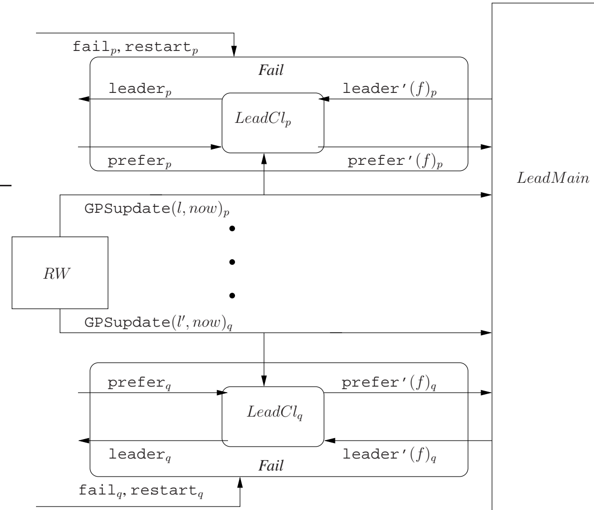


Figure 10-1: Leader election service. A *LeadCl* for a client performs a prefer$'(f)$ to indicate that its client should be considered by *LeadMain* as the leader of its client's region. *LeadMain* determines the winners of the leader competition for each region and communicates the results to each *LeadCl*. A winning process's *LeadCl* might then produce a leader output to its client, indicating the client is a leader.

tential leaders at the beginning of each round, determines up to one leader per region, and performs leader outputs for those leaders that remain alive and in their regions up to when the round is exactly d old. We assume that rounds are of length t_{slice}, where $t_{slice} > 2d + \epsilon$. Rounds begin on multiples of t_{slice}. A new leader competition for each region begins fresh (remembering none of the prior round's leaders or nominations) at the start of each round. This simple round-based structure, with little information remembered from one round to the next, is helpful when discussing stabilization in Section 10.3.3.

LeadMain is the central decision-making portion of the leader election service, collecting nominations from processes for leadership, and determining leaders for each region from these nominations. $LeadCl_p$ sits between *LeadMain* and a process p. At the start of each round, it communicates with *LeadMain* to nominate its process as the current round's leader for its region by providing Boolean priority inputs to *LeadMain*, letting it know that the process it represents is an alive process with knowledge of its region, and hence competing for leadership. If it received an indication from its process that its entry should be favored, the Boolean it communicates is true. Otherwise, the Boolean is false. These Boolean priorities are later used by the emulation algorithm (Section 11.2) to communicate

```
1  Signature:
     Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^{≥0}, p ∈ P
3    Input prefer′(val)_p, val ∈ Bool, p ∈ P
     Internal reset
5    Output leader′(val)_p, val ∈ Bool, p ∈ P

7  State:
     analog now: ℝ^{≥0}, initially 0
9    reg: P → U ∪ {⊥}, initially ⊥ for all p ∈ P
     pref: P → Bool, initially false for all p ∈ P
11   cand: U → (P ×Bool) ∪ {⊥}, initially ⊥ for all u ∈ U
     serviced: 2^P, initially ∅
13
   Trajectories:
15   evolve
       d(now) = 1
17   stop when
       (now mod t_slice = d_phys ∧ serviced ≠ P)
19     ∨ (now mod t_slice = d_phys+2ε ∧ serviced ≠ ∅)

21 Transitions:
     Input GPSupdate(l, t)_p
23   Effect:
       reg(p) ← region(l)

   Input prefer′(b)_p                                      26
   Effect:
     pref(p) ← b                                           28
     if ∃q ∈ P: ⟨q, b⟩ = cand(reg(p)) then
       cand(reg(p)) ← choose{cand(reg(p)), ⟨p, b⟩}         30
     else if b ∨ cand(reg(p)) = ⊥then
       cand(reg(p)) ← ⟨p, b⟩                               32

   Output leader′(val)_p                                   34
   Precondition:
     now mod t_slice ∈ (0, d_phys ] ∧ p ∉ serviced         36
     val ⇔ ⟨p, pref(p)⟩ = cand(reg(p))
   Effect:                                                 38
     serviced ← serviced ∪ {p}
                                                           40
   Internal reset
   Precondition:                                           42
     now mod t_slice > d_phys+ε ∧ serviced ≠ ∅
   Effect:                                                 44
     for all u ∈ U
       cand(u) ← ⊥                                         46
     for all p ∈ P
       pref(p) ← false                                     48
     serviced ← ∅
```

Figure 10-2: $LeadMain$, electing a leader.

whether the submitting process is currently emulating its local region's VSA.

LeadMain takes Boolean priority inputs at the beginning of the round from the *LeadCl*s, and each time such an input occurs, $LeadMain$ decides whether to replace whoever is the current winner for the input process's region with the new process, always selecting a process that submits a true value over one that submits a false value.

By the time the round is d old, *LeadMain* submits an input to each $LeadCl_p$ saying whether its process p is the round's leader for its region. If the input says it is and $LeadCl_p$ has a record of participating in the latest leader competition then it performs an output to let its process know that it is the leader.

We describe the $LeadMain$ and $LeadCl$ components in more detail below.

10.1.1 *LeadMain*

Here we provide a description of *LeadMain* (Figure 10-2), the central leader-deciding service. The interface of *LeadMain* consists of three kinds of actions:

- **Input** GPSupdate$(l, t)_p, l \in R, t \in \mathbb{R}^{\geq 0}, p \in P$: This input indicates that a process p is currently located at position l.

- **Input** $\mathsf{prefer}'(val)_p, val \in Bool, p \in P$: This input indicates that process p is proposing itself as a candidate to be leader of its current region. The Boolean val indicates whether the process should have priority in leader selection. (Later, we use this mechanism to give priority to processes in a region that are participating in emulation of their region's VSA (Section 11.2).)

- **Output** $\mathsf{leader}'(val)_p, val \in Bool, p \in P$: This output indicates to process p whether or not it was chosen as the leader for its current region. A true val indicates yes, while a false val indicates no.

The state variables are:

- $now : \mathbb{R}^{\geq 0}$: This variable is the real-time. It is initially 0.

- $reg(p) : U \cup \{\bot\}, p \in P$: This variable is the last reported region for each process, initially $\bot$. For each $\mathsf{GPSupdate}(l, t)_p$ input, the value $region(l)$ is stored in $reg(p)$.

- $pref(p) : Bool, p \in P$: This variable is the priority for the process p. For each $\mathsf{prefer}'(val)_p$ input, the value val is stored in $pref(p)$.

- $cand(u) : (P \times Bool) \cup \{\bot\}, u \in U$: This variable communicates who the current leader of the region is. It is initially $\bot$, but when a $\mathsf{prefer}'(b)_p$ occurs when $reg(p) = u$, it is updated to $\langle p, b\rangle$ if $cand(u)$ was $\bot$ or b is true and the current pair is false. If b matches the Boolean of a pair already in $cand(u)$ then $cand(u)$ may or may not be updated to $\langle p, b\rangle$.

- $services : 2^P$: This is a bookkeeping variable used by *LeadMain* to keep track of the processes for which a prefer' output has not yet occurred.

Whenever a $\mathsf{prefer}'(b)_p$ occurs (line 26) at the start of a round, *LeadMain* stores b as $pref(p)$ (line 28). Then it checks to see if p's region has a current candidate for leader. If not or if b is true and the current candidate tuple is false, the tuple $\langle p, b\rangle$ is stored as $cand(u)$ (lines 31-32). If b matches the Boolean in the current candidate tuple, then *LeadMain* nondeterministically decides whether or not to replace the current candidate tuple with $\langle p, b\rangle$ (lines 29-30).

```
1  Signature:
     Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^≥0
3    Input prefer_p
     Input leader′(val)_p, val ∈ Bool
5    Output prefer′(val)_p, val ∈ Bool
     Output leader_p
7
   State:
9    analog clock ∈ ℝ^≥0 ∪ {⊥}, initially ⊥
     reg: U ∪ {⊥}, initially ⊥
11   pref, participated: Bool, initially false

13 Trajectories:
     evolve
15     if clock ≠ ⊥ then
         d(clock) = 1
17     else constant clock
     stop when
19     (clock mod t_slice = 0 ∧ ¬ participated)
       ∨ (clock mod t_slice = d_phys + ϵ ∧ participated)
21
   Transitions:
23   Input GPSupdate(l, t)_p
     Effect:
25     if reg ≠ region(l) ∨ clock ≠ t then
         reg ← region(l)
27       clock ← t
         pref, participated ← false

     Input prefer_p                                              30
     Effect:
       if clock mod t_slice = 0 then                             32
         pref ← true
         participated ← false                                    34

     Output prefer′(val)_p                                       36
     Precondition:
       clock mod t_slice = 0 ∧ ¬ participated ∧ val = pref       38
     Effect:
       participated ← true                                       40

     Input leader′(val)_p                                        42
     Effect:
       if clock ≠ ⊥ ∧ (¬ val ∨ ¬ participated) then              44
         pref, participated ← false
                                                                 46
     Output leader_p
     Precondition:                                               48
       clock mod t_slice = d_phys + ϵ ∧ participated
     Effect:                                                     50
       pref, participated ← false
```

Figure 10-3: $LeadCl_p$, client portion for electing a leader.

After some non-zero amount of time into the round and no later than d_{phys} into a round, *LeadMain* services processes. For each process p not in $serviced$, it performs a leader′$(val)_p$ output, where val is true exactly when $cand(reg(p))$ is equal to the tuple $\langle p, pref(p)\rangle$ (lines 34-37). It then updates $serviced$ to contain p, indicating that it has been serviced (line 39).

After more than d into a round, *LeadMain* performs a reset, initializing $pref, cand$, and $serviced$ for the next leader election round (lines 41-49).

10.1.2 *LeadCl*

Here we provide a description of $LeadCl_p$. This piece communicates high priorities for leader election from a process to *LeadMain* and acts as an intermediary for communicating leadership decisions from *LeadMain* to a process. This piece is also the portion of *LeadSpec* that allows us to model the impact of failures. For example, *LeadMain* may choose as leader a process that has failed since the beginning of a round; *LeadCl* prevents that process from

becoming a leader.

Its interface consists of five kinds of actions:

- **Input** $\mathsf{GPSupdate}(l,t)_p, l \in R, t \in \mathbb{R}^{\geq 0}, p \in P$: This input indicates that a process p is currently located at position l.

- **Input** $\mathsf{prefer}_p, p \in P$: This input indicates that the process is to have priority in leader election. (As mentioned earlier, this is used in Section 11.2 by processes currently emulating their local VSA to indicate that they should be chosen as leader over processes that are not yet participating in VSA emulation.)

- **Input** $\mathsf{leader}'(val)_p, val \in Bool, p \in P$: This input indicates whether or not *Lead-Main* has chosen this process as the winning candidate for leader for p's current region.

- **Output** $\mathsf{prefer}'(val)_p, val \in Bool, p \in P$: This output is the process putting itself up for consideration as leader. The value val is true if a prefer has occurred in this round at the process.

- **Output** $\mathsf{leader}_p, p \in P$: This output communicates that a process is the leader for its current region.

Its state variables are the following:

- $clock : \mathbb{R}^{\geq 0} \cup \{\perp\}$: This is the process's local clock. It is initially $\perp$, but is set to the system's real-time when a $\mathsf{GPSupdate}$ occurs at the process.

- $reg : U \cup \{\perp\}$: This is the last reported region of the process since initialization.

- $pref : Bool$: This value indicates priority of the process. If a prefer occurs at the beginning of a round, this value is set to true and triggers a prefer' output. Otherwise, this value is false.

- $participated : Bool$: This indicates whether the process has participated in its current region's leader election via a prefer' action. It can be reset after it has participated if a prefer input occurs.

At the start of a round, $LeadCl_p$ performs a $\mathsf{prefer}'(pref)_p$ output, setting $participated$ to true so as to prevent additional such outputs (lines 36-40). It may also receive a prefer_p input (indicating that its client wants process p to have higher priority in the leader election competition), resulting in the setting of $pref$ to true and $participated$ to false, triggering a(nother) prefer' output (lines 30-34). Whenever a GPSupdate occurs at the process that changes its region or clock, $pref$ and $participated$ are set to false, preventing the process from later performing a leader output in the region it left (line 28).

Later, if it receives a $\mathsf{leader}'(val)_p$ input (line 42), if val is false (meaning it was not chosen as leader for its region) or if $clock = \perp$ (meaning it has restarted and has not yet received a GPSupdate) or $participated$ is false (meaning it has moved or restarted since the beginning of the round), then $LeadCl_p$ sets $pref$ and $participated$ to false, initializing those values for the next round (lines 44-45). Otherwise, it does nothing.

If, at exactly d into the round, $participated$ is still true (meaning that it did not receive a leader' input reporting it was not leader for its region) then $LeadCl_p$ performs a leader_p output (lines 47-49), and initializes $pref$ and $participated$ for the next round (line 51).

10.1.3 *LeadSpec*

As mentioned earlier, the full specification, *LeadSpec*, for the leader election service is equal to the composition of the central leader-choosing service and RW composed with the *Fail*-transformed $LeadCl$ portion for each process, with certain actions hidden:

Definition 10.1 *Let* $H_{Leadspec}$ *be* $\{\mathsf{leader}'(val)_p, \mathsf{prefer}'(val)_p \mid val \in Bool, p \in P\}$. *Then define* $LeadSpec$ *to be* $\mathsf{ActHide}(H_{Leadspec}, \prod_{p\in P} Fail(LeadCl_p)\|LeadMain\|RW)$.

Legal states of *LeadSpec*

Here we characterize a set of legal states for $LeadSpec$ by providing a list of properties describing those states. We then show that the set of states is legal.

Properties 1, 2, and 5 ensure that the state of RW is reachable and consistent with the state of *LeadMain*. Properties 3 and 4 describe some basic facts about the state of *LeadMain*

based on the age of a round. The remaining properties describe facts about states based on the value of each *LeadCl*.

Definition 10.2 *Define* $Inv_{LeadSpec}$ *to be the set of states* x *of* $LeadSpec$ *such that the following properties hold:*

1. $x\lceil X_{RW} \in Inv_{RW}$.

   This says that the RW *components are in a reachable state.*

2. $RW.now = LeadMain.now \wedge RW.reg = LeadMain.reg$.

   This says that the clock time and region mapping is the same between RW *and* $LeadMain$.

3. $RW.now \mod t_{slice} > d_{phys} + 2\epsilon$

   $\Rightarrow (serviced = \emptyset \wedge \forall u \in U : cand(u) = \bot \wedge \forall p \in P : \neg pref(p))$.

   This says that if the current round is greater than $d_{phys} + 2\epsilon$ *old, then* $LeadMain$*'s* $serviced, cand$*, and* $pref$ *variables are initialized.*

4. $RW.now \mod t_{slice} = 0 \Rightarrow serviced = \emptyset$ *and* $RW.now \mod t_{slice} \in (d_{phys}, d] \Rightarrow serviced = P$.

   This says that when a round starts, $serviced$ *must be empty. Also,when the round is more than* d_{phys} *old and up to* d *old, all processes must have been serviced.*

5. $\forall u \in U : \forall p \in P : \forall b \in Bool : \forall t = t_{slice}\lfloor RW.now/t_{slice}\rfloor : cand(u) = \langle p, b\rangle \Rightarrow u \in \{RW.reg^{-}(p,t), RW.reg^{+}(p,t)\}$.

   This says that if $cand(u)$ *is set to a pair containing some process, then that process was in region* u *at the start of the current round.*

6. $\forall p \in P : \neg failed_p \wedge clock_p \neq \bot$*:*

   (a) $reg_p = RW.reg(p) \neq \bot \wedge clock_p = RW.now$.

   This says that an alive process with $clock_p \neq \bot$ *has a* reg *variable and time corresponding to its region in* RW *and the time at* RW.

(b) $pref_p \Rightarrow (participated_p \vee clock \mod t_{slice} = 0)$.

This says that an alive process with $clock_p \neq \bot$ and $pref_p$ set to true either has a $participated$ variable set to true, or the round has just started.

7. $\forall p \in P : \neg failed_p \wedge clock_p \neq \bot \wedge participated_p$:

(a) $clock_p \mod t_{slice} \leq d$.

This says if there is an alive process with $clock_p \neq \bot$ and $participated_p$, then the round is at most d old.

(b) $\forall t \geq t_{slice} \lfloor clock_p / t_{slice} \rfloor : RW.reg^+(p, t) = reg_p$.

This says that an alive process with $clock_p \neq \bot$ and $participated_p$ has been in its current region since the time at the start of the current round.

(c) $cand(reg(p)) \neq \bot \wedge (pref_p \Rightarrow \exists q \in P : cand(reg(p)) = \langle q, true \rangle)$.

This says that if there is an alive process with $clock_p \neq \bot$ and $participated_p$, then its current region has a candidate for leader. If $pref_p$ is true in addition, then the process's current region has a candidate for leader that is tagged with "true" value.

(d) $pref(p) = pref_p$.

This says that if a process is alive and has $clock_p \neq \bot$ and $participated_p$ is true, then its local $pref_p$ value is the same preference value as that recorded in $LeadMain$.

(e) $p \notin serviced \vee (RW.now \mod t_{slice} > 0 \wedge cand(reg(p)) = \langle p, pref_p \rangle)$.

This says that an alive process with $clock_p \neq \bot$ and $participated_p$ equal to true is either not already serviced in $LeadMain$ or the round is older than 0 and the process's current region has a candidate leader pair that is equal to p paired with $pref_p$.

We now show that the set of properties describing $Inv_{LeadSpec}$ is a legal set for $LeadSpec$. (Together with the fact that the initial state of the system is in $Inv_{LeadSpec}$, this means that $Inv_{LeadSpec}$ is a set of invariant states.)

Lemma 10.3 *$Inv_{LeadSpec}$ is a legal set for $LeadSpec$.*

Proof: Let x be any state in $Inv_{LeadSpec}$. By Definition 3.12 of a legal set, we must verify two things for state x:

- For each state x' of $LeadSpec$ and action a of $LeadSpec$ such that (x, a, x') is in the set of discrete transitions of $LeadSpec$, state x' is in $Inv_{LeadSpec}$.

- For each state x' and closed trajectory τ of $LeadSpec$ such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in $Inv_{LeadSpec}$.

We previously showed that property 1 always holds. This leaves the remaining properties to verify.

For the first cast of the legal set definition, we check that if the properties hold in some state x and some action is performed that leads to state x', then the properties hold in state x'. We break this down by action:

- fail_p, $\mathsf{restart}_p$, reset, prefer_p, leader_p: The properties are trivial to verify with these actions.

- $\mathsf{GPSupdate}(l, t)_p$: The only relevant properties are 2, 6, and 7. The properties are trivial to check.

- $\mathsf{prefer}'(val)_p$: The only relevant properties are 5 and 7.

  For property 5, consider if $cand(reg_p)$ is updated as a result of the action. If not, then property 5 still holds since it did in state x. If $cand(reg_p)$ is updated, then it is updated to $\langle p, val\rangle$. Since property 7(b) held in state x, then property 5 holds in x'.

  For property 7, we know that $x'(participated_p)$ is true. Also, properties 7(a), 7(b), 7(d), and 7(e) obviously still hold. For the first part of 7(c), we need to check that $cand(reg(p)) \neq \bot$. If prefer' did not update $cand(reg(p))$, it must have been that $cand(reg(p))$ was not equal to $\bot$. If it did update $cand(reg(p))$, it updated it to $\langle p, val\rangle$. Either way, the first part of 7(c) holds. For the second part of 7(c), we need to check that if $x'(pref_p)$ is true then $cand(reg(p))$ is set to some tuple with a true boolean. If prefer' did not update $cand(reg(p))$, it must have been that $cand(reg(p))$ was already set to a tuple with a true tag since $val = pref_p$ and the post-state only

fails to adopt a true tag if it already has one. If prefer' did update $cand(reg(p))$ then it was updated to $\langle p, true\rangle$, satisfying the property.

- $\mathsf{leader}'(val)_p$: The only interesting property to check is 7(e). Consider the two cases for val. If val is false, then $x'(participated_p)$ does not hold, and we are done. If val is true and $x(participated_p)$ is false, then $x'(participated_p)$ is also false, and we are again done. If val is true and $x(participated_p)$ is true, then $x'(participated_p)$ is also true and p is in $serviced$, so we have to verify that $RW.now \mod t_{slice} > 0$ and $cand(reg(p)) = \langle p, pref_p\rangle$. That $RW.now \mod t_{slice} > 0$ is easy to see by virtue of the precondition for the action. To see that $cand(reg(p)) = \langle p, pref_p\rangle$, notice that the precondition for the action implies that $x(cand(reg(p))) = \langle p, x(pref(p))\rangle$. Since $cand$ and $pref(p)$ are not updated by the action, we have that $cand(reg(p)) = \langle p, x(pref(p))\rangle$. Since state x satisfies property 7(d), we know that $x(pref(p)) = x(pref_p)$. Since $pref_p$ is not changed when val is true and $participated_p$ is true, we have our result.

For the second case of the legal set definition, we check that for any closed trajectory τ starting with a state x where the properties hold and ending in a state x', the properties hold in state x'. The most interesting properties to check for this are 3, 6(b), and 7(a). Property 3 is preserved by the stopping conditions on line 19 of $LeadMain$, forcing a reset action to occur by $d_{phys} + 2\epsilon$ into a round. Property 4 is preserved by the stopping conditions on line 18 of $LeadMain$, forcing a leader' output to occcur for any unserviced processes. Property 6(b) is preserved by the stopping conditions on line 19 of $LeadCl$, forcing a prefer' output to occur to update $participated$. Property 7(a) is preserved by the stopping conditions on line 20 of $LeadCl$, forcing a leader output to occur to update $participated$. ■

Properties of *LeadSpec*

In each execution α of *LeadSpec* such that $\alpha.fstate \in Inv_{LeadSpec}$, we can show that the following properties hold for each region $u \in U$:

For each state x in α and process id $j \in P$, we define $aware(u, j, x)$ to be true exactly when $\neg x(failed_j), x(clock_p) \neq \bot$, and $x(reg(j)) = u$. (This is a way of saying that

process j is alive and knows it is in region u in state x.) Then for each $t \in \mathbb{R}^{\geq 0}$:

1. Say that $t \mod t_{slice} = 0$, $\alpha.fstate(RW.now) < t \leq \alpha.lstate(RW.now)$, and there exists some $p \in P$ and state x in α where $x(RW.now) = t$ and $aware(u, p, x)$ is true. Then there exists some $q \in P$ and state x' in α where $x'(RW.now) = t$, $aware(u, q, x')$ is true, and there exists a state x^* after x' where either (a) $x^*(RW.now) = t+d$ and there exists a leader_q at state x^* or (b) $x^*(RW.now) \leq t+d$ and $aware(u, q, x^*)$ is not true.

   In other words, if there are processes in region u at the start of the timeslice and none of those processes fail or leave the region until after the round is d old, then a leader_p output occurs when the round is d old at one of those processes.

2. For each $p \in P$, if a leader_p event occurs in α at state x where $x(RW.now) = t$ and $x(reg(u)) = u$ then:

   (a) $t \mod t_{slice} = d$.

   This says that leader outputs can only occur when a round is exactly d old.

   (b) If $\alpha.fstate(RW.now) \leq t-d$, then there exists a state x' where $x'(RW.now) = t - d$ and for all states x'' in α from x' until x, $aware(u, p, x'')$ is true.

   This says that if a leader_p occurs then it must be that process p was aware that it was in region u from the beginning of the round until the leader output.

   (c) If $\alpha.fstate(RW.now) < t - d$ and there exists a process q and prefer_q at time $t - d$ where $aware(u, q, x')$ is true for all states x' from the prefer_q until some state where $RW.now > t - d$, then there exists some prefer_p at a state x'' where $x''(RW.now) = t - d$ and $aware(u, p, x^*)$ is true for all x^* from x'' until x.

   This says that a leader output will not occur at a process that did not experience a prefer input at the beginning of the round unless no other process in its region experienced a prefer at the beginning of the round and remained aware it was in the region for some non-0 time. In other words, if there exists a higher priority process that remains aware it is in the region past the very beginning of a round, then no lower priority process will become leader of the region in that round.

Verification of these properties is relatively trivial under the assumption that α starts in a state in $Inv_{LeadSpec}$ (guaranteeing that appropriate regions and clock times are present in all components and that rounds begin fresh). For property 1, any process that is aware it is in some region at the start of a round will participate in the leader competition for that round unless it fails or moves before getting a chance to do so. If no such process fails or moves from the region until the round is more than d old, then *LeadMain*'s $cand$ for the region will be set to a pair consisting of one of those process ids, together with its submitted Boolean. The *LeadCl* for this process will then not reset its $participated$ variable until the round is exactly d old, when it performs a leader output.

Property 2(a) holds because of the precondition on line 49 of *LeadCl*. Property 2(c) holds because priority nominations are preferred by *LeadMain*, and high priority nominees that manage to fail, restart and be renominated with low priority will not receive a leader$'(true)$ input.

Property 2(b) is the most interesting to show. It holds because GPSupdates that indicate a region change or restarts after a process has failed both reset $participated$ to false at a *LeadCl*. If $participated$ is set to false after the round is more than 0 old, then it does not get reset to true again in the round, preventing a leader output at the process from being enabled if a region change or restart happens when a round is more than 0 old. This means that the only situation we need to examine is the one where a process is nominated for more than one region.We need to verify that in this case, the process will not perform a leader because it won the competition in the old region. The key observation here is that for this case to occur, a GPSupdate must have occurred that changed the process's region. If the process does not revert to the old region, then *LeadMain* will perform a leader$'(false)$ for the process, preventing a leader output. If the process does revert to the old region, it must be when the round is more than 0 old (since at most one GPSupdate per process is permitted per real-time value), implying that $participated$ is false, as described in the discussion of property 1.

```
Signature:
2   Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^≥0
    Input prefer_p
4   Input brcv(m)_p, m ∈ {candidate} × Bool × P × U
    Output bcast(m)_p, m ∈ {candidate} × Bool × P × U
6   Output leader_p

8 State:
    analog clock: ℝ^≥0 ∪ {⊥}, current real time, initially ⊥
10  reg: U ∪ {⊥}, current region, initially ⊥
    pref, participated: Bool, initially false
12
  Trajectories:
14  evolve
      if clock ≠ ⊥ then
16      d(clock) = 1
      else constant clock
18  stop when
      (clock mod t_slice = 0 ∧ ¬ participated ∧ reg ≠ ⊥)
20    ∨ (clock mod t_slice = d_phys + ε ∧ participated)

22 Transitions:
    Input GPSupdate(l, t)_p
24  Effect:
      if reg ≠ region(l) ∨ clock ≠ t ∨ (t mod t_slice > d_phys + ε ∧ participated)
26          ∨ (t mod t_slice > 0 ∧ pref ∧ ¬ participated) then
        clock ← t
28      reg ← region(l)
        pref, participated ← false

Input prefer_p
Effect:                                                   32
  if clock mod t_slice = 0 then
    pref ← true                                           34
    participated ← false
                                                          36
Output bcast(⟨candidate, val, p, u⟩)_p
Precondition:                                             38
  clock mod t_slice = 0 ∧ ¬ participated
  val = pref ∧ u = reg ≠ ⊥                                40
Effect:
  participated ← true                                     42

Input brcv(⟨candidate, val, q, v⟩)_p                      44
Effect:
  if v = reg ∧ clock mod t_slice ∈ (0, d_phys] then        46
    if (val ∧ ¬ pref) ∨ (val = pref ∧ q < p) then
      pref, participated ← false                          48

Output leader_p                                           50
Precondition:
  clock mod t_slice = d_phys + ε ∧ participated           52
Effect:
  pref, participated ← false                              54
```

Figure 10-4: Leader$_p$, electing a leader.

10.2 *LeadImpl*: Implementation

Here we describe our implementation of *LeadSpec* (Figure 10-4). *LeadSpec* is implemented by $Leader_p$ automata with access to *RW*$\|$*Pbcast*. At the beginning of each round, a process tosses its hat into the ring as a possible leader for its region by broadcasting a candidate message, together with its id and priority. Each process then collects these messages until d time into the round. Whenever such a message for its region is received, if the process is still participating then it compares the id and Boolean priority to its own local id and priority. If the message's priority does not have priority over the process's local priority and the message's process id is not lower, then the process does nothing. Otherwise, the process ceases participating and readies itself for the next round.

The interface of $Leader_p$ consists of the following five kinds of actions:

- **Input** GPSupdate$(l, t)_p, l \in R, t \in \mathbb{R}^{\geq 0}, p \in P$: This input indicates that a process p is currently located at position l.

- **Input** $\mathsf{prefer}_p, p \in P$: This input indicates that the process is to have priority in leader election. (This is the prefer input for $LeadCl_p$.)

- **Input** $\mathsf{brcv}(\langle \mathsf{candidate}, val, q, v\rangle)_p, val \in Bool, v \in U, q, p \in P$: This is the receipt of a candidate message from some process.

- **Output** $\mathsf{bcast}(\langle \mathsf{candidate}, val, p, u\rangle)_p, val \in Bool, u \in U, p \in P$: This output is the process putting itself up for consideration as leader for its region $u = reg_p$. The value val is true if $pref$ is true, indicating a prefer has occurred in this round at the process.

- **Output** $\mathsf{leader}_p, p \in P$: This output communicates that a process is the leader for its current region.

Its state variables are the following:

- $clock : \mathbb{R}^{\geq 0} \cup \{\perp\}$: This is the process's local clock. It is initially $\perp$, but is set to the system's real-time when a GPSupdate occurs at the process.

- $reg : U \cup \{\perp\}$: This is the last reported region of the process since initialization.

- $pref : Bool$: This value indicates priority of the process. If a prefer occurs at the beginning of a round, this value is set to true and triggers a bcast output. Otherwise, this value is false.

- $participated : Bool$: This indicates whether the process has or needs to participated in its current region's leader election via a bcast action.

At the start of a round, $Leader_p$ performs a $\mathsf{bcast}(\langle \mathsf{candidate}, pref, p, reg\rangle)_p$ output, setting $participated$ to true so as to prevent additional such outputs (lines 37-42). It may also receive a prefer_p input (indicating that its client wants to have priority in the leader election), resulting in the setting of $pref$ to true and $participated$ to false, triggering a(nother) bcast output (lines 31-35). Whenever a GPSupdate occurs at the process that changes its region or clock, $pref$ and $participated$ are set to false, preventing the process from later performing a leader output in the region it left (line 29).

Later, if it receives a $\mathsf{brcv}(\langle \mathsf{candidate}, val, q, reg \rangle)_p$ input (line 44), then if it is no later than d_{phys} into the round and either val is true while $pref$ is false (meaning the sender had a higher priority) or val and $pref$ are the same but $q < p$ (lines 46-47), then $Leader_p$ sets $pref$ and $participated$ to false, initializing those values for the next round (lines 48). Otherwise, it does nothing.

If, at exactly d into the round, $participated$ is still true (meaning that it did not receive a candidate message for its region from a higher priority or same priority but lower id process) then $Leader_p$ performs a leader_p output (lines 50-52), and initializes $pref$ and $participated$ for the next round (line 54).

10.3 Correctness of the implementation

In this section we describe aspects of the correctness of our implementation of the leader election service. We define the complete implementation system to be the composition of the *Fail*-transformed $Leader$ automata together with $Pbcast$ and RW, with certain actions hidden:

Definition 10.4 *Let* $H_{Leadimpl}$ *be* $\{\mathsf{bcast}(m)_p, \mathsf{brcv}(m)_p \mid p \in P, m \in \{\mathsf{candidate}\} \times Bool \times P \times U\}$*, and let* LeadImpler *be* $\prod_{p \in P} Fail(Leader_p)$*. Then define* $LeadImpl$ *to be* $\mathsf{ActHide}(H_{Leadimpl}, LeadImpler \| Pbcast \| RW)$.

To show correctness, we use the strategy described in Section 9.3:

1. Describe a legal set L_{Leader} of *LeadImpl*, and show that it is a legal set (Definition 10.8).

2. Define a legal set $Inv_{LeadSpec}$ for the specification $LeadSpec$, and show that the set is a legal set. (This was done in Section 10.1.3.)

3. Show that $Start(LeadImpl, L_{Leader}) \leq Start(LeadSpec, Inv_{LeadSpec})$ (Lemma 10.14). We show this in the following way:

   (a) Define a simulation relation $\mathcal{R}_{Leader}$ between *LeadImpl* and *LeadSpec* (see Definition 10.9). Show the relation is a simulation relation (Lemma 10.10).

(b) Show that for each state in L_{Leader}, there exists a state in the invariant set $Inv_{LeadSpec}$ such that $\mathcal{R}_{Leader}$ holds between the states (Lemma 10.13).

4. Show that *LeadImpler* is self-stabilizing to L_{Leader} relative to $R(RW \| Pbcast)$ (Theorem 10.17).

5. Conclude that the set of traces of the implementation stabilizes to the set of traces of executions of *LeadSpec* starting in $Inv_{LeadSpec}$ (Theorem 10.18).

10.3.1 Legal sets

Here we describe a legal set of *LeadImpl* by describing two legal sets, one a subset of the other. Recall from Lemma 3.13 that a legal set of states for a TIOA is one where each closed execution fragment starting in a state in the set ends in a state in the set. We break the definition of the legal set up into two sets in order to simplify the proof reasoning and more easily prove stabilization later.

The first set of legal states describes some properties that become true at an alive process at the time of the first GPSupdate for the process.

Definition 10.5 *Define L'_{Leader} to be the set of states x of $LeadImpl$ such that each of the following properties hold:*

1. $x\lceil X_{Pbcast\|RW} \in Inv_{Pbcast\|RW}$.
This says that the state restricted to $Pbcast\|RW$ is a reachable state of the $Pbcast\|RW$.

2. $\forall p \in P : \neg failed_p \wedge clock_p \neq \bot$:

(a) $reg_p = RW.reg(p) \neq \bot \wedge clock_p = RW.now$.
This says that alive processes with $clock_p \neq \bot$ have a local clock variable and region setting that matches the clock and region setting in RW.

(b) $participated_p \Rightarrow clock_p \mod t_{slice} \leq d$.
This says that if there is an alive processes with $clock_p \neq \bot$ and $participated_p$, then the round is at most d old.

(c) $pref_p \Rightarrow (participated_p \vee clock_p \mod t_{slice} = 0)$.

This says that if there is an alive processes with $clock_p \neq \bot$ and $pref_p$ then either $participated_p$ is true or the round has just started.

It is easy to check that L'_{Leader} is a legal set for $LeadImpl$.

Lemma 10.6 *L'_{Leader} is a legal set for $LeadImpl$.*

Next we define a set of "reset" states for the algorithm. The reset states correspond to states of *LeadImpl* after the leader election competition for one round has completed and before the competition for the next round begins (when the leader competition state is "reset"). It also turns out that it is relatively simple to show that an execution fragment of *LeadImpl* reaches a reset state. When we define our final set of legal states in Definition 10.8 as states reachable from reset states, it makes the task of showing stabilization of *LeadImpl* in Section 10.3.3 much simpler.

Definition 10.7 *Define $Reset_{Leader}$ to be the set of states x of $LeadImpl$ such that each of the following properties hold:*

1. $x \in L'_{Leader}$.

   This says that x is a state in L'_{Leader}.

2. $RW.now \mod t_{slice} = 0 \vee RW.now > d_{phys} + \epsilon$.

   This says that x is either at the beginning of a round or more than $d_{phys} + \epsilon$ into one.

3. $\forall p \in P : (\neg failed_p \wedge clock_p \neq \bot) : \neg pref_p$.

   This says that each alive process with $clock_p \neq \bot$ has $pref_p$ set to false.

4. $\forall p \in P : \forall \langle \langle \mathsf{candidate}, b, q, u \rangle, t, P' \rangle \in Pbcast.pbcastq(p) : P' = \emptyset$.

   This says that there are no candidate *messages in transit.*

The reset states are used to define our final set of legal states L_{Leader} for $LeadImpl$. L_{Leader} is the set of states reachable from a reset state.

Definition 10.8 *Define L_{Leader} to be $reachable_{Start(LeadImpl, Reset_{Leader})}$.*

It is obvious that L_{Leader} is a legal set for $LeadImpl$.

10.3.2 Simulation relation

Here we show that $Start(LeadImpl, L_{Leader})$ implements $Start(LeadSpec, Inv_{LeadSpec})$ (Lemma 10.14). We do this by first describing a simulation relation $\mathcal{R}_{Leader}$ from our implementation of the leader election service, *LeadImpl*, to the TIOA specification of the leader election service, *LeadSpec* (Definition 10.9). We prove that $\mathcal{R}_{Leader}$ is a simulation relation, and then conclude that *LeadImpl* implements *LeadSpec*. In other words, we conclude that the traces of our implementation are traces of leader election. We then show that for each state in L_{Leader}, there exists a state in the invariant set $Inv_{LeadSpec}$ such that $\mathcal{R}_{Leader}$ holds between the states (Lemma 10.13).

You may notice in the definition below that for $x\mathcal{R}_{Leader}y$ to hold, state x must be a state in the legal set L_{Leader}. This constrains the simulation relation to only be concerned with implementation states which we will then show are related to states of $LeadSpec$ in $Inv_{LeadSpec}$.

Now we define the simulation relation for our algorithm.

Definition 10.9 *$\mathcal{R}_{Leader}$ is a relation between states of $LeadImpl$ and $LeadSpec$ such for any states x and y of the two machines respectively, $x\mathcal{R}_{Leader}y$ exactly when the following conditions are satisfied:*

1. *State x satisfies the following:*

   (a) $x \in L_{Leader}$.

   This says that x is a state in L_{Leader}.

   (b) $\forall p \in P : \forall \langle\langle \mathsf{candidate}, b, q, u\rangle, t, P'\rangle \in Pbcast.pbcastq(p)$:

   i. $P' = \emptyset \vee t = t_{slice}\lfloor RW.now/t_{slice}\rfloor$.

   This says that candidate *messages submitted to $Pbcast$ have either been processed for each process or were sent at the beginning of the current round.*

   ii. $(RW.now \mod t_{slice} \leq d_{phys} + \epsilon \wedge t = t_{slice}\lfloor RW.now/t_{slice}\rfloor) \Rightarrow (q = p \wedge u \in \{RW.reg^{-}(p,t), RW.reg^{+}(p,t)\})$.

This says that if the current round is not more than d old, then any candidate *messages sent at the beginning of the round are tagged with the correct source and region for the process that sent it.*

(c) $\forall p \in P : \neg failed_p \wedge clock_p \neq \perp \wedge participated_p$:

i. $\exists P' \subseteq P : \langle\langle \mathsf{candidate}, pref_p, p, reg_p\rangle, t_{slice}\lfloor RW.now/t_{slice}\rfloor, P'\rangle \in Pbcast.pbcastq(p)$.

This says that each nonfailed process with non-$\perp$ $clock$ and $participated$ set to true sent a candidate *message at the beginning of the round for its current region and $pref$ variable.*

ii. $\forall q \in P : \forall\langle\langle \mathsf{candidate}, b, q, reg_p\rangle, t_{slice}\lfloor RW.now/t_{slice}\rfloor, P'\rangle \in Pbcast.pbcastq(q) : (p \in P' \vee [pref_p = b \wedge p \leq q] \vee [pref_p \wedge \neg b])$.

This says that for each nonfailed process p with non-$\perp$ $clock$ and $participated$ set to true and for each candidate *message sent for the process's region at the beginning of the current timeslice and processed for p, either the* candidate *message's Boolean was false and $pref_p$ is true or $pref_p$ is equal to the message's Boolean and id p is ordered before q.*

2. *State y satisfies the following:*

(a) $RW.now = LeadMain.now \wedge RW.reg = LeadMain.reg$.

This says that LeadMain*'s clock is the real-time, and that its stores regions for processes is consistent with RW's.*

(b) $\forall p \in P : (\neg failed_p \wedge clock_p \neq \perp \wedge participated_p) \Rightarrow (pref(p) = pref_p \wedge [p \notin serviced \vee (RW.now \mod t_{slice} > 0 \wedge cand(reg(p)) = \langle p, pref_p\rangle)])$.

This says that for each nonfailed process with non-$\perp$ $clock$ and $participated$ set must have $pref(p)$ match $pref_p$ and either p has not yet received a leader′ *input or the round is more than 0 old and the process is the leader of its current region.*

(c) $RW.now \mod t_{slice} > d_{phys} + \epsilon \Rightarrow (serviced = \emptyset \wedge \forall u \in U : cand(u) = \perp \wedge \forall p \in P : \neg pref(p))$.

This says that when a round is more than d old, $serviced$ must be empty, $cand(u)$ must be initialized for each region, and $pref(p)$ must be false for each $p \in P$.

(d) $RW.now \mod t_{slice} = 0 \Rightarrow serviced = \emptyset$.

This says that at the beginning of a round, $serviced$ must be empty.

3. $x(RW) = y(RW)$.

   This says that RW matches in both states.

4. $\forall p \in P : x(Fail(Leader_p)) = y(Fail(LeadCl_p))$.

   This says that each process's failure status is the same in x and y.

5. *Let $leadCand : (U \times Bool) \rightarrow P \cup \{\bot\}$ be a function that takes a region u and Boolean b, and returns the lowest id p such that $\exists P' \subseteq P : \langle\langle \mathsf{candidate}, b, p, u\rangle, t_{slice}\lfloor x(RW.now)/t_{slice}\rfloor, P'\rangle \in x(Pbcast.pbcastq(p))$, or $\bot$ if no such p exists.*

   Let $leader : U \rightarrow P \cup \{\bot\}$ be a function that takes a region u and returns $\langle p, true\rangle$ if $leadCand(u, true) = p \neq \bot$, $\langle p, false\rangle$ if $leadCand(u, true) = \bot$ and $leadCand(u, false) = p \neq \bot$, or $\bot$ if $leadCand(u, true) = leadCand(u, false) = \bot$.

   Then $\forall u \in U : y(RW.now) \mod t_{slice} \leq d_{phys} + \epsilon \Rightarrow y(cand(u)) = leader(u)$.

   This says that $cand(u)$ in state y is set to the process, if it exists, with the lowest id amongst the process tags for candidate *messages with the same Boolean value for the region sent at the beginning of the current round, and for which the Boolean was either true or there were no such true Boolean-tagged messages in the round.*

Now we show that $\mathcal{R}_{Leader}$ is a simulation relation from *LeadImpl* to *LeadSpec*.

Lemma 10.10 *$\mathcal{R}_{Leader}$ is a simulation relation between states of $LeadImpl$ and $LeadSpec$.*

Proof: By definition of a simulation relation (Definition 2.20) we must show three things for all states of the automaton:

1. We must show that for any $x \in \Theta_{LeadImpl}$ there exists a state $y \in \Theta_{LeadSpec}$ such that $x\mathcal{R}_{Leader}y$.

   The corresponding state y of the specification is the one with the same RW as x, with $x(Fail(Leader_p)) = y(Fail(LeadCl_p))$ for all $p \in P$, and with the variables of $LeadMain$ set to their unique initial values. It is easy to check that $x\mathcal{R}_{Leader}y$.

2. Say that x and y are states such that $x\mathcal{R}_{Leader}y$. Then for any action $a \in A_{LeadImpl}$, if $LeadImpl$ performs action a and the state changes from x to x', we must show there exists a closed execution fragment β of $LeadSpec$ with $\beta.fstate = y, trace(\beta) = trace(\wp(x)a\wp(x'))$, and $x'\mathcal{R}_{Leader}\beta.lstate$. For this proof we should consider each action a.

   - fail_p, $\mathsf{restart}_p$, $\mathsf{GPSupdate}(l,t)_p$, prefer_p: The corresponding execution fragment is $\wp(y)a\wp(y')$. The traces of α and β are the same, and checking that the relation holds between x' and y' is trivial.

   - $\mathsf{brcv}(\langle \mathsf{candidate}, b, q, u\rangle)_p$: The corresponding execution fragment is either: $\wp(y)$ or $\wp(y)\mathsf{leader}'(false)_p\wp(y')$. It is obvious that in each of these cases, the traces of α and β are both empty.

     We select the corresponding execution fragment in the following way: If p is alive and $clock_p \neq \perp$, $u = x(reg_p)$, $x(participated_p)$ and $[(b \wedge \neg x(pref_p)) \vee (b = x(pref_p) \wedge q < p)]$ then the fragment is $\wp(y)\mathsf{leader}'(false)_p\wp(y')$. To see that the $\mathsf{leader}'(false)_p$ action is enabled, we need to check that $p \notin x(serviced)$ and that $x(cand(reg(p))) \neq \langle p, x(pref(p))\rangle$. To see that x' and y' are related, since state x satisfies property 2(b) of the simulation relation, we know that either $p \notin x(serviced)$ or $x(cand(reg(p))) = \langle p, x(pref_p)\rangle$. Since x also satisfies property 5 of the relation, we know that $p \notin x(serviced)$ and that $x(cand(reg(p))) \neq \langle p, x(pref(p))\rangle$.

     Otherwise the fragment is $\wp(y)$. To see that $x'$and y' are related, the only properties we need to recheck are properties 1(c)(ii) and 4. These are easy to check.

   - $\mathsf{bcast}(\langle \mathsf{candidate}, b, p, u\rangle)_p$: The corresponding execution fragment is $\wp(y)\mathsf{prefer}'(pref_p)_p\wp(y')$, where state $y'(LeadMain.cand(reg(p)))$ is selected

in the following way: If $cand(reg(p)) = \perp \vee \exists \langle q, b' \rangle = cand(reg(p)) : [(b \wedge \neg b') \vee (b = b' \wedge p < q)]$, then update $cand(reg(p))$ to be $\langle p, b \rangle$. Otherwise, leave $cand(reg(p))$ the same.

The traces of α and β are both empty. To see that the prefer′ action is enabled, note that the two actions basically have the same precondition. To see that x' and y' are related, it is trivial to check that properties 1-4 of the simulation relation hold. For property 5, notice that if $y(cand(reg(p))) = \perp$, then $y'(cand(reg(p))) = \langle p, b \rangle$. This obviously satisfies property 5. If $y(cand(reg(p))) \neq \perp$ then since property 5 holds in state x, it must be that there is some $\langle q, b' \rangle = y(cand(reg(p)))$ such that there is an associated candidate message in $pbcastq$ in x for the current round and $reg(p)$. Since we update $y'(cand(reg(p)))$ exactly when $\langle p, b \rangle$ is such that b is true and b' is not, or $p < q$ and $b = b'$, then we know that $leader(reg(p))$ in state x' is equal to $cand(reg(p))$ in state y'.

- leader_p: The corresponding execution fragment is $\wp(y)a\wp(y')$. The traces of α and β are the same, and checking that the action is enabled and that the relation holds between x' and y' is trivial.

- $\mathsf{drop}(\langle \mathsf{candidate}, b, p', u \rangle, t, q, p)$: The corresponding execution fragment is $\wp(y)$. The traces of α and β are both empty. To see that x' and y' are related, since property 1 holds in state x, we know that the message was for a different region than q's, meaning it has no bearing on the properties covered by the simulation relation.

3. Say that $x \mathcal{R}_{Leader} y$. Let α be an execution fragment of $LeadImpl$ consisting of one closed trajectory, with $\alpha.fstate = x$. We must show that there is a closed execution fragment β of $LeadSpec$ with $\beta.fstate = y, trace(\beta) = trace(\alpha)$, and $\alpha.lstate \mathcal{R}_{Leader} \beta.lstate$.

   Let t_0 be $\alpha.fstate(RW.now)$ and t_3 be $\alpha.lstate(RW.now)$. Let t_1 be $d_{phys} + t_{slice}\lfloor t_0/t_{slice} \rfloor$. If $t_3 \mod t_{slice} > d_{phys} + \epsilon$ then let t_2 be $min(t_3, d_{phys} + 2\epsilon + t_{slice}\lfloor t_0/t_{slice} \rfloor)$, else let t_2 be $d_{phys} + 2\epsilon + t_{slice}\lfloor t_0/t_{slice} \rfloor$.

Let $p_1, \cdots, p_m$ be an ordering of the set of $p \in P$ such that $p \notin y(serviced)$ and $\langle p, y(pref(p))\rangle \neq y(cand(reg(p)))$. Let $p_{m+1}, \cdots, p_n$ be an ordering of the set of $p \in P$ such that $p \notin y(serviced)$ and $\langle p, y(pref(p))\rangle = y(cand(reg(p)))$.

If $t_1 \in [t_0, t_3)$ and $t_2 \in (t_0, t_3]$ then β is the execution fragment $\tau_1 \mathsf{leader}'(false)_{p_1} \tau_{1,1} \mathsf{leader}'(false)_{p_2} \tau_{1,2}, \cdots \mathsf{leader}'(false)_{p_m} \tau_{1,m}, \mathsf{leader}'(true)_{p_{m+1}} \tau_{1,m+1}, \cdots, \mathsf{leader}'(true)_{p_n} \tau_{1,n}, \tau_2, \mathsf{reset}\tau_3$, where $\beta.ltime = \alpha.ltime$ and $\tau_i.lstate(RW.now) = t_i$.

If $t_1 \in [t_0, t_3)$ but $t_2 \notin (t_0, t_3]$ then β is the same as above, except that it ends with τ_2 and $\tau_2.lstate = t_3$.

If $t_1 \notin [t_0, t_3)$ and $t_2 \in (t_0, t_3]$, then β is $\tau_2 \mathsf{reset} \tau_3$.

If $t_1 \notin [t_0, t_3)$ and $t_2 \notin (t_0, t_3]$ then β is just τ_3.

In other words, we fill in leader′ actions for processes that have not been serviced when the time is d_{phys} after the start of the round after other actions have been completed, so as to not violate the trajectory stopping conditions on line 18 of $LeadMain$. We also fill in reset actions at times $d_{phys} + 2\epsilon$ into a round, or at time t_3 if t_3 is before $d_{phys} + 2\epsilon$ and after d_{phys} into a round, so as to not violate the trajectory stopping conditions on line 19 of $LeadMain$ and to satisfy property 2(c) of the simulation relation.

It is easy to check that $\alpha.lstate \mathcal{R}_{Leader} \beta.lstate$.

■

The following theorem concludes that our implementation of the leader election service implements *LeadSpec*.

Theorem 10.11 $LeadImpl \leq LeadSpec$.

Proof: This follows directly from the previous lemma and Corollary 2.23. ■

One useful observation about the proof that $\mathcal{R}_{Leader}$ is a simulation relation is the following, which says that for any execution fragment of *LeadImpl* starting in a state x in

L_{Leader} and for any state y in $Inv_{LeadSpec}$ such that $x\mathcal{R}_{Leader}y$, there is some fragment of the leader election specification starting in state y that not only has the same trace but also has the same RW and $Fail$-related projections. (This is very useful later, when reasoning about the $Fail$-transformed composition of the leader election implementation pieces with pieces of other services):

Lemma 10.12 *Let α be in $frags_{LeadImpl}^{L_{Leader}}$ and y be a state in $Inv_{Leadspec}$ such that $\alpha.fstate\mathcal{R}_{Leader}y$. Then there exists an α' in $frags_{LeadSpec}^{Inv_{LeadSpec}}$ such that:*

1. *$\alpha'.fstate = y$.*

2. *$trace(\alpha) = trace(\alpha')$.*

3. *If α is a closed exectuion fragment, then $\alpha.lstate\mathcal{R}_{Leader}\alpha'.lstate$.*

4. *$\alpha\lceil(A_{RW}, V_{RW}) = \alpha'\lceil(A_{RW}, V_{RW})$.*

5. *For each $p \in P$, $\alpha\lceil(\{\mathsf{fail}_p, \mathsf{restart}_p\}, \{failed_p\}) = \alpha'\lceil(\{\mathsf{fail}_p, \mathsf{restart}_p\}, \{failed_p\})$.*

The first three properties of the lemma follow from the fact that $\mathcal{R}_{Leader}$ is a simulation relation, while the last two properties follow from the construction of the matching execution of $LeadSpec$ in the proof that $\mathcal{R}_{Leader}$ is a simulation relation, which preserves the actions and variables of RW and each of the processes' $Fail$-transform variables and actions.

Now, to show that each state in L_{Leader} is related to a legal state of the specification, it is enough to show that each state in $Reset_{Leader}$ is related to a legal state of the specification.

Lemma 10.13 *For each state $x \in L_{Leader}$, there exists a state $y \in Inv_{LeadSpec}$ such that $x\mathcal{R}_{Leader}y$.*

Proof: Let x be a state in L_{Leader}. By definition of L_{Leader}, x is a state reachable from a state in $Reset_{Leader}$. Hence, we just need to show that for any state x in $Reset_{Leader}$, we can construct a state y based on state x such that $x\mathcal{R}_{Leader}y$ holds.

Let state $y(RW) = x(RW)$, $y(LeadMain.now) = x(RW.now)$, $y(LeadMain.reg) = x(RW.reg)$, $y(serviced) = \emptyset$, $\forall u \in U : y(cand(u)) = \bot$, $\forall p \in P : y(prob(p)) = false$, and $y(Fail(LeadCl_p)) = x(Fail(Leader_p))$.

It is trivial to verify both that state y satisfies the properties of $Inv_{LeadSpec}$ and that $x\mathcal{R}_{Leader}y$ holds. ■

We can now conclude that a trace of an execution of *LeadImpl* started in a state in L_{Leader} is the same as the trace of some execution fragment of *LeadSpec* starting in a legal state.

Lemma 10.14 $tracefrags_{LeadImpl}^{L_{Leader}} \subseteq tracefrags_{LeadSpec}^{Inv_{LeadSpec}}$.

Proof: This follows from Lemma 10.13 and Lemma 10.12. ■

10.3.3 Self-stabilization

We've seen that L_{Leader} (Definition 10.8) is a legal set for $LeadImpl$, and that each state in L_{Leader} is related to a state in $Inv_{LeadSpec}$ (Lemma 10.13). Here we show that ***LeadImpler*** self-stabilizes to L_{Leader} relative to $R(RW\|Pbcast)$ (Theorem 10.17), meaning that if certain program portions of the implementation are started in an arbitrary state and run with $R(RW\|Pbcast)$, the resulting execution eventually gets into a state in L_{Leader}. This is done in two phases, corresponding to each legal set L'_{Leader} and L_{Leader}.

After we show that ***LeadImpler*** self-stabilizes to L_{Leader} relative to $R(RW\|Pbcast)$, we use the fact that $\mathcal{R}_{Leader}$ (see Definition 10.9) is a simulation relation that relates states in L_{Leader} with states of $LeadSpec$ in $Inv_{LeadSpec}$ to conclude that after an execution of ***LeadImpl*** has stabilized, the trace fragment from the point of stabilization with bcast and brcv actions hidden is the suffix of some trace of ***LeadSpec*** starting in $Inv_{LeadSpec}$ (Theorem 10.18).

It is easy to check that $\prod_{p\in P} Fail(Leader_p)$ is self-stabilizing to L'_{Leader} in time t^1_{lead} relative to $R(Pbcast\|RW)$, where t^1_{lead} is any t such that $t > \epsilon_{sample}$. (To see this stabilization result, just consider the moment after each node has received a GPSupdate, which takes at most ϵ_{sample} time to happen.)

Lemma 10.15 *Let* t^1_{lead} *be any* t *such that* $t > \epsilon_{sample}$.
$\prod_{p\in P} Fail(Leader_p)$ *is self-stabilizing to* L'_{Leader} *in time* t^1_{lead} *relative to* $R(Pbcast\|RW)$.

We show that starting from a state in L'_{Leader}, *LeadImpl* ends up in a state in L_{Leader} within t^2_{lead} time, where t^2_{lead} is any t such that $t > 2d_{phys} + \epsilon$.

Lemma 10.16 *Let t^2_{lead} be any t such that $t > 2d_{phys} + \epsilon$. $frags^{L'_{Leader}}_{LeadImpl}$ stabilizes in time t^2_{lead} to $frags^{L_{Leader}}_{LeadImpl}$.*

Proof: We just need to show that for any length-t^2_{lead} prefix α of an element of $frags^{L'_{Leader}}_{LeadImpl}$, $\alpha.lstate \in L_{Leader}$. By the definition of L_{Leader}, we just need to show that there is at least one state in $Reset_{Leader}$ that occurs in α.

Let t_0 be equal to $\alpha.fstate(RW.now)$, the time of the first state in α. In $\alpha.fstate$, there may be messages in $Pbcast.pbcastq$ that can take up to d_{phys} time to be dropped or delivered at each process. We'll call any of these above messages "bad" messages. We know that all "bad" messages will be processed (dropped or delivered at each process) by some state x in α such that $x(RW.now) = t_1 = t_0 + d_{phys}$.

Code inspection tells us that for any state in L'_{Leader} and hence for any state in α, any new bcast transmissions of candidate messages will occur exactly when $RW.now \mod t_{slice} = 0$, and will be processed (dropped or delivered at each process) by d_{phys} later. Notice that in each of these cases, any bcast transmission is processed by d_{phys} into a round. This implies that any state after state x in α where $RW.now \mod t_{slice} > d_{phys}$ or $RW.now \mod t_{slice} = 0$ satisfies properties 1, 2, and 4 of $Reset_{Leader}$.

Notice that any state after x in α where $RW.now \mod t_{slice} > d_{phys} + \epsilon$ also satisfies property 3 of $Reset_{Leader}$. This means that to complete our proof we just need to bound the amount of time that could be required to get from state x to a state x^* such that $t_2 = x^*(RW.now) \mod t_{slice} > d_{phys} + \epsilon$ and $x^*(RW.now) > x(RW.now)$.

We consider three cases for time t_1. First, if $t_1 \mod t_{slice} > d_{phys} + \epsilon$, then for any t_2 such that $t_2 - t_1 \in (0, t_{slice} - (t_1 \mod t_{slice}))$, we're done. Second, if $t_1 \mod t_{slice} \leq d_{phys} + \epsilon$ but does not equal 0, then for $t_2 = t_1 + d_{phys} + \epsilon$, $t_2 \mod t_{slice} > d_{phys} + \epsilon$, and we're done. Last, if $t_1 \mod t_{slice} = 0$, then for any t_2 such that $t_2 - t_1 \in (d_{phys} + \epsilon, t_{slice})$, $t_2 \mod t_{slice} > d_{phys} + \epsilon$, and we're done.

This implies the total time for stabilization is any $t > 2d_{phys} + \epsilon$, which t^2_{lead} satisfies. ■

Now we can combine our stabilization results to conclude that $Fail(Leader_p)$ compo-

nents started in an arbitrary state and run with $R(Pbcast\|RW)$ stabilizes to L_{Leader} in time t_{lead}, where t_{lead} is any t such that $t > 2d_{phys} + \epsilon + \epsilon_{sample}$.

Theorem 10.17 *Let t_{lead} be any t such that $t > 2d_{phys} + \epsilon + \epsilon_{sample}$.*
$LeadImpler$ is self-stabilizing to L_{Leader} in time t_{lead} relative to $R(Pbcast\|RW)$.

Proof: We must show that $execs_{U(LeadImpler)\|R(RW\|Pbcast)}$ stabilizes in time t_{lead} to $frags^{L_{Leader}}_{LeadImpler\|R(RW\|Pbcast)}$. By Corollary 3.11, $frags^{L_{Leader}}_{LeadImpler\|R(RW\|Pbcast)}$ is the same as $frags^{L_{Leader}}_{LeadImpl}$. The result follows from application of transitivity of stabilization (Lemma 3.6), applied to the two lemmas above. Let $t^1_{lead} = \epsilon_{sample} + (t_{lead} - 2d_{phys} - \epsilon - \epsilon_{sample})/2$ and $t^2_{lead} = 2d_{phys} + \epsilon + (t_{lead} - 2d_{phys} - \epsilon - \epsilon_{sample})/2$.

First, let B be $execs_{U(LeadImpler)\|R(RW\|Pbcast)}$, C be $frags^{L'_{Leader}}_{LeadImpl}$, and D be $frags^{L_{Leader}}_{Leadimpl}$ in Lemma 3.6. Then by Corollary 3.11 and Lemmas 10.15 and 10.16, we have that $execs_{U(LeadImpler)\|R(RW\|Pbcast)}$ stabilizes in time $t^1_{lead} + t^2_{lead}$ to $frags^{L_{Leader}}_{LeadImpl}$.

Since $t_{lead} = t^1_{lead} + t^2_{lead}$, we conclude that $LeadImpler$ self-stabilizes in time t_{lead} to L_{Leader} relative to $R(RW\|Pbcast)$. ■

We can finally pull our results together to conclude that traces of $LeadImpl$ with $Fail(Leader_p)$ components started in an arbitrary state and run with $R(Pbcast\|RW)$ stabilize in time t_{lead} to traces of $LeadSpec$ starting from a state in $Inv_{LeadSpec}$.

Theorem 10.18 *Let t_{lead} be any t such that $t > 2d_{phys} + \epsilon + \epsilon_{sample}$.*
$traces_{U(LeadImpler)\|R(Pbcast\|RW)}$ stabilizes in time t_{lead} to $traces_{Start(LeadSpec, Inv_{LeadSpec})}$.

Proof: By Theorem 10.17 and the definition of self-stabilization, we have that $traces_{U(LeadImpler)\|R(Pbcast\|RW)}$ stabilizes in time t_{lead} to $tracefrags^{L_{Leader}}_{LeadImpl}$. Since we showed in Lemma 10.14 that $tracefrags^{L_{Leader}}_{LeadImpl} \subseteq traces_{Start(LeadSpec, Inv_{LeadSpec})}$, we have our result. ■

Chapter 11

Implementation of the VSA layer

Here we describe an implementation of the VSA layer (defined in Chapter 7) by the mobile nodes in a network. This implementation uses RW, the totally ordered broadcast service, and the leader election service.

We present the implementation as a trivial client implementation, together with a more involved VSA implementation. We then reason that this implementation describes a stabilizing VSA layer emulation algorithm.

11.1 Client implementation

Recall the VSA abstraction consists not just of VSAs and $Vbcast$, but also client automata, corresponding to mobile nodes in the network. The implementation of client automata is almost trivial; $CE[alg]_p$ is equal to $alg(p)$, except that the vcast and vrcv actions are replaced by tocasts and torcvs of message tuples. A vcast(m) becomes a tocast$(\langle \mathsf{vmsg}, false, m\rangle)$. A vrcv$(m)$ input becomes a torcv$(\langle \mathsf{vmsg}, b, m\rangle)$, $b \in Bool$, action. The effect on local state is the same for both actions.

11.2 VSA implementation

We describe a fault-tolerant implementation of a VSA by mobile nodes in its region of the network. At a high level, the individual mobile nodes in a region share emulation of

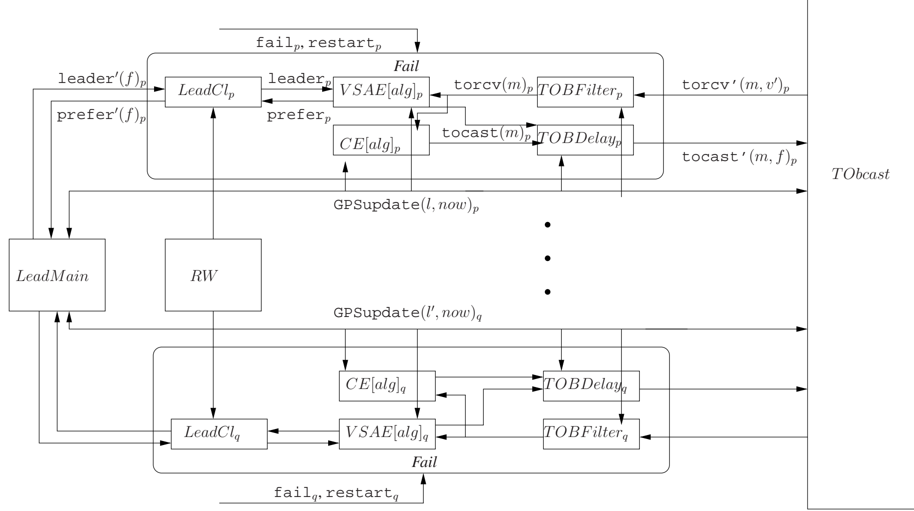


Figure 11-1: VSA layer implementation. Each process runs a collection of algorithms: $LeadCl, TOBDelay$, and $TOBFilter$, defined previously, together with $CE[alg]$ and $VSAE[alg]$, the client and VSA emulator algorithms.

the virtual machine through a deterministic state replication algorithm while also being coordinated by a leader. Each mobile node runs its portion of the totally ordered broadcast service, leader election service, and a Virtual Node Emulation ($VSAE$) algorithm, for each virtual node. The TIOA implementation for $VSAE_p$ is in Figure 11-2.

For each $alg \in VALgs$, $VSAE[alg]_p$ has five kinds of interface actions:

- **Input** GPSupdate$(l, t)_p, l \in R, t \in \mathbb{R}^{\geq 0}, p \in P$: This input indicates that a process p is currently located at position l.

- **Input** leader$_p, p \in P$: This input communicates that a process is the leader for its current region.

- **Input** torcv$(m)_p, m \in VM, p \in P$: This input is either of a vmsg containing a virtual node layer message to be received by the region or a vstate, which contains the state of a VSA.

- **Output** tocast$(m)_p, m \in VM, p \in P$: This output is either of a vmsg containing a virtual node layer message from the current region's VSA or a vstate message containing the state of the current region's VSA.

- **Output** $\mathsf{prefer}_p, p \in P$: This input indicates that the process is an emulator of its current region's VSA.

It also has the following state variables:

- $clock : \mathbb{R}^{\geq 0} \cup \{\bot\}$: This variable is $\bot$ initially, and then updated to real-time through a GPSupdate input. Once set, it progresses at the rate of real-time.

- $reg : U \cup \{\bot\}$: This variable is $\bot$ initially, but it is updated to $reg(l)$ whenever a $\mathsf{GPSupdate}(l, t)_p$ input occurs.

- $part : Bool$: This Boolean indicates whether the process is attempting to participate in the virtual machine emulation in the current round.

- $leader : Bool$: This Boolean indicates whether the process is currently the leader of its region.

- $vstate : \cup_{u \in U} Q_{alg(u)} \cup \{\bot\}$: This variable stores the local copy of the emulator state, if it is known by the emulator. Otherwise, it is $\bot$.

- $savedq : (Msg \times \mathbb{R}^{\geq 0})^*$: This queue stores timestamp-tagged messages to be received by the VSA. Whenever a vmsg is received via a torcv, the included message is stored together with the current time at the end of $savedq$.

- $outq : Msg^*$: This queue is a queue of outgoing messages for the local region's VSA.

Mobile nodes in a region u use a leader-based emulation algorithm to implement the region u's virtual node. At a high level, a leader is periodically selected in a zone by the leader election service (described in Chapter 10). A leader is responsible for both broadcasting the messages that would have been sent by the virtual machine in its region in the last e time, where e is the $VBDelay$ buffer delay parameter, and broadcasting an up-to-date version of the VSA state. This broadcast is used to both stabilize the state of the emulation algorithm, forcing all emulators in the same region to have the same virtual machine state, and to allow newly joining emulators (those that have just restarted or moved

into the region) to start participating in emulation. This virtual machine state is frozen from the point of the sending of this virtual machine state message, until the mobile nodes again participate in the leader election service. During that time, the virtual machine runs at an accelerated pace, simulating the receipt of messages received from $TOBcast$ while doing so, until the machine is caught up with real-time and the next leader is chosen. Any broadcasts that this emulation of the virtual machine produces are stored in a local outgoing queue for broadcast if the emulator becomes a leader.

We now describe the emulation algorithm in more detail.

Round-based virtual machine emulation. Our VSA emulation algorithm follows a round-based structure. As in the leader election service, time is divided into rounds of length $t_{slice} = e$, where each round begins at a multiple of t_{slice}.

All active simulation of VSA actions is done only in the first d time of a round, after which the VSA state is frozen until the next round. During that d period, each emulator in a region stores and updates the state of the VSA (including the VSA's clock value) locally, simulating all actions of the VSA based on it. To guarantee the VSA emulation satisfies the specifications from Chapter 7 (bounding the time the output trace of the emulation may be behind that of the VSA being emulated), the virtual clock must catch up to real time. This is done by having the virtual clock advance at a rate that allows it to simulate an entire timeslice's worth of the VSA in d time. This is illustrated in Figure 11-3, where the virtual clock proceeds in fits and starts relative to real time, occasionally falling behind and then catching up. It is formally described in lines 34-36.

At any time, when an emulator receives a TObcast message with a vmsg tuple (containing a Vbcast message), it places the message in a local saved message queue (lines 50-52) from which it later simulates the VSA vrcving (processing) the message (lines 72-78). If the VSA is to perform a local action, the emulator simulates its effect on the VSA state (lines 80-87). If the VSA action is to vcast a message, the emulator places the message in an outgoing VSA queue (lines 86-87), to be removed and tocasted in a vmsg message as a VSA message by the leader, in the VSA's stead (lines 89-93). This queue starts each round empty.

Leader responsibilities. For fault-tolerance and load balancing reasons, it is necessary

Signature:

2 $VM = (\{\mathsf{vstate}\} \times U \times \cup_{u \in U} Q_{alg(u)}) \cup (\{\mathsf{vmsg}\} \times Bool \times Msg)$
Input GPSupdate$(l, t)_p$, $l \in R$, $t \in \mathbb{R}$
4 **Input** leader$_p$
Input torcv$(m)_p$, $m \in VM$
6 **Output** tocast$(m)_p$, $m \in VM$
Output prefer$_p$
8 **Internal** participate$_p$
Internal VSArcv$(m)_p$, $m \in Msg$
10 **Internal** VSAlocal$(act)_p$, $act \in \cup_{u \in U}(H_{alg(u)} \cup O_{alg(u)})$
Internal resetRound$_p$
12

State:

14 **analog** $clock$: $\mathbb{R}^{\geq 0} \cup \{\bot\}$, initially $\bot$
reg: $U \cup \{\bot\}$, initially $\bot$
16 $part$, $leader$: $Bool$, initially false
$vstate$: $\cup_{u \in U} Q_{alg(u)} \cup \{\bot\}$, initially $\bot$
18 $savedq$: $(Msg \times \mathbb{R}^{\geq 0})^*$, initially λ
$outq$: Msg^*, initially λ
20

Derived variable: $legal$: $Bool == clock \neq \bot \Rightarrow$
22 $[(leader \Rightarrow clock \ \mathbf{mod}\ t_{slice} = d) \wedge (part \Rightarrow clock\ \mathbf{mod}\ t_{slice} \leq 2d+\epsilon)$
$\wedge\, reg \neq \bot \wedge sorted(savedq) \wedge \forall \langle m,t \rangle \in savedq{:}t \leq clock$
24 $\wedge\, (vstate \neq \bot \Rightarrow [vstate \in Q_{alg(reg)} \wedge \forall \langle m,t \rangle \in savedq{:}t \geq vstate.clock$
$\wedge\, \forall t = clock\ \mathbf{mod}\ t_{slice}{:}([t \geq d \Rightarrow clock\text{-}vstate.clock = t\text{-}d] \wedge [t \in$
26 $(0,2d) \Rightarrow part] \wedge [t \leq d \Rightarrow clock\text{-}vstate.clock = (t\text{-}d)(1\text{-}\frac{t_{slice}}{d})])])]$

28 **Trajectories**:
evolve
30 **if** $clock \neq \bot$ **then**
$\mathbf{d}(clock) = 1$
32 **else constant** $clock$
$\tau(clock).vstate = \tau_{alg(reg)}(\tau(clock).vstate.clock)$
34 **if** $vstate.clock < clock \wedge clock\ \mathbf{mod}\ t_{slice} \leq d$ **then**
$\mathbf{d}(vstate.clock) = t_{slice} / d$
36 **else constant** $vstate$
stop when
38 Any precondition is satisfied.

40 **Transitions**:
Input GPSupdate$(l, t)_p$
42 **Effect**:
if $clock \neq t \vee reg \neq region(l) \vee \neg\, legal$ **then**
44 $clock \leftarrow t$
$reg \leftarrow region(l)$
46 $part, leader \leftarrow$ **false**
$vstate \leftarrow \bot$
48 $savedq, outq \leftarrow \lambda$

50 **Input** torcv$(\langle \mathsf{vmsg}, b, m \rangle)_p$
Effect:
52 $savedq \leftarrow \mathbf{append}(savedq, \langle m, clock \rangle)$

54 **Output** prefer$_p$
Precondition:
56 $clock\ \mathbf{mod}\ t_{slice} = 0 \wedge \neg\, part \wedge vstate \neq \bot$
Effect:
58 $outq \leftarrow \lambda$
$part \leftarrow$ **true**

Internal participate$_p$
Precondition: 62
$clock\ \mathbf{mod}\ t_{slice} = 0 \wedge \neg\, part \wedge vstate = \bot$
Effect: 64
$part \leftarrow$ **true**
66

Input leader$_p$
Effect: 68
if $clock\ \mathbf{mod}\ t_{slice} = d$ **then**
$leader \leftarrow$ **true** 70

Internal VSArcv$(m)_p$ 72
Precondition:
$vstate.clock < clock \wedge \mathbf{next}(vstate, \delta_{alg(reg)}) = \bot$ 74
$part \wedge \langle m, vstate.clock \rangle = \mathbf{head}(savedq)$
Effect: 76
$vstate \leftarrow \delta_{alg(reg)}(vstate, \mathsf{vrcv}(m))$
$savedq \leftarrow \mathbf{tail}(savedq)$ 78

Internal VSAlocal$(act)_p$ 80
Precondition:
$vstate.clock < clock \wedge part$ 82
$act = \mathbf{next}(vstate, \delta_{alg(reg)}) \neq \bot$
Effect: 84
$vstate \leftarrow \delta_{alg(reg)}(vstate, act)$
if $act = \mathsf{vcast}(m)$ **then** 86
$outq \leftarrow \mathbf{append}(outq, m)$
88

Output tocast$(\langle \mathsf{vmsg}, \mathbf{true}, m \rangle)_p$
Precondition: 90
$clock \neq \bot \wedge leader \wedge vstate \neq \bot \wedge m = \mathbf{head}(outq)$
Effect: 92
$outq \leftarrow \mathbf{tail}(outq)$
94

Output tocast$(\langle \mathsf{vstate}, u, vstate' \rangle)_p$
Precondition: 96
$clock \neq \bot \wedge reg = u \wedge leader$
$vstate = vstate' \wedge [vstate = \bot \vee outq = \lambda]$ 98
Effect:
$leader \leftarrow$ **false** 100

Input torcv$(\langle \mathsf{vstate}, u, vstate' \rangle)_p$ 102
Effect:
if $clock\ \mathbf{mod}\ t_{slice} = 2d \wedge (part \vee vstate \neq \bot)$ 104
$\wedge\, reg = u$ **then**
$vstate \leftarrow vstate'$ 106
if $vstate \notin Q_{alg(reg)}$ **then**
$vstate \leftarrow start_{alg(reg)}(clock\text{-}d)$ 108
$vstate.clock \leftarrow clock\ \text{-}d$
$savedq \leftarrow savedq - \{\langle m, t \rangle{:}\ t < clock\ \text{-}d\}$ 110
$part \leftarrow$ **false**
112

Internal resetRound$_p$
Precondition: 114
$clock\ \mathbf{mod}\ t_{slice} = 2d + \epsilon \wedge part$
Effect: 116
$vstate \leftarrow \bot$
$part \leftarrow$ **false** 118

Figure 11-2: VSAE$[alg]_p$, emulator at p of $alg \in VAlgs$.

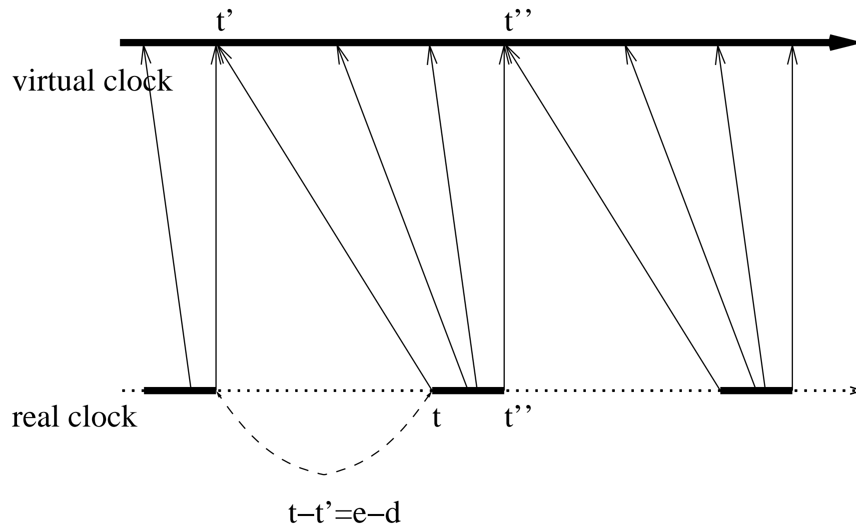


Figure 11-3: Relationship between virtual and real time. A virtual clock behind real time runs faster until it catches up.

to have more than just one process maintaining a VSA. In our virtual machine emulation, at the beginning of a round, each process already emulating the VSA performs a prefer output, and d later at most one of the mobile nodes in a VSA's region is chosen as a leader by the leader election service. Recall from Section 10.1.3 that processes that perform a prefer output are chosen over processes that do not by the leader election service. Hence, a process that is already participating in the emulation of its local region's VSA is chosen as a leader over a process that is not already participating in the emulation. The leader has primary responsibility for performing VSA outputs and helping new emulators join the virtual machine emulation. In our multiple emulator approach, a VSA is maintained by several emulators, including at most one leader. However, only a process that is leader actually performs the sending of the stored messages in $outq$, preventing multiple transmission of messages from the VSA.

To keep emulators consistent, each emulator must develop the VSA variables in the same way, and choose the same discrete actions to perform at the same points in a VSA execution and with the same results. We assume that each emulator chooses the same VSA trajectory from any particular VSA state and uses the same deterministic function **next**, mapping a state and a transition set to a next action to perform. The results of a transition

are also determinized; if more than one state is possible as a result of a transition, then some deterministic method for selecting one of the states is employed by each process. In addition, emulators continue to simulate locally controlled VSA actions until no more are possible before simulating receipt of received messages (line 83), also helping to ensure that local emulator state remains consistent.

Emulation details. There are several complications in VSA emulation that arise due to both message delays and process failure:

Joining: When a node discovers it is in a new region, it sets its local region and clock to match that from GPSupdate, and initializes its remaining variables (lines 41-48). At the beginning of the first full round it is alive for, the process will perform a participate action, setting its local $part$ variable to true (lines 61-65), indicating that the process has been in the region since the beginning of the round, allowing us to conclude that the process expects to receive all TObcast messages sent since the beginning of the round. $2d$ time later, when any process in the region with $part$ set to true or already emulating the VSA receives a vstate message for its region, it computes its region's VSA's state from the information in the message and stores it as the frozen VSA state for use in the next round (lines 102-111). The clock in the resulting VSA state is set to be the time when the VSA state was current, d into the current round. It also removes any messages from its local saved message queue that were sent before the beginning of the round (line 110).

If no such vstate message arrives in the round (meaning the leader of the round failed or left the region before sending such a message), then each process with $part$ set to true, regardless of whether or not the process is emulating the VSA, sets $part$ to false and erases its local VSA state (lines 113-118).

Restarting a VSA: If a process is leader and has no value for the VSA state (implying that all processes that entered the leader competition for that region u in that round were not emulating the VSA), it restarts the emulation (95-99). It does this by sending a vstate message with attached state of $\perp$. Let $\Theta'_{alg(u)}$ be a designated element of $\Theta_{alg(u)}$ and $start_{alg(u)}(t)$ be $\delta_{alg(u)}(\Theta'_{alg(u)}, \mathsf{time}(t))$. When a process in the region that is either emulating the VSA or has $part$ set to true receives such a vstate message, it computes and stores the state $start_{alg(u)}(clock - d)$ as its local VSA state (lines 107-108), corresponding

to having restarted the VSA d time ago and immediately having it receive a time$(clock - d)$ input.

Self-stabilization. In order to make the implementation self-stabilizing, we use local correction during GPSupdate actions and the receipt of the periodic vstate messages sent by leaders. The vstate messages sent by a leader contain state information which overwrites any VSA state information at other emulators, bringing emulators into agreement about VSA state.

11.3 Correctness of the implementation

Here we discuss several aspects of the correctness of our implementation of the VSA layer. Each process runs the *Fail*-transformed composition of its local $CE[alg]$, $VSAE[alg]$, $TOBDelay$, $TOBFilter$, and $LeadCl$ automata (see Figure 11-1). We define our implementation system as the composition of these automata with $RW\|TObcast\|LeadMain$, with certain actions hidden.

Definition 11.1 *For each* $alg \in VAlgs$, *define* $VEmu[alg]$ *to be*
ActHide$(H_{TOspec} \cup H_{Leadspec}, RW\|TObcast\|LeadMain$
$\quad \| \prod_{p\in P} Fail(CE[alg]_p\|VSAE[alg]_p\|LeadCl_p\|TOBDelay_p\|TOBFilter_p))$.

Before continuing, we will first present several definitions useful in our discussion. Then, to show correctness, we use the strategy described in Section 9.3, tweaked slightly to account for the fact that we are building on self-stabilizing implementations of services in our implementation:

1. For each $alg \in VAlgs$, describe a legal set $L^3_{VEmu[alg]}$ of *VEmu[alg]*, and show that it is a legal set (Lemma 11.8).

2. Define a legal set for the specification, and show that the set is a legal set. (The legal set is the relatively trivial one,where $RW\|VW\|Vbcast$ is started in a reachable state, and all other state is arbitrary.)

3. Show that the implementation started in $L^3_{VEmu[alg]}$ implements $VLayer[alg]$ started in a reachable state of $RW\|VW\|Vbcast$ (Lemma 11.15). We show this in the following way:

   (a) Define a simulation relation $\mathcal{R}_{Emu[alg]}$ between *VEmu[alg]* and *VLayer[alg]* (see Definition 11.9). Show the relation is a simulation relation, after some action hiding (Lemma 11.10).

   (b) Show that for each state in $L^3_{VEmu[alg]}$, there exists a state in the invariant set $\{x \in Q_{VLayer[alg]} \mid x\lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}\}$ such that $\mathcal{R}_{Emu[alg]}$ holds between the states (Lemma 11.14) (Recall that $Inv_{RW\|VW\|Vbcast}$, the reachable states of the composition, is defined in Definition 7.5).

4. Show that the set of executions of the implementation started in invariant states of $TOBspec$ and $LeadSpec$ stabilizes to the set of executions of $VEmu[alg]$ started in $L^3_{VEmu[alg]}$ (Lemma 11.19). Notice that this differs slightly from the strategy described in Section 9.3, since we will ultimately be using implementations of the totally ordered broadcast and leader election services that stabilize themselves so that they appear to be starting in invariant states of the two services. As we mention below, we add an additional set of results to the strategy in Section 11.3.4 that allows us to conclude that these stabilizing implementations together with the main emulation algorithm self-stabilize to reach states related to states in $L^3_{VEmu[alg]}$.

5. Conclude that the set of traces of $VEmu[alg]$ stabilizes to the set of traces of executions of *VLayer[alg]* starting in $\{x \in Q_{VLayer[alg]} \mid x\lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}\}$ (Theorem 11.20).

We also conclude another result (Theorem 11.21), which constrains the execution fragments of $VLayer$ that implementation fragments correspond to after stabilization. We then add one more set of results, connecting the implementations of the totally ordered broadcast and leader election services to conclude that we actually have a stabilizing emulation of the virtual node layer (Section 11.3.4).

```
procVstate_u(t: ℝ^≥0, sentseq: (Msg×U×P×ℝ^≥0)*): Q_alg(u) ∪ {⊥}
2                                                             ==
   rVstate: Q_alg(u) ∪ {⊥} ← ⊥
4  while sentseq ≠ λ
     if head(sentseq) = ⟨⟨vstate, u, vstate⟩,v,p,t⟩ ∧ v ∈ nbrs+(u) then
6        rVstate ← vstate
         if rVstate ∉ Q_alg(u) then
8          rVstate ← start_alg(u)(t)
         rVstate.clock ← t
10   sentseq ← tail(sentseq)
   return rVstate
12
procVmsgs(u:U,t:ℝ^≥0,sentseq:(Msg×U×P×ℝ^≥0)*):(Msg×ℝ^≥0)*
14                                                            ==
   rSeq: (Msg ×ℝ^≥0)* ← λ
16 while sentseq ≠ λ
     if head(sentseq) = ⟨⟨vmsg,v,m⟩,w,p,t'⟩ ∧ w ∈ nbrs+(u) ∧ t' ≥ t
18                                                          then
         rSeq ← append(rSeq, ⟨m, t' + d⟩)
20   sentseq ← tail(sentseq)
   return rSeq
```

```
to_rcvVmsgs(to_rcv:(Msg×U)*,t:ℝ^≥0):(Msg×ℝ^≥0)* ==
   rSeq: (Msg ×ℝ^≥0)* ← λ                                          46
   while to_rcv ≠ λ
     if head(to_rcv) = ⟨⟨vmsg, u, m⟩, v⟩ then                       48
         rSeq ← append(rSeq, ⟨m, t⟩)
     to_rcv ← tail(to_rcv)                                         50
   return rSeq
                                                                   52
lookAhead(u: U, vstate, vstate': Q_alg(u), savedq:
     (Msg×ℝ^≥0)*,outq,to_send:Msg*):Boolean ==                     54
   return ∃τ1 a1 τ2 a2 ··· τn ∈ frags_V_alg(u):
       1. τ1.fstate = vstate ∧ τn.lstate = vstate'                 56
       2. next(τn.lstate, δ_alg(u)) = ⊥
       3. ∀ai: ai = next(τi.lstate, δ_alg(u)) ≠ ⊥                   58
              ∨ (next(τi.lstate, δ_alg(u)) = ⊥
                    ∧ ∃m ∈ Msg : ai = vrcv(m))                     60
       4. Let ⟨m1, t1⟩, ··· , ⟨mm, tm⟩ = sequence
          of received messages and vstate.clock                   62
          values for the vrcv actions in a1, ··· , an.
          Then savedq = ⟨m1, t1⟩, ··· , ⟨mm, tm⟩.                   64
       5. Let m1, ··· ml = sequence of sent messages
          for the vcast actions in a1, ··· , an.                  66
          Then to_send = append(outq, (m1, ··· , ml)).
```

Figure 11-4: Functions for use in correctness proofs.

Now, we'll describe several functions and definitions helpful for the rest of the chapter (Figure 11-4). This next definition is simply shorthand for the queue of messages in $TObcast.oldsent$ followed by the head of $TObcast.sent$ if the message has already been processed by $TObcast$ for the input process.

Definition 11.2 *Define* $procSent(p)$ *to be* $oldsent$ *if* $p \notin procs$ *or* $append(oldsent, head(sent))$ *if* $p \in procs$.

The functions in Figure 11-4 are described in more detail below:

- $procVstate_u : (\mathbb{R}^{\geq 0} \times (Msg \times U \times P \times \mathbb{R}^{\geq 0})^*) \rightarrow Q_{alg(u)} \cup \{\bot\}, u \in U$: Consider $procVstate_u(t, sentseq)$. This function takes a time t and a sequence of message tuples $sentseq$, and returns a state for the VSA in region u. It finds the last vstate tuple, of the form $\langle\langle \mathsf{vstate}, u, vstate\rangle, v, p, t\rangle$ in the sequence. A state for the VSA in region u is then calculated based on the tuple's $vstate$: if $vstate \in Q_{alg(u)}$ then the function returns $vstate$ after replacing $vstate.clock$ with t, and if not then the function returns $start_{alg(u)}(t)$. (We later use this function to calculate the state of a region's VSA based on vstate messages that have been sent (Section 11.3.2).)

- $procVmsgs : (U \times \mathbb{R}^{\geq 0} \times (Msg \times U \times P \times \mathbb{R}^{\geq 0})^*) \rightarrow (Msg \times \mathbb{R}^{\geq 0})^*$: Consider $procVmsgs(u, t, sentseq)$. This function takes a region u, time t, and sequence of

message tuples $sentseq$, and returns a sequence of timestamped messages. It takes all tuples in $sentseq$ of the form $\langle\langle \textsf{vmsg}, v, m\rangle, w, p, t'\rangle$, where $w \in nbrs^+(u)$ amd $t' \geq t$, and returns the sequence projected onto m and t', after adjusting t' up by d. (We later use this function to calculate the list of messages to be received by a VSA based on information in $TObcast$ (Section 11.3.2). The timestamp indicates the virtual time at which the VSA should receive the message.)

- $to_rcvVmsgs : ((Msg \times U)^* \times \mathbb{R}^{\geq 0}) \rightarrow (Msg \times \mathbb{R}^{\geq 0})^*$: Consider $to_rcvVmsgs(to_rcv, t)$. This function takes a sequence to_rcv of messages tagged with regions and a time t, and returns a sequence of timestamped messages. It selects the messages in to_rcv of the form $\langle\langle \textsf{vmsg}, u, m\rangle, v\rangle$, and returns the sequence projected onto m and then paired with t. (This function is used for a reason similar to the one for why we use $procVmsgs$. It calculates messages to be received by a VSA based on information in $TOBFilter$. (See Section 11.3.2.))

- $lookAhead : (U \times Q_{alg(u)} \times Q_{alg(u)} \times (Msg \times \mathbb{R}^{\geq 0})^* \times Msg^* \times Msg^*) \rightarrow Bool$: Consider $lookAhead(u, vstate, vstate', savedq, outq, to_send)$. This function takes a region u, an early state of u called $vstate$, an ending state of u called $vstate'$, a queue of timestamped messages to process called $savedq$, and queues of outgoing messages called $outq$ and to_send, and returns a Boolean indicating whether there exists some execution of $alg(u)$ such that:

  1. The execution begins in $vstate$ and ends in $vstate'$.
  2. There are no locally controlled actions enabled in $vstate'$.
  3. Each locally controlled action in the execution is the one arrived at from use of the function **next**, and no vrcv actions occur unless no locally controlled action is enabled.
  4. Consider the sequence of vrcv actions in the execution, and construct a sequence of tuples corresponding to the messages in the vrcv actions, paired with the value of $vstate.clock$ when the action occurred. This sequence is equal to $savedq$.

5. Consider the sequence of vcast actions in the execution, and construct a sequence corresponding to the messages in the vcast actions. Then $outq$ followed by this sequence is equal to to_send.

In other words, this function takes a VSA in state $vstate$ and with a queue of outgoing messages $outq$ and indicates whether the VSA can then consume the messages in $savedq$ in a carefully prescribed way and end in state $vstate'$ with a new sequence of outgoing messages to_send. (We later use this function to verify that the frozen state of a VSA emulation is consistent with a future abstract VSA state. (See Section 11.3.2.))

11.3.1 Legal sets

Fix some $alg \in VAlgs$. Here we describe a legal set for *VEmu[alg]*. Recall from Lemma 3.13 that a legal set of states for a TIOA is one where each closed execution fragment starting in a state in the set ends in a state in the set. We break down the legal set definition into three legal sets in order to simplify the proof reasoning and more easily prove stabilization in Section 11.3.3.

Legal state set $L^1_{VEmu[alg]}$:

The first set of legal states describes some properties that become true at an alive process at the time of the first GPSupdate for the process.

Definition 11.3 *$L^1_{VEmu[alg]}$ is the set of states x of $VEmu[alg]$ where each of the following properties hold:*

1. *$x \lceil X_{LeadSpec} \in Inv_{LeadSpec}$ and $x \lceil X_{TOBspec} \in Inv_{TOBspec}$.*
   This says that the state is such that when restricted to the variables of $LeadSpec$ or the variables of $TOBspec$, the result is in the respective invariant set.

2. *For each $p \in P$: $\neg failed_p \wedge clock_p \neq \bot$ (nonfailed client with a non-$\bot$ clock value):*

(a) $reg_p = reg(p) \wedge clock_p = LeadCl_p.clock = now \wedge updated_p \wedge rtimer_p \neq \bot$.

This says that the client's reg matches its actual region, its clock is set to the real time as is its clock in $LeadCl$, and its $rtimer$ in $TOBcast$ has started running.

(b) $[leader_p \Rightarrow clock_p \mod t_{slice} = d] \wedge [part_p \Rightarrow clock_p \mod t_{slice} \leq 2d + \epsilon]$.

This says that if the $leader$ bit is set, then it is d into the current round. Also, if $part$ is set, then it is at most $2d + \epsilon$ into the current round.

(c) $sorted(savedq_p) \wedge \forall \langle m, t \rangle \in savedq_p : t \leq clock_p$.

This says that the elements of $savedq$ are ordered with respect to timestamp, and that the highest timestamp the can be observed is the current time.

3. *For each $p \in P : \neg failed_p \wedge clock_p \neq \bot \wedge vstate_p \neq \bot$ (nonfailed client with a non-$\bot$ $clock$ value and a non-$\bot$ $vstate$):*

(a) $vstate_p \in Q_{alg(reg_p)} \wedge \forall \langle m, t \rangle \in savedq_p : t \geq vstate_p.clock$.

This says that the client's $vstate$ must be a state of the client's current region VSA, and that all messages in $savedq$ must have timestamps that are not smaller than the $vstate$'s clock value.

(b) $clock_p \mod t_{slice} \in (0, 2d) \Rightarrow part_p$.

This says that if the round is greater than 0 but less than $2d$ old, then $part$ is true.

(c) $clock_p \mod t_{slice} \geq d \Rightarrow clock_p - vstate_p.clock = (clock_p \mod t_{slice}) - d$.

This says that when the round is at least d old, then the virtual clock's value is set to equal what the real time was d into the current round.

(d) $clock_p \mod t_{slice} \leq d \Rightarrow$

$$clock_p - vstate_p.clock = (d - clock_p \mod t_{slice})(\tfrac{t_{slice}}{d} - 1).$$

This says that when the round is at most d old, then the virtual clock's value is behind the real time by exactly the amount of time remaining until the round is d old, times $(t_{slice} - d)/d$.

It is easy to observe that $L^1_{VEmu[alg]}$ is a legal set for the implementation.

Lemma 11.4 *$L^1_{VEmu[alg]}$ is a legal set for $VEmu[alg]$.*

Legal state set $L^2_{VEmu[alg]}$:

The next legal set describes a subset of states of $L^1_{VEmu[alg]}$ that satisfy some additional properties with respect to the relationship between states of the leader election algorithm and the core emulation algorithm.

Definition 11.5 *$L^2_{VEmu[alg]}$ is the set of states x of $VEmu[alg]$ where each of the following hold:*

1. *$x \in L^1_{VEmu}$.*
   This says that L^2_{VEmu} is a subset of L^1_{VEmu}.

2. *For each $p \in P : \neg failed_p \wedge clock_p \neq \bot$ (nonfailed client with non-$\bot$ $clock$ value):*

   (a) $part_p \Rightarrow rtimer_p \geq \min(d, clock_p \mod t_{slice})$.
   This says that if $part$ is set, then $TOBcast$'s $rtimer$ is either d or is at least as large as the age of the current round.

   (b) $[pref_p \Rightarrow vstate_p \neq \bot] \wedge [vstate_p \neq \bot \Rightarrow rtimer_p = d]$.
   This says that if $pref$ is set in the leader election service, then $vstate$ is not $\bot$. Also, if $vstate$ is not $\bot$ then $rtimer$ must be equal to d.

   (c) $participated_p \Rightarrow (part_p \vee clock_p \mod t_{slice} = 0)$.
   This says that if $participated$ is set, then either $part$ is set or the round has just begun.

   (d) $[clock_p \mod t_{slice} \leq d \wedge part_p]$
   $\Rightarrow [(clock_p \mod t_{slice} = 0 \wedge \neg participated_p \wedge [pref_p \Leftrightarrow vstate_p \neq \bot])$
   $\vee ([participated_p \vee p \in serviced] \wedge [pref(p) \Leftrightarrow vstate_p \neq \bot] \wedge$
   $cand(reg_p) \neq \bot \wedge [vstate_p \neq \bot \Rightarrow \exists q \in P : cand(reg_p) = \langle q, true \rangle])]$.
   This says that if $part$ is set and the round is at most d old, then either: (a) the round has just begun, $pref_p$ indicates whether $vstate$ is not $\bot$, and a prefer′ *is about to occur; or (b) $participated$ is set or p is in $serviced$, $pref(p)$ indicates*

whether $vstate_p \neq \bot$, some process is the leader candidate for reg_p, and if $vstate$ is not $\bot$ then that leader candidate's pair is a "true" pair.*

(e) $leader_p \Rightarrow [\neg participated_p \wedge cand(reg_p) = \langle p, pref(p)\rangle]$.
This says that if $leader$ is set, then $participated$ is not set and $\langle p, pref(p)\rangle$ is the leader tuple for reg_p.

3. *For each $p \in P : \neg failed_p : \forall u, u' \in U$:*

   (a) $[(u = reg^-(p) \wedge \exists\langle \mathsf{vmsg}, true, m\rangle \in to_send^-_p) \vee (u = reg_p \wedge \exists\langle \mathsf{vmsg}, true, m\rangle \in to_send^+_p)]$
   $\Rightarrow [now \mod t_{slice} = d \wedge \exists b \in Bool : \langle p, b\rangle = cand(u)]$.
   This says that if a non-failed client has a $\langle \mathsf{vmsg}, true, m\rangle$ tuple in one of its $TOBDelay$ buffers then the round is exactly d old, and the process is the one that won the leader competition for the region of the vmsg *tuple.*

   (b) $[(u = reg^-(p) \wedge \exists\langle \mathsf{vstate}, u', q\rangle \in to_send^-_p) \vee (u = reg_p \wedge \exists\langle \mathsf{vstate}, u', q\rangle \in to_send^+_p)] \Rightarrow [now \mod t_{slice} = d \wedge \neg leader_p \wedge \neg participated_p \wedge u = u' \wedge \exists b \in Bool : \langle p, b\rangle = cand(u) \wedge \forall i \in [1, |to_send^-_p to_send^+_p|] : [to_send^-_p to_send^+_p[i] = \langle \mathsf{vstate}, u, q'\rangle \Rightarrow \forall j > i : to_send^-_p to_send^+_p[j] \notin \{\langle \mathsf{vstate}, u, q\rangle \mid q \in Top\} \cup \{\langle \mathsf{vmsg}, true, m\rangle \mid m \in Msg\}]]$.
   This says that if a non-failed client has a vstate *message in one of its $TOBDelay$ buffers then the region tag on the message corresponds to the region it was broadcast in, the client is the one that won the leader competition for that region, the process will not be performing more leader-related actions, and no* vmsg *or* vstate *messages for the region's VSA were sent after the* vstate *message.*

4. $\forall u \in U : \forall v \in nbrs^+(u) : [(now \mod t_{slice} = d \wedge [\exists\langle\langle \mathsf{vstate}, u, q\rangle, v, p, now\rangle \in sent : v \in nbrs^+(u) \vee \exists\langle\langle \mathsf{vmsg}, true, m\rangle, u, p, now\rangle \in sent : \nexists b \in Bool : \langle p, b\rangle = cand(u)]) \Rightarrow \forall p \in P : \forall b \in Bool : [(\neg failed_p \wedge cand(u) = \langle p, b\rangle) \Rightarrow ([u = reg_p \Rightarrow (\neg participated_p \wedge \neg leader_p \wedge \forall m \in Msg : \langle \mathsf{vmsg}, true, m\rangle \notin to_send^+_p)]$
   $\wedge [u = reg^-(p) \Rightarrow \forall m \in Msg : \langle \mathsf{vmsg}, true, m\rangle \notin to_send^-_p]$

$\wedge \forall q \in Top : \langle \mathsf{vstate}, u, q\rangle \notin to_send^-_p to_send^+_p)]]$.

This says that if a vstate *message for a region exists or if a* vmsg *for the region exists but was sent by a process that did not win the region's leader competition, then the process that won the region's leader competition will not be producing any* vstate *or* vmsg *messages for the region, and does not have any such messages in its* $TOBDelay$ *buffers. (In other words, any* vstate *and* vmsg *messages in existence are not going to be second-guessed by another process sending more messages, and we can find a single virtual layer state to map to that won't be changed based on leader actions. Non-leader messages might have been sent, but they are not problematic if the leader won't be performing any more emulation-related broadcasts.)*

5. *For each* $p \in P : \neg failed_p : [clock \mod t_{slice} = d \wedge \neg leader_p \wedge \neg participated_p \wedge part_p \wedge cand(reg_p) = \langle p, pref(p)\rangle] \Rightarrow \exists q \in Top : (\langle \mathsf{vstate}, reg_p, q\rangle \in to_send^-_p to_send^+_p \vee \langle\langle \mathsf{vstate}, reg_p, q\rangle, reg_p, p, now\rangle \in sent)$.

   This says that if the round is d *old,* $leader$ *and* $participated$ *are not set,* $part$ *is set, and* $\langle p, pref(p)\rangle$ *is the leader pair (meaning the client has completed its leader duties), then a* vstate *message for the region has been sent by the client.*

Lemma 11.6 $L^2_{VEmu[alg]}$ *is a legal set for* $VEmu[alg]$.

Proof: Let x be any state in $L^2_{VEmu[alg]}$. By Definition 3.12 of a legal set, we must verify two things for state x:

- For each state x' of $VEmu[alg]$ and action a of $VEmu[alg]$ such that (x, a, x') is in the set of discrete transitions of $VEmu[alg]$, state x' is in $L^2_{VEmu[alg]}$.
- For each state x' and closed trajectory τ of $VEmu[alg]$ such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in $L^2_{VEmu[alg]}$.

By Lemma 11.4, we know that if x satisfies the first property of $L^2_{VEmu[alg]}$, then any discrete transition of $VEmu[alg]$ will lead to a state x' that still satisfies the first property, and any closed trajectory starting with state x will end in some state that satisfies the first property. This implies that we just need to check that in the two cases of the legal set

definition, the state x' satisfies all parts of the remaining properties of $L^2_{VEmu[alg]}$. Since the state of $CE[alg]_p$ is not constrained in the legal set definition, we consider only the tocast outputs of those automata while checking the legal set properties.

For the first case of the legal set definition, the proof is one large, rather simple, case analysis for each action. For each action, most properties are trivial to verify:

- fail$_p$, restart$_p$, drop(p), resetRound$_p$, VSArcv$(m)_p$, VSAlocal$(act)_p$, torcv$'(m, u)_p$: All properties will still hold in state x' after any of these actions.

- GPSupdate$(l, t)_p$: Since x is in $L^1_{VEmu[alg]}$, we know that the only time a state change could occur that might affect any of the properties is if the process is changing regions. However, in that case, local boolean variables are changed to be mostly false, making properties 2 and 5 trivially hold. Properties 3 and 4 are also easy to verify in this case.

- tocast$(m)_p$: The cases of interest to check are where the message being sent is a true-tagged vmsg message or a vstate message.

  When the message is a true-tagged vmsg message, the only interesting properties to check are properties 3 and 4. For property 3(a), we need to check that $now \mod t_{slice} = d$ and p is the winner of the leader competition for its current region. That $now \mod t_{slice} = d$ follows from the fact that the precondition specifies that $leader_p$ must hold, which implies that $now \mod t_{slice} = d$ because property 2(a) and property 2(b) of $L^1_{VEmu[alg]}$ held in state x. That the process is the winner of the leader competition follows from the fact that property 2(e) holds in state x.
  For property 3(b) , we need to check that there were not already any vstate messages for the region in a $TOBDelay$ queue. However, the fact that $leader_p$ was a precondition for the action implies that in state x there could have been no such messages in a $TOBDelay$ queue. Similar reasoning reveals that property 4 also must still hold.

  When the message is a vstate message, the interesting cases to check are for properties 3(b), 4, and 5. Property 5 is easy to immediately see since a vstate message is added to a $TOBDelay$ queue. Property 3(b) and 4 hold for reasons similar to reasons they held in the vmsg case.

- $\mathsf{tocast}'(m, f)_p$: The interesting cases to check are for properties 4 and 5.

  For property 4, the main thing to check is that if the message transferred to $TObcast$ is a vstate for some region u or if it is a vmsg purportedly for the region's VSA but not sent by the leader, then there wasn't already a vstate or vmsg for the region in a process's $TOBDelay$ buffer and that the leader of the region would not be submitting any. The case where the message is a vmsg follows from the fact that property 3(a) held in state x, making the vmsg case impossible. The vstate case follows from the fact that property 3(b) held in state x, meaning that the leader must be done doing work for the round and there are no other vstate or vmsg messages for the region in its $TOBDelay$ buffers.

  For property 5, it is trivial to see that since a vstate message tuple is only transferred via tocast′ if the vstate tuple was in a to_send queue. Also, since property 3(b) held in state x, the region attached to the message was the correct one with respect to the to_send queue it was in. As a result of the tocast′ action, the tuple is decorated with the same region as in the to_send tuple and put into $sent$, implying property 5 holds in state x.

- $\mathsf{torcv}(m)_p$: The only interesting case to check is that of the receipt of a vstate tuple message. The property that is interesting to check is the second conjunct of property 2(b). We need to show that if $vstate_p \neq \perp$, then $rtimer_p = d$. If $vstate_p$ is not updated by this action, then the fact that this property held in state x implies it still holds in state x'. If $vstate_p$ is updated by this action, then it must be that either $vstate_p \neq \perp$ in state x or $part_p$ was true. If $vstate_p$ was not $\perp$ in state x, then the fact that this property held in state x implies that $rtimer_p$ is still equal to d. If $part_p$ was true, then by the fact that property 2(a) held in state x, we know that $rtimer_p \geq \min(d, clock_p \mod t_{slice})$. For $vstate_p$ to have been updated, we know that $clock_p \mod t_{slice} = 2d$, implying that $rtimer_p = d$.

- reset: Since the precondition for this leader election action specifies that a round is more than d old, all properties will hold in x'.

- prefer_p: The properties that must be checked are 2(a)-2(d).

  For property 2(a), notice that the action sets $part_p$ to true. Since the precondition for the action says that $clock_p \mod t_{slice} = 0$, we must show that $rtimer_p \geq 0$. In other words, we need to check that $rtimer_p$ is not $\perp$. This follows from property 2(a) of $L^1_{VEmu[alg]}$.

  For property 2(b), notice that the precondition for the action says that $vstate_p \neq \perp$, and that one of the results of the action is the setting of $pref_p$ to true. This implies that the first conjunct in property 2(b) holds. To see that the second conjunct holds, we need to check that $rtimer_p = d$. However, since $vstate_p$ was not $\perp$ in state x and this property held in state x, we know that $rtimer_p = d$ in state x. Since $rtimer_p$ and $vstate_p$ are not changed by this action, we can conclude that $rtimer_p = d$.

  For property 2(c), notice that a result of the action is that $participated_p$ is set to false, making property 2(c) trivially true.

  For property 2(d), notice that since a precondition of the action is that $clock_p \mod t_{slice} = 0$, and a result of the action is that $participated_p$ is set to false and $pref_p$ is set to true, property 2(d) holds.

- $\mathsf{prefer}'(val)_p$: The properties that must be checked are 2(c) and 2(d).

  For property 2(c), since one of the preconditions for the action is that $clock_p \mod t_{slice} = 0$, this property trivially holds.

  For property 2(d), since $participated_p$ is set to true by this action, we must show that if $part_p$ is true, then $cand(reg_p)$ is set to a non-$\perp$ value, $pref(p)$ indicates whether or not $vstate_p \neq \perp$, and if $vstate_p \neq \perp$ then some true-tagged process is $cand(reg_p)$. Since two of the preconditions for this action are that $participated_p$ is not true and $val = pref_p$, if $part_p$ is true then since property 2(d) held in state x it must have been the case that $pref_p$ indicated whether $vstate_p \neq \perp$. As a result of this action, we know then that $pref(p)$ would also indicate this. Also as a result of this action, we know that $cand(reg_p)$ will not be set to $\perp$, and that if $vstate_p \neq \perp$ then $cand(reg_p)$ would be set to some "true" pair.

- leader_p: The properties to verify are properties 2(e), 3(b), and 4. Since the two preconditions of this action are that $clock_p \mod t_{slice} = d$ and that $participated_p$ be set, we know that properties 3(b) and 4 trivially still hold. For property 2(e), we need to verify that $participated_p$ is false and that $cand(reg_p) = \langle p, pref(p) \rangle$. That $participated_p$ is false is a result of the action. That $cand(reg_p) = \langle p, pref(p) \rangle$ is true is because properties 4 and 7(e) of $Inv_{LeadSpec}$ hold in state x.

- $\mathsf{leader}'(val)_p$: The only nontrivial check is for property 2(d). However, this is also easy to check since the property is assumed to have held in state x and a precondition for this action is that $now \mod t_{slice} \neq 0$.

- $\mathsf{participate}_p$: The properties that must be checked are 2(a)-2(d). The reasoning for property 2(a) is the same as for prefer_p.

  For property 2(b), since one precondition of the action is that $vstate_p = \bot$, we must show that $pref_p$ does not hold. This follows from the fact that this property held in state x, when $vstate_p$ also was equal to $\bot$, and $pref_p$ is not updated as a result of this action.

  For property 2(c), since one precondition of the action is that $clock_p \mod t_{slice} = 0$, the property trivially holds.

  For property 2(d), we must verify that either $participated_p$ is false and $pref_p$ is false, or that $participated_p$ is true, $pref(p)$ is false, and $cand(reg_p)$ is not $\bot$. Since property 2(c) held in state x and $pref_p$ is not changed by this action, we know that $pref_p$ is false in state x'. This means we just have left to show that if $participated_p$ is true, then $pref(p)$ is false and $cand(reg_p)$ is not $\bot$. However, since property 2(c) held in state x and this action does not change the value of $participated_p$, we know that $participated_p$ must have been true in state x as well. By property 7(c) and 7(d) of $Inv_{LeadSpec}$, we know that $pref(p)$ is false and $cand(reg_p)$ is not $\bot$ in state x. Since none of these variables were modified by this action, the property must still hold in state x'.

For the second case of the legal set definition, we now consider any closed trajectory τ such

that $x = \tau.fstate$. Let x' be $\tau.lstate$. We must show that $x' \in L^2_{VEmu[alg]}$. The interesting cases to verify are for properties 2(a), 2(c), 2(d), and 3. Property 2(a) is preserved by the fact that $rtimer$ and $clock$ variables both increase at the same rate until $rtimer$ hits d. Property 2(c) is preserved because of stopping conditions on lines 56 and 63 that force the $part$ variable to be changed to true when $clock \mod t_{slice} = 0$. Property 2(d) is preserved because of leader election service stopping conditions forcing a process with false $participated$ to perform a prefer′ action when $now \mod t_{slice} = 0$. Property 3 is preserved because stopping conditions for $TOBDelay$ force messages in to_send buffers to immediately be sent. ■

Legal state set $L^3_{VEmu[alg]}$:

The final legal set describes a subset of states of $L^2_{VEmu[alg]}$ from which the system demonstrates consistency for the emulated state of a VSA.

Definition 11.7 *L^3_{VEmu} is the set of states x of $VEmu$ where each of the following hold:*

1. $x \in L^2_{VEmu}$.
   This says that L^3_{VEmu} is a subset of L^2_{VEmu}.

2. *For each $p \in P : \neg failed_p \wedge clock_p \neq \bot$ (non-failed client with non-$\bot$ $clock$ value):*

   (a) $[part_p \vee clock_p \mod t_{slice} = 0] \Rightarrow procVmsgs(reg_p, t_{slice}\lfloor now/t_{slice}\rfloor, procSent(p)) = append\ (savedq_p - \{\langle m, t\rangle \mid t - d < t_{slice}\lfloor now/t_{slice}\rfloor\}, to_rcvVmsgs(to_rcv_p, now))$.
   This says that if $part$ is set, then each message sent since the beginning of the current round that can be received by the client's current region will be received by the client or has been stored in the client's $savedq$.

   (b) $part_p \Rightarrow \forall \langle\langle \mathsf{vstate}, reg_p, vstate'\rangle, v, q, d + t_{slice}\lfloor clock_p/t_{slice}\rfloor\rangle \in procSent(p) : (v \notin nbrs^+(reg_p) \vee \langle\langle \mathsf{vstate}, reg_p, vstate'\rangle, v\rangle \in to_rcv_p)$.
   This says that if $part$ is set then it has not yet received a vstate *message sent at d into the current round for its current region.*

(c) $[vstate_p \neq \perp \wedge \neg part_p] \Rightarrow [procVmsgs(reg_p, vstate_p.clock - d, procSent(p)) = append(savedq_p, to_rcvVmsgs(to_rcv_p, now))$
$\wedge(vstate_p = procVstate(d + t_{slice}(\lceil now/t_{slice}\rceil - 1), procSent(p))$
$\vee [clock_p \mod t_{slice} = 2d \wedge \exists\langle\langle \mathsf{vstate}, reg_p, vstate'\rangle, v\rangle \in to_rcv_p])]$.

This says that if $vstate$ is not $\perp$ and $part$ is not set, then each message sent since d before the client's virtual clock time that can be received by the client's current region will be received by the client or has been stored in the client's $savedq$, which contains no other messages but these. Also, either the client's $vstate$ is equal to the one from the last vstate *message for the region, or the round is $2d$ old and the client is about to receive such a message.*

(d) $[vstate_p \neq \perp \wedge part_p \wedge clock_p \mod t_{slice} = 0] \Rightarrow$
$\exists seq = \langle m_1, vstate_p.clock\rangle, \langle m_2, vstate_p.clock\rangle \cdots, \langle m_n, vstate_p.clock\rangle :$
$[lookAhead(reg_p, procVstate(reg_p, vstate_p.clock, procSent(p)), vstate_p, seq, \lambda, outq_p)$
$\wedge procVmsgs(reg_p, vstate_p.clock - d, procSent(p)) =$
$append(seq, append(savedq_p, to_rcvVmsgs(to_rcv_p, now)))]$.

This says that if the round has just begun, $vstate$ is not $\perp$, and the process has already performed a prefer *output, then there is a tagged sequence seq of messages such that seq followed by $savedq$ and the* vmsg *messages about to be received from $TOBFilter$ is equal to the sequence of* vmsg *tagged messages sent d before $vstate.clock$ for receipt in the current region and processed for p by $TObcast$. Also, $vstate$ and $outq$ are consistent with the state and outgoing buffer that would result if the virtual machine ran starting from the attached virtual state of the last* vstate *message for the region in the prior round, and performed* vrcv *actions based on the messages and timestamps in seq.*

Lemma 11.8 *$L^3_{VEmu[alg]}$ is a legal set for $VEmu[alg]$.*

Proof: Let x be any state in $L^3_{VEmu[alg]}$. By Definition 3.12 of a legal set, we must verify two things for state x:

- For each state x' of $VEmu[alg]$ and action a of $VEmu[alg]$ such that (x, a, x') is in the set of discrete transitions of $VEmu[alg]$, state x' is in $L^3_{VEmu[alg]}$.

- For each state x' and closed trajectory τ of $VEmu[alg]$ such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in $L^3_{VEmu[alg]}$.

By Lemma 11.6, we know that if x satisfies the first property of $L^3_{VEmu[alg]}$, then any discrete transition of $VEmu[alg]$ will lead to a state x' that still satisfies the first property, and any closed trajectory starting with state x will end in some state that satisfies the first property. This implies that we just need to check that in the two cases of the legal set definition, the state x' satisfies all parts of the remaining properties of $L^3_{VEmu[alg]}$. Since the state of $CE[alg]_p$ is not constrained in the legal set definition, we only consider the tocast outputs of those automata while checking the legal set properties.

For the first case of the legal set definition, we consider each action:

- fail_p, $\mathsf{restart}_p$, reset, $\mathsf{prefer}'(val)_p$, leader_p, $\mathsf{leader}'(val)_p$, $\mathsf{tocast}(m)_p$, $\mathsf{tocast}'(m, f)_p$, $\mathsf{drop}(p)$: All properties will still hold in state x' after any of these actions.

- $\mathsf{GPSupdate}(l, t)_p$: The only interesting case is where GPSupdate changes the region of a process. However, in that case, emulation-related Boolean variables are all set to false, $savedq$ is cleared, and $vstate$ is set to $\perp$, making property 2 trivially hold.

- $\mathsf{torcv}(m)_p$: If the message is a vmsg message, then the interesting properties to check are properties 2(a), 2(c), and 2(d).

  For property 2(a), it is easy to see that if $part$ is true, it must have also held in state x and that the property holds because the message at the head of $to_rcvVmsgs(x(to_rcv_p), now)$ is now moved to the end of $savedq$, preserving the property. The reasoning for properties 2(c) and 2(d) is similar.

  If the message is a vcast message for the process's current region and the round is exactly $2d$ old, then the interesting properties to check are properties 2(b) and 2(c).

  For property 2(b), if $part_p$ was set in state x, then a result of this action is that $part_p$ is set to false, making this property hold. For property 2(c), this action only changes $vstate$ or $part$ if $part$ held in state x or $vstate$ was not $\perp$. The result of the

action is then to set $vstate$ to a non-$\perp$ value consistent with the one in the m, strip $savedq$ of messages sent before the start of the round, and set $part$ to false. That $procVmsgs$ has the appropriate relationship to $savedq$ and to_rcv holds because in state x either $vstate$ was not $\perp$, implying that this statement held in that state, or $part_p$ held, implying property 2(a) held in state x, and hence still holds in this one. For the second conjunct we must show this vstate message was the one that $procVstate$ uses to calculate the virtual state it returns or that that message is still in to_rcv. We know by property 7(d) of $Inv_{TOBspec}$ that to_rcv contains a suffix of the messages that the process was to receive in $procSent(p)$. Hence, if there exists no other vstate message for the region in to_rcv we know that m must have been the one consistent with the result of $procVstate$. (This is partly because properties 2(a) and 2(b) of $L^2_{VEmu[alg]}$ tells us that $rtimer$ must be d and hence that the process will receive any messages that should be received by processes in its region.)

- torcv$'(m, u)_p$: The reasoning for this action is very similar to the reasoning for torcv$(m)_p$.

- prefer$_p$: The reasoning for properties 2(a) and 2(b) mirror those of the participate$_p$ action. The only additional property to check for this action is property 2(d). Since a precondition of the action is that $vstate_p \neq \perp$ and $clock_p \mod t_{slice} = 0$ and a result is that $part_p$ is set to true, we note that since property 2(c) held in state x, it must be that $vstate_p = procVstate(d + t_{slice}(\lceil now/t_{slice} \rceil - 1), procSent(p))$ and $procVmsgs(reg_p, vstate_p.clock - d, procSent(p)) = append(savedq_p, to_rcvVmsgs(to_rcv_p, now))$. We also know that $outq$ is set to λ by this action. Hence, it is apparent that by selecting $seq = \lambda$, the condition holds.

- participate$_p$: Since a precondition of this action is that $vstate = \perp$ and the result is that $part$ gets set to true, the only properties we need to verify are properties 2(a) and 2(b). Property 2(b) trivially holds since no such vstate messages could yet exist. To show property 2(a) holds, notice that the left hand side of the equality consists of no messages, due to properties 4 and 5 of $Inv_{TOBspec}$. Also property 7(d) of $Inv_{TOBspec}$ implies that the result of $to_rcvVmsgs$ is also empty. Hence, all that remains is to

show that $savedq$ contains no messages tagged with the current time. This follows from the fact that a precondition is that $clock \mod t_{slice} = 0$ and this property held in state x, which implies that $savedq$ then contained no such tagged messages.

- VSArcv$(m)_p$: Since a precondition of this action is that $part$ is true and that the message at the head of $savedq$ is timestamped less than d into the round, the only interesting property we need to check is property 2(d). Since this property held for some seq and $vstate$ in state x, we simply extend seq by appending $\langle m, vstate_p.clock\rangle$, which gives the second conjunct. Also, since the result of this action is exactly the same change in $vstate$ as with a vrcv(m) action, we have that the first conjunct must also hold.

- VSAlocal$(act)_p$: Since a precondition of this action is that $part$ is true, the only interesting property we need to check is property 2(d). Since this property held for some seq and the $vstate$ in state x, we keep the same seq, which preserves the second conjunct. For the first, notice that since the result of this action is exactly the same change in $vstate$ as with a locally controlled action, and a vcast message will be added to $outq$, we have that the first conjunct must also hold.

- resetRound$_p$: Since a result of this action is that $vstate_p$ is set to $\perp$ and $part$ is set to false, property 2 will trivially hold.

For the second case of the legal set definition, we now consider any closed trajectory τ such that $x = \tau.fstate$. Let x' be $\tau.lstate$. We must show that $x' \in L^3_{VEmu[alg]}$. The interesting properties to check are properties 2(c) and 2(d). Property 2(c) holds because $TOBDelay$ stopping conditions force the processing of messages in to_send queues, guaranteeing receipt of any vstate messages before time moves beyond $2d$ into a round. Also, the local copies of $vstate$ cannot be updated at time 0 until $part$ is updated through a prefer action, which line 56 guarantees. Property 2(d) holds because of stopping conditions on lines 74-75 and 82-83, restricting the order in which simulated actions are performed on virtual VSA state. ∎

11.3.2 Simulation relation

Here we show that the implementation started in set $L^3_{VEmu[alg]}$ implements the $VLayer$ started in a reachable state of $RW\|VW\|Vbcast$. We do this by first describing a simulation relation $\mathcal{R}_{Emu[alg]}$ for each $alg \in VAlgs$ from our implementation of the VSA layer to the VSA layer. We prove that $\mathcal{R}_{Emu[alg]}$ is a simulation relation in Lemma 11.10, and then conclude that $VEmu[alg]$ implements the VSA layer (Theorem 11.11). In other words, we conclude that the traces of our implementation are traces of the VSA layer. We then show in Lemma 11.14 that for each state in $L^3_{VEmu[alg]}$ there exists some state of *VLayer[alg]* where $RW\|VW\|Vbcast$ is in a reachable state that is related to it under the simulation relation. We also show another result, that ties traces of the implementation to traces of a constrained set of execution fragments of the VSA layer (Lemma 11.13).

The definition is structured in the following way: Property 1 constrains the relation so that for $x\mathcal{R}_{Emu[alg]}y$ to hold, state x must be a state in the legal set $L^3_{VEmu[alg]}$. This constrains the simulation relation to only be concerned with implementation states which we will show are related to certain desirable states of $VLayer[alg]$ (see Lemma 11.14). Property 2 states some consistency properties of state y of the virtual layer. Property 3 relates the value of RW between the implementation and the specification. Property 4 constrains the value of $vbcastq$ in the specification based on messages sent in the implementation. Properties 5 and 6 relate the failure status and state of physical nodes in the implementation to the state of client nodes in the specification. Property 7 describes the failure status and state of the virtual nodes based on the state of the implementation. One of the other things to note in property 7 is the relationship between the failure status of a VSA and the state of the emulation in a region. Intuitively, a VSA is failed when there are no emulators in a region that will be able to continue or perform emulation of the VSA. The conditions describing exactly when a VSA for some region is failed is described in property 7(a).

Definition 11.9 *For each $alg \in VAlgs$, define $\mathcal{R}_{Emu[alg]}$ to be a relation between states x of $VEmu[alg]$ and states y of $VLayer[alg]$ such that $x\mathcal{R}_{Emu[alg]}y$ if each of the following holds:*

1. $x \in L^3_{VEmu[alg]}$.

This says that state x must be a state in the legal set $L^3_{VEmu[alg]}$.

2. *State y satisfies the following properties:*

   (a) $y\lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}$.

   This says that the $RW\|VW\|Vbcast$ components of y are in $Inv_{RW\|VW\|Vbcast}$.

   (b) $\forall u \in U : [(\neg failed_u \Rightarrow clock_u = RW.now) \wedge last(u) \geq \max(\{t \in \mathbb{R}^{\geq 0} \mid \exists l \in R : \exists p \in P : \langle l, t\rangle \in updates(p)\})]$.

   This says that any non-failed VSA has a $clock$ equal to the real-time and that VW has updated each region with a time *action no longer ago than the last* GPSupdate.

   (c) $\forall u \in U : \neg failed_u : \forall \langle m, t\rangle \in to_send_u :$

   $$[now \mod t_{slice} > d \Rightarrow e + t - rtimer_u \geq d - now \mod t_{slice} + t_{slice}]$$
   $$\wedge\, [now \mod t_{slice} \leq d \Rightarrow e + t - rtimer_u \geq d - now \mod t_{slice}].$$

   This says that at any nonfailed VSA, the oldest message in its $VBDelay$ buffer is one that will not be older than e by the next time a round is d old.

3. $x(RW) = y(RW)$.

   This says that the RW component in both states is the same.

4. *Let $\langle\langle \mathsf{vmsg}, b^x_1, m^x_1\rangle, u^x_1, p_1, t^x_1\rangle, \cdots, \langle\langle \mathsf{vmsg}, b^x_n, m^x_n\rangle, u^x_n, p_n, t^x_n\rangle$ be the subsequence of $x(oldsent)x(sent)$ of* vmsg *messages where $t^x_i \geq now - d$. Let $\langle m^y_1, u^y_1, t^y_1, P'_1\rangle, \cdots, \langle m^y_m, u^y_m, t^y_m, P'_m\rangle$ be the subsequence of $y(vbcastq)$ such that $t^y_i \geq now - d$. Then there exists a bijection between elements of the two sequences such that for any two related tuples $\langle\langle \mathsf{vmsg}, b^x_i, m^x_i\rangle, u^x_i, p_i, t^x_i\rangle$ and $\langle m^y_j, u^y_j, t^y_j, P'_j\rangle$:*

   (a) $m^x_i = m^y_j \wedge u^x_i = u^y_j \wedge t^x_i = t^y_j$.

   This says that the related tuples match with respect to the message sent, the region they were sent from, and the time they were sent.

   (b) $\forall u \in U : u \notin P'_j \Leftrightarrow [i \leq n - |x(sent)| \vee (i = 1 + n - |x(sent)| \wedge |procs| < |P|)]$.

   This says that a region is not in the set of "to-be-processed" ids in $Vbcast$ exactly when the $TObcast$ tuple it is associated with is either in $x(oldsent)$ or is the head of $x(sent)$ and the message was processed for some process.

(c) $\forall p \in P : p \notin P'_j \Leftrightarrow$

$[([i \leq n - |x(sent)| \vee (i = 1 + n - |x(sent)| \wedge p \notin x(procs))] \wedge [failed_p \vee \langle\langle \mathsf{vmsg}, b_i^x, m_i^x\rangle, u_i^x\rangle \notin x(to_rcv_p)]]) \vee (t_i^x \neq now \wedge \neg regSpan(p, u_i^x, t_i^x))]$.

This says that a client id is not in the set of "to-be-processed" ids in $Vbcast$ exactly when either (a) the process fails the $regSpan$ test and the timestamp for the message is not now or (b) the $TObcast$ tuple it is associated with is either in $x(oldsent)$ or is the head of $x(sent)$ and p was processed, and either the client is failed or has already processed the message tuple from its $TOBFilter$ queue.

5. $\forall p \in P : x(failed_p) = y(failed_p)$.

   This says that the fail status matches between the states for each client.

6. $\forall p \in P : \neg failed_p$:

   (a) $x(updated_p) = y(updated_p) \wedge x(CE[alg]_p) = y(alg(p))$.

   This says that the $updated$ variable matches between the $VBDelay$ and $TOBDelay$ automata in the two states. It also says that the state of the client algorithm for the virtual layer being run is the same.

   (b) *Let* $\langle \mathsf{vmsg}, false, m_1\rangle, \cdots \langle \mathsf{vmsg}, false, m_n\rangle$ *be the subsequence of* $x(to_send_p^-)$ *of* $\langle \mathsf{vmsg}, false, m\rangle$ *tuples. Then* $m_1, \cdots, m_n = y(to_send_p^-)$.

   This says that to_send^- delay buffer in $VBDelay$ corresponds to the sequence of false-tagged vmsg *tuples in the to_send^- delay buffer in $TOBDelay$.*

   (c) *Let* $\langle \mathsf{vmsg}, false, m_1\rangle, \cdots \langle \mathsf{vmsg}, false, m_n\rangle$ *be the subsequence of* $x(to_send_p^+)$ *of* $\langle \mathsf{vmsg}, false, m\rangle$ *tuples. Then* $m_1, \cdots, m_n = y(to_send_p^+)$.

   This says that to_send^+ delay buffer in $VBDelay$ corresponds to the sequence of false-tagged vmsg *tuples in the to_send^+ delay buffer in $TOBDelay$.*

7. *For each $u \in U$: Let $\langle m_1, t_1\rangle, \cdots, \langle m_n, t_n\rangle$ be $y(to_send_u)$.*

   For each $p \in P$, let $\langle \mathsf{vmsg}, true, n_1^p\rangle, \cdots, \langle \mathsf{vmsg}, true, n_m^p\rangle$ be the subsequence of $x(to_send_p^-)\ x(to_send_p^+)$ of $\langle \mathsf{vmsg}, true, m\rangle$ tuples.

   (a) $y(failed_u) \Leftrightarrow$ *each of the following holds in state x:*

i. $\not\exists p \in P : [\neg failed_p \wedge ([u = reg^-(p) \wedge \exists \langle \mathsf{vmsg}, true, m\rangle \in x(to_send^-_p)] \vee [u = reg_p \wedge \exists \langle \mathsf{vmsg}, true, m\rangle \in x(to_send^+_p)])]$.

ii. $now \mod t_{slice} \geq d \Rightarrow$
$(\not\exists p \in P : [\neg failed_p \wedge clock_p \neq \bot \wedge reg_p = u \wedge (part_p \vee vstate_p \neq \bot)]$
$\vee (procVstate_u(d + t_{slice}\lfloor now/t_{slice}\rfloor, oldsent\ sent) = \bot \wedge \not\exists p \in P :$
$[\neg failed_p \wedge \exists \langle \mathsf{vstate}, u, vstate\rangle \in x(to_send^-_p)x(to_send^+_p)]))$.

iii. $\not\exists p \in P : [\neg failed_p \wedge clock_p \neq \bot \wedge reg_p = u \wedge (\neg part_p \vee \neg participated_p) \wedge vstate_p \neq \bot \wedge now \mod t_{slice} = 0]$.

iv. $\not\exists p \in P : [\neg failed_p \wedge clock_p \neq \bot \wedge reg_p = u \wedge part_p \wedge cand(u) = \langle p, pref(p)\rangle \wedge (participated_p \vee leader_p) \wedge (vstate_p \neq \bot \vee now \mod t_{slice} = d)]$.

This property describes exactly when a VSA is failed. These four properties are basically the negation of the preconditions that will be described in part(c)(ii)-(v). In part(c)(ii)-(v), we describe how to determine the state of non-failed VSAs based on a case analysis of the state of the implementation. For each region, property 7(a) makes the region be failed if it doesn't fit into any of the cases in part(c)(ii)-(v).

(b) $\forall p \in P : \neg failed_p : [(u = reg^-(p) \wedge \exists \langle \mathsf{vmsg}, true, m\rangle \in x(to_send^-_p)) \vee (u = reg_p \wedge \exists \langle \mathsf{vmsg}, true, m\rangle \in x(to_send^+_p))] \Rightarrow (n^p_1, \cdots, n^p_m) = (m_1, \cdots, m_m)$.
This says that true-tagged vmsg*s in a non-failed process's* $TOBDelay$ *buffers correspond to a prefix of the sequence of messages in the appropriate region's VSA* $VBDelay$ *buffer.*

(c) If there exists a $\langle m, now - d\rangle = head(sent)$ *and* $|x(procs)| < |P|$ *then let* $procSent = append(oldsent, head(sent))$*, else let* $procSent = oldsent$*.*
Then $\neg y(failed_u) \Rightarrow \exists vstate \in Q_{alg(u)} : \exists savedq \in (Msg \times \mathbb{R}^{\geq 0})^* : \exists outq \in Msg^*$ *such that each of the following holds:*

i. $lookAhead(u, vstate, y(vstate_u), savedq, outq, (m_1, \cdots, m_n))$.

ii. $\forall v \in Top : [now \mod t_{slice} = d$
$\wedge \exists p \in P : (\neg failed_p \wedge clock_p \neq \bot \wedge reg_p = u \wedge [part_p \vee vstate_p \neq \bot])$

$\wedge \exists p \in P : (\neg failed_p \wedge \exists \langle\langle \mathsf{vstate}, u, v\rangle\rangle \in x(to_send^-_p)x(to_send^+_p))] \Rightarrow$

- $[v \in Q_{alg(u)} \Rightarrow v\lceil(X_{alg(u)} - \{clock\}) = vstate\lceil(X_{alg(u)} - \{clock\})] \wedge$ $[v \notin Q_{alg(u)} \Rightarrow vstate = start_{alg(u)}(now)] \wedge vstate.clock = now.$
- $savedq = procVmsgs(u, t_{slice}\lfloor now/t_{slice}\rfloor, procSent).$
- $outq = (n^p_1, \cdots, n^p_m).$

This is the case where a round is d old and there exists a process in the region that is eligible to process an incoming vstate *message for the region in that round, and some alive process has a* vstate *message for the region in a $TOBDelay$ buffer. (In other words, the case where a* vstate *message for a region has been queued for sending and some process is currently eligible to receive it and continue the emulation.) Then $y(vstate_u)$ and $y(to_send_u)$ are consistent with the state that would result if the VSA at region u were to start at the state calculated from that* vstate *message, process messages that were sent starting in the beginning of the round and that would be received in the region, and add messages generated by* vcast *actions to the end of the true-tagged* vmsg*s in the process's $TOBDelay$ buffer.*

iii. $[now \mod t_{slice} \geq d$
$\wedge \exists p \in P : (\neg failed_p \wedge clock_p \neq \perp \wedge reg_p = u \wedge [part_p \vee vstate_p \neq \perp])$
$\wedge procVstate_u(d + t_{slice}\lfloor now/t_{slice}\rfloor, oldsent\ sent) \neq \perp] \Rightarrow$

- $vstate = procVstate_u(d + t_{slice}\lfloor now/t_{slice}\rfloor, oldsent\ sent).$
- $savedq = procVmsgs(u, t_{slice}\lfloor now/t_{slice}\rfloor, procSent).$
- $outq = \lambda.$

This is the case where there is a vstate *message for the region sent in this round and there exists a process in the region that is eligible to process it. (In other words, the case where a* vstate *message has successfully been transmitted and some process is currently eligible to receive it and continue the emulation.) Then $y(vstate_u)$ and $y(to_send_u)$ are consistent with the state that would result if the VSA at region u were to start at the state*

calculated from the vstate *message, process messages that were sent starting in the beginning of the round that would be received in the region, and add messages generated by* vcast *actions to an initially empty* to_send_u.

iv. $[now \mod t_{slice} = 0 \wedge \exists p \in P : (\neg failed_p \wedge clock_p \neq \bot \wedge reg_p = u \wedge vstate_p \neq \bot \wedge [\neg part_p \vee \neg participated_p])] \Rightarrow$

- $vstate = procVstate_u(now - t_{slice} + d, oldsent)$.
- $savedq = procVmsgs(u, now - t_{slice}, procSent)$.
- $outq = \lambda$.

This is the case where it is the beginning of the round and there is still some process in the region with a non-$\bot$ $vstate$ *that has not yet competed in the leader election service. (In other words, the case where a round has just begun and some emulator has yet to participate in the leader election service, meaning it is still possible that an emulator will continue the VSA emulation.) Then* $y(vstate_u)$ *and* $y(to_send_u)$ *are consistent with the state that would result if the VSA at region* u *were to start at the state calculated from the* vstate *message for the region in the last round, process messages that were sent starting in the beginning of the last round that would be received in the region, and add messages generated by* vcast *actions to an initially empty* to_send_u.

v. If $head(x(sent))$ *is equal to some* $\langle\langle \text{vmsg}, b, m\rangle, v, p', now - d\rangle$ *where* $v \in nbrs^+(u)$ *and* $|x(procs)| < |P|$*, then let* $\chi = \langle m, now\rangle$*, else let it be* λ*. Then* $\forall p \in P : [\neg failed_p \wedge clock_p \neq \bot \wedge reg_p = u \wedge part_p \wedge (participated_p \vee leader_p) \wedge cand(u) = \langle p, pref(p)\rangle \wedge (vstate_p \neq \bot \vee now \mod t_{slice} = d)] \Rightarrow$

- $[x(vstate_p) \neq \bot \wedge vstate = x(vstate_p)]$
  $\vee [x(vstate_p) = \bot \wedge vstate = start_{alg(u)}(now)]$.
- $savedq = append(savedq_p - \{\langle m', t'\rangle \mid t' < now - d\}, append(to_rcvVmsgs(to_rcv_p, now), \chi))$.
- $[x(vstate_p) \neq \bot \wedge outq = append((n_1^p, \cdots n_m^p), x(outq_p))]$

$\vee[x(vstate_p) = \bot \wedge outq = \lambda]$.

This says that if a non-failed process is in a region, has $part$ set, and is going to send a vstate *message (it won the leader competetion for the region and has not yet switched both its $participated$ and $leader$ bits off) then:*
(a) if its $vstate$ is not $\bot$ then $y(vstate_u)$ and $y(to_send_u)$ are consistent with the state that would result if region u's VSA were to start at the process's current $vstate$, process messages that were sent starting in the beginning of the round that would be received in the region, and add messages generated by vcast *actions to the end of the concatenation of the true-tagged* vmsg*s in the process's $TOBDelay$ buffer with the process's $outq$;*
(b) if its $vstate$ is $\bot$ (meaning the leader was not previously an emulator) and the round is d old then $y(vstate_u)$ and $y(to_send_u)$ is consistent with the state that would result if region u's VSA were to start at state $start_{alg(u)}(now)$, process messages that were sent starting in the beginning of the round that would be received in the region, and add messages generated by vcast *actions to an initially empty to_send_u.*

Now we show that $\mathcal{R}_{Emu[alg]}$ is a simulation relation from $VEmu[alg]$ to $VLayer[alg]$, both with some actions hidden.

Lemma 11.10 *Define H_{VEmu} be $\{\mathsf{tocast}(m)_p, \mathsf{torcv}(m)_p, \mathsf{leader}_p, \mathsf{prefer}_p \mid m \in Msg, p \in P\}$. Then for each $alg \in VAlgs$, $\mathcal{R}_{Emu[alg]}$ is a simulation relation from $\mathsf{ActHide}(H_{VEmu}, VEmu[alg])$ to $\mathsf{ActHide}(H_{VL}, VLayer[alg])$.*

Proof: By definition of a simulation relation we must show three things for all states of the two automata:

1. We must show that for any $x \in \Theta_{VEmu[alg]}$ there exists a state $y \in \Theta_{VLayer[alg]}$ such that $x\mathcal{R}_{Emu[alg]}y$. There is one unique initial non-*failed* and non-*loc* state for mobile nodes in both the first and the second automaton, and any values of *failed* and *loc* for each $p \in P$ is possible for either automaton. Have each VSA be failed. It is easy to check that $\mathcal{R}_{Emu[alg]}$ holds between any two such states.

2. Say that $x \in Q_{VEmu[alg]}$ and $y \in Q_{VLayer[alg]}$, and that $x\mathcal{R}_{Emu[alg]}y$. Then for any action $a \in A_{VEmu[alg]}$, if $\mathsf{ActHide}(H_{VEmu}, VEmu[alg])$ performs action a and the state changes from x to x', we must show that there exists a closed execution fragment β of $\mathsf{ActHide}(H_{VL}, VLayer[alg])$ with $\beta.fstate = y, trace(\beta) = trace(\wp(x)a\wp(x'))$, and $x'\mathcal{R}_{Emu[alg]}\beta.lstate$. The interesting thing to note in this portion of the proof is the failures of VSAs. There are several actions that can result in the failure of a VSA: a fail of a process in its region, a GPSupdate that indicates that a process has left its region, a tocast′ of a vmsg message for the region by a process not in the region, or a prefer′$(true)$ at a process that will not win the leader election competition. In each case, the VSA fails in the abstract level only if the resulting state in the implementation is one that satisfies property 7(a) of $\mathcal{R}_{VEmu[alg]}$, which describes the conditions corresponding to VSA failure.

   By Lemma 11.8, Property 1 of $\mathcal{R}_{Emu[alg]}$ holds in x'.

   For the other properties, we consider each action:

   - Internal action a of $CE[alg]_p$: Let β be $\wp(y)\ a\ \wp(y')$. It is trivial to see that $x'\mathcal{R}_{Emu[alg]}y$ and that the trace of both β and α are empty.

   - reset, participate$_p$, resetRound$_p$, prefer$_p$, leader′$(val)_p$, or leader$_p$: Let β be the point trajectory $\wp(y)$. It is easy to check that $x'\mathcal{R}_{Emu[alg]}y$ for each of these cases and that the trace of both β and α are empty.

   - fail$_p$: If the conditions in property 7(a) hold for reg_p in state x', then let β be $\wp(y)\ \mathsf{fail}_p\wp(y^*)\ \mathsf{fail}_{reg_p}\wp(y')$. Otherwise, let β be $\wp(y)\ \mathsf{fail}_{reg_p}\wp(y')$. It is trivial to see that the traces of α and β are the same in both cases. It is obvious that all properties of the simulation relation hold between states x' and y'.

   - restart$_p$: Let β be $\wp(y)\ \mathsf{restart}_p\ \wp(y')$. It is trivial to see that $x'\mathcal{R}_{Emu[alg]}y$ and that the traces of β and α are the same.

   - GPSupdate$(l,t)_p$: Let $u_1 \cdots u_{|u|}$ be some ordering of the region ids. Let $\langle n_1, j_1\rangle, \cdots, \langle n_k, j_k\rangle$ be an ordering of the indices n_i of tuples $\langle m_i, u_i, t_i, P_i'\rangle$ in $y(vbcastq)$ and process ids j_i such that $t_i \neq x(now)$,

$j_i \in P'_i$, and $\neg x(regSpan(j_i, u_i, t_i))$. If $x(reg_p) \neq x'(reg_p)$ and the properties in 7(a) hold for $x(reg_p)$ in state x', then let β be $\wp(y)$ $\mathsf{GPSupdate}(l,t)_p$ $\wp(y^*)$ $\mathsf{time}(t)_{u_1}$ $\wp(y_{u_1}) \cdots \mathsf{time}(t)_{u_{|U|}}$ $\wp(y_{u_{|U|}})$ $\mathsf{drop}(n_1, j_1)$ $\wp(y_1) \cdots \mathsf{drop}(n_k, j_k)$ $\wp(y_k)$ fail_{reg_p} $\wp(y')$. Otherwise, let β stop after $\wp(y_k)$. It is trivial to see that the traces of α and β are the same in both cases.

The only interesting properties to check are properties 4(c) and 7(a). For property 4(c), it is obvious that if the property held between state x and y, then it will also hold between x' and $y_k.lstate$ since the GPSupdate removes the associated message tuple for $TOBFilter$ or will drop the message in $TObcast$ when exactly d time has passed since it was sent.

For property 7(a), the GPSupdate only affects the property if the process has changed regions from some region u. If it has, then if the conditions in 7(a) hold in state x', the simulation relation implies that that $y'(failed_u)$ must be true. This is obviously the case after the addition of the fail event.

- tocast$(m)_p$: If there exists an m' such that $m = \langle \mathsf{vmsg}, false, m' \rangle$, then let β be $\wp(y)$ $\mathsf{vcast}(m')_p$ $\wp(y')$. It is obvious that the properties of the simulation relation hold between x' and y'. It is obvious that the traces of α and β are the same.

  If there is no such m', then let β be the point trajectory $\wp(y)$. It is obvious that the traces of α and β are the same. The only interesting properties to check are property 7(b) if the message was a true-tagged vmsg message or property 7(c)(ii) if the message was a vstate message.

  In the case of a true-tagged vmsg, we need to verify that the resulting $TOBDelay$ buffer of such messages corresponds to a prefix of the VSA's $VBDelay$ messages. This follows from the fact that property 7(c)(v) holds between state x and y, implying that when the head of $x(outq_p)$ is removed and decorated to sit at the end of the $TOBDelay$ buffers, the resulting $outq$ computed by the property for state x' is the same as in state x, implying the property

still holds.

In the case of a $\langle \mathsf{vstate}, u, q\rangle$ message, we look at two cases, where q is in $Q_{alg(u)}$ and where q is not. Notice that for this action to occur, it must be that $\neg failed_p, clock_p \neq \bot, reg_p = u$, and $leader_p$.

If q is in the set of states, then since state x and y satisfied property 7(c)(v) and a precondition for the action is that $outq_p = \lambda$ and no changes to $vstate_p$, $savedq_p$, to_rcv_p, or $outq_p$ are made by the current action, then the $lookAhead$ statement over the same arguments most hold between states x' and y'.

If q is not in the set of states, then since state x and y satisfied property 7(c)(v) and a precondition for the action is that $vstate = \bot$ and no changes to $vstate_p$, $savedq_p$, to_rcv_p, or $outq_p$ are made by the current action, then the $lookAhead$ statement over the same arguments most hold between states x' and y'.

- $\mathsf{tocast}'(m, f)_p$: If m is not a vmsg tuple then let β be the point trajectory $\wp(y)$. It is obvious that the traces of α and β are the same. The only interesting property to check in this case is property 7(c)(iii). In order for a tocast' of a vstate message for a region u to occur it must be that the message was in a $TOBDelay$ buffer, and by property 3(b) of $L^2_{VEmu[alg]}$ there can be no other vmsg or vstate messages after it in $TOBDelay$. This implies that the $x(procVmsgs(u, t_{slice}\lfloor now/t_{slice}\rfloor, procSent)) = x'(procVmsgs(u, t_{slice}\lfloor now/t_{slice}\rfloor, procSent))$, $x(n^p_1, \cdots, n^p_m) = \lambda$ and $x'(procVstate(d + t_{slice}\lfloor now/t_{slice}\rfloor, oldsent\ sent))$ is equal to the calculated $vstate$ in state x for property 2(c)(ii). All this implies that the result of the $lookAhead$ function between x' and y is still true.

  If there exists an $m' \in Msg$ such that $m = \langle \mathsf{vmsg}, false, m'\rangle$ then let β be $\wp(y)\ \mathsf{vcast}'(m', f)_p\ \wp(y')$. If there exists an $m' \in Msg$ such that $m = \langle \mathsf{vmsg}, true, m'\rangle$ then we have four cases. If f is true and state x' satisfies the conditions in property 7(a) then let β be $\wp(y)\ \mathsf{vcast}'(m', true)_{reg_p}\ \wp(y^*)\mathsf{fail}_{reg_p}\wp(y')$. If f is true and state x' does not satisfy the conditions in property 7(a) then let β be

$\wp(y)$ $\mathsf{vcast}'(m', true)_{reg_p}$ $\wp(y')$. The remaining two cases are for when f is false, and where we replace reg_p with $reg^-(p)$. The most interesting property to check is property 7(c)(v). However, since this same property held between x and y and the only difference is that the tuple in $TOBDelay$ that is associated with $x(n_1^p)$ is removed both from $TOBDelay$ and the VSA's $VBDelay$, preserving the property between state x' and y'.

- $\mathsf{torcv}(m)_p$: If there exists an $m' \in Msg$ and a $b \in Bool$ such that $m = \langle \mathsf{vmsg}, b, m' \rangle$ then let β be $\wp(y)$ $\mathsf{vrcv}(m')_p$ $\wp(y')$. Otherwise, let β be the point trajectory $\wp(y)$. It is obvious that the properties of the simulation relation hold between x' and the either of the final states of β. It is obvious that the traces of α and β are the same.

- $\mathsf{torcv}'(m, u)_p$, $\mathsf{drop}(p)$: If there exists a $\langle \langle \mathsf{vmsg}, b, m' \rangle, u, q, x(now) - d \rangle = head(x(sent))$ and $|x(procs)| = |P|$, then let n be the index of the element of $y(vbcastq)$ associated with the tuple at the head of $x(sent)$, let $u_1, \cdots u_k$ be an ordering of the elements in $nbrs^+(u)$ and let $u_{k+1}, \cdots u_{k+l}$ be an ordering of the elements in $U - nbrs^+(u)$. Then let β be $\wp(y)$ $\mathsf{vrcv}(m')_{u_1}$ $\beta_1 \cdots \mathsf{vrcv}(m')_{u_k}$ $\beta_k \mathsf{drop}(n, u_{k+1})$ $\beta_{k+1} \cdots \mathsf{drop}(n, u_{k+l})$ β_{k+l}, where for each $i \in [1, k]$, β_i reflects the maximal local computation of the VSA in region u_i after receipt of the message. Otherwise, let β be the point trajectory $\wp(y)$. It is obvious that the traces of α and β are the same in either case.

  In the case where the message is a vmsg message, the interesting properties to check are properties 4(b) and 7(c)(ii-iv). Property 4(b) holds since all region ids are removed from the associated $vbcastq$ message tuples's P' variable exactly when $procs$ in $TObcast$ goes from being full to having processed a member.

  The portions of property 7(c) of interest will hold between x' and $\beta.lstate$ because the only difference in the computed $vstate, savedq$, and $outq$ for the $lookAhead$ function is in the possible extension of the $savedq$ from state x by the appropriate received vmsg. The message is added to the computed $savedq$ for the implementation exactly when it is processed by the VSA in reg_p, imply-

ing that if property 7(c) held between state x and y then it also holds between x' and $\beta.lstate$.

- $\mathsf{prefer}'(val)_p$: If val is true and the properties of 7(a) hold in state x' then let β be $\wp(y)\ \mathsf{fail}_{reg_p}\wp(y')$. Otherwise let β be the point trajectory $\wp(y)$. It is obvious that the traces of α and β are the same. It is obvious that the simulation relation holds between state x' and the final state of β.
- $\mathsf{VSArcv}(m)_p$: Let β be the point trajectory $\wp(y)$. It is obvious that the traces of α and β are the same. The only interesting property of the simulation relation to check is property 7(c)(v). We know that the only difference in the calculated $vstate$, $savedq$, and $outq$ is that the $vstate$ is the result of receiving mesasge m and performing local computations until no more are possible at the VSA, and removing the first element of the calculated $savedq$. Inspection of the function $lookAhead$ reveals that since property 7(c)(v) held between state x and y, it must hold between state x' and y.
- $\mathsf{VSAlocal}(act)_p$: Let β be the point trajectory $\wp(y)$. It is obvious that the traces of α and β are the same. The only interesting property of the simulation relation to check is property 7(c)(v), but the reasoning is similar to that of VSArcv.

3. Say that $x \in Q_{VEmu[alg]}, y \in Q_{VLayer[alg]}$, and $x\mathcal{R}_{Emu[alg]}y$. Let α be an execution fragment of $\mathsf{ActHide}(H_{VEmu}, VEmu[alg])$ consisting of one closed trajectory, with $\alpha.fstate = x$.

   We must show that there is a closed execution fragment β of $\mathsf{ActHide}(H_{VL}, VLayer[alg])$ with $\beta.fstate = y, trace(\beta) = trace(\alpha)$, and $\alpha.lstate\mathcal{R}_{Emu[alg]}\beta.lstate$. The interesting thing to note in this portion of the proof is the VSA $\mathsf{restart}$s in the abstract level. They occur when rounds are d old and certain conditions are satisfied. They are added to executions of the abstract layer based on trajectories of the implementation that straddle the point where a round is d old.

   If there exists a time t such that $x(now) \leq t \leq x'(now)$ and $t \mod t_{slice} = d$ then let $u_1, \cdots, u_k$ be some ordering of the region ids for which for each $i \in$

$[1, k]$ there exists a process p_i such that $\neg failed_{p_i}$, $clock_{p_i} \neq \bot$, $reg_{p_i} = u_i$, $participated_{p_i} \vee leader_{p_i}$, and $cand(u_i) = \langle p_i, pref(p_i) \rangle$. If such a t exists then let β be $\beta_0 \mathsf{restart}_{u_1} \wp(y_1') \mathsf{time}(t)_{u_1} \wp(y_1) \cdots \mathsf{restart}_{u_k} \wp(y_k') \mathsf{timed}(t)_{u_k}, y_k$, where $\beta_0.lstate(now) = t$. Otherwise, let β consist just of y_k. Both β_0 and y_k are required to provide maximal ordered local computation at the VSAs (the actions performed at each VSA are the ones as indicated by the **next** function for the VSA and there exist no locally controlled actions for any VSA in state $y_k.lstate$).

Finally, if $x(now) < x'(now)$ then y_k contains a $\mathsf{drop}(n, j)$ action at time $y_k.lstate(now)$ for each tuple $\langle m, u, t, P' \rangle = vbcastq[n]$ and process id j such that $j \in P'$ and $\neg x'(regSpan(j, u, t))$. This ensures that for property 7(c) the calculated $savedq$ for each subpart corresponds to the messages that have been received by the VSA.

Since each of the actions possibly added above are internal to the abstract system, it is apparent that the traces of α and β are the same. To check that the simulation relation holds between state x and state $y_k.lstate$, we note that the most interesting properties to check are properties 7(a) and 7(c).

For property 7(a), notice that by construction at d into a round, if there is a process that will perform send a vstate message, then the VSA of the process's region is alive, tacking the fourth part of property 7(a). Since the VSA cannot fail until a discrete action occurs to change a variable referenced in property 7(a)(iv), we know that property 7(a) holds between states x' and $y_k.lstate$.

For property 7(c), notice that in state y_k, each alive VSA performs an ordered sequence of locally controlled events until no more are enabled. Since x and y are related and each VSA has simply developed its state forward from y in a manner consistent with the $lookAhead$ function, it is obvious that property 7(c) holds between states x' and $y_k.lstate$.

■

The following theorem concludes that for each $alg \in VAlgs$, our implementation of the VSA layer implements $VLayer[alg]$, after the hiding of several actions.

Theorem 11.11 *For each $alg \in VAlgs$,* $\mathsf{ActHide}(H_{VEmu}, VEmu[alg]) \leq \mathsf{ActHide}(H_{VL}, VLayer[alg])$.

Proof: This follows directly from the previous lemma and Corollary 2.23. ■

One useful corollary of this result and the construction of the matching execution in the proof of the simulation relation $\mathcal{R}_{Emu[alg]}$ is that fragments of $\mathsf{ActHide}(H_{VEmu}, VEmu[alg])$ starting in states in $L^3_{VEmu[alg]}$ correspond to fragments of $\mathsf{ActHide}(H_{VL}, VLayer[alg])$ started in states in $\{y \in Q_{VLayer[alg]} \mid y\lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}\}$ that are also in the set S defined below. S describes execution fragments of the virtual layer that satisfy certain properties with respect to the failure status of a VSA. In particular, it describes when a fail or restart for a VSA is allowed to occur, and when a restart of a VSA is guaranteed to occur.

Definition 11.12 *Define S to be the function that maps, for each $alg \in VAlgs$, $VLNodes[alg]$ to the suffix-closed set of execution fragments α of $VLayer[alg]$ where for each $u \in U$:*

1. *If a* $\mathsf{restart}_u$ *occurs in α at time t then $t \mod t_{slice} = d$ and no* fail *or* GPSupdate *actions occur in α at time t before* $\mathsf{restart}_u$.
   This says that a VSA can only restart exactly d into a timeslice, before any fail *or* GPSupdate *actions have occurred.*

2. *For each $t \in \mathbb{R}^{\geq 0}$ such that $t \mod t_{slice} = 0$ and $\alpha.fstate(RW.now) < t \leq \alpha.lstate(RW.now)$ we define the following:*

   - *For each state x in α and process id $j \in P$, we define $aware(u, j, x)$ to be true exactly when $\neg x(failed_j)$, $x(VBDelay_j.updated) =$ true, and $x(reg(j)) = u$. (This is a way of saying that process j is alive and knows it is in region u in state x.)*
   - *Define J_u to be the set of process ids j such that there exists a state x in α with $x(RW.now) = t$ such that $aware(u, j, x)$ is true.*

- *For each* $j \in J_u$*, define* x_j *to be the first state in* α *such that* $x(RW.now) = t$ *and* $aware(u, j, x)$ *is true.*

Then

(a) *Let* x *be the first state in* α *such that* $RW.now = t$*. If* $\neg x(failed_u)$*, then there exists some process* j *such that* $aware(u, j, x)$*.*

This says that at the beginning of a new timeslice, if a VSA is not failed then it must be the case that there is some alive process that knows it is in the VSA's region.

(b) *If* $\alpha.lstate(RW.now) \geq t + d$*,* $|J_u| > 0$ *and* $aware(u, j, x')$ *is true for each* $j \in J_u$ *and each state* x' *in* α *starting from state* x_j *and ending with the first state such that* $RW.now = t+d$*, then there exists a* $\mathsf{restart}_u$ *action in* α *at time* $t + d$*.*

This says that if the set of processes alive and aware they are in region u *at the beginning of a timeslice is nonempty and none of the processes in the set fail or leave the region before* d *into the timeslice, then a* $\mathsf{restart}_u$ *action will occur for the region's VSA.*

(c) *If there exists a* fail_u *action at state* x *in* α *at time* t' *such that* $t' - t' \mod t_{slice} = t$ *and* $\neg x(failed_u)$*, then there exists a* $j \in J_u$ *and a state* x'_j *in* α *after* x_j *and no later than* x *such that* $\neg aware(u, j, x'_j)$*.*

This says that if an alive VSA fails, then there must have been some process that was alive and aware it was in the VSA's region at the beginning of the timeslice but that has failed or left the region in the meantime.

(d) *If there exists a* fail_u *action at state* x *in* α *at time* t' *such that* $t' \in (t+d, t+t_{slice})$ *and* $\neg x(failed_u)$*, then for each* $j \in J_u$*, there exists a state* x'_j *in* α *after* x_j *and before the* fail_u *such that* $\neg aware(u, j, x'_j)$*.*

This says that if an alive VSA fails when the round is more than d *old, then it must be the case that each process that was alive and aware it was in the VSA's region at the beginning of the timeslice has failed or left the region in the meantime.*

As mentioned before Definition 11.12, the following result says that for any execution fragment of $VEmu[alg]$ starting in a state in $L^3_{VEmu[alg]}$ and for any state in $\{y \in Q_{VLayer[alg]} \mid y\lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}\}$ such that the two states are related, there is some fragment of $VLayer[alg]$ that not only has the same trace but also has the same RW and $Fail$-related projections. In addition, that fragment is a fragment allowed by S.

Lemma 11.13 *Let alg be in $VAlgs$ and α be in $frags^{L^3_{VEmu[alg]}}_{\mathsf{ActHide}(H_{VEmu},VEmu[alg])}$. Let y be a state in $\{y \in Q_{VLayer[alg]} \mid y\lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}\}$ such that $\alpha.fstate\mathcal{R}_{Emu[alg]}y$. Then there exists an α' in $frags_{\mathsf{ActHide}(H_{VL},VLayer[alg])}$ such that:*

1. *$\alpha'.fstate = y$.*

2. *$trace(\alpha) = trace(\alpha')$.*

3. *If α is a closed execution fragment, then $\alpha.lstate\mathcal{R}_{Emu[alg]}\alpha'.lstate$.*

4. *$\alpha\lceil(A_{RW}, V_{RW}) = \alpha'\lceil(A_{RW}, V_{RW})$.*

5. *For each $p \in P$, $\alpha\lceil(A_{Fail(CE[alg]_p)}, V_{Fail(CE[alg]_p)}) = \alpha'\lceil(A_{Fail(alg(p))}, V_{Fail(alg(p))})$.*

6. *$\alpha' \in S[VLNodes[alg]]$.*

The first three properties of the lemma follow from the fact that $\mathcal{R}_{Emu[alg]}$ is a simulation relation, while the fourth and fifth follow from the construction of the matching execution of $VLayer[alg]$ in the proof that $\mathcal{R}_{Emu[alg]}$ is a simulation relation in Lemma 11.10, which preserves the actions and variables of RW and each of the processes' $Fail$-transform variables and actions. The only interesting property to show is property 6, and in particular, property 2(b) of the definition of S. This can be shown by noting that this property follows immediately from use of the leader election service, which guarantees that in the circumstance described in property 2(b), a leader output will be ready to be performed exactly d into the timeslice (see Property 1 of Section 10.1.3), and the construction of the $VLayer$ execution will add a restart action for the region at that time.

The following result ties the legal states $L^3_{VEmu[alg]}$ to certain states of $VLayer[alg]$.

Lemma 11.14 *For any* $alg \in VAlgs$ *and state* $x \in L^3_{VEmu[alg]}$*, there exists a state* $y \in Q_{VLayer[alg]}$ *such that* $y \lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}$ *and* $x \mathcal{R}_{Emu[alg]} y$*.*

Proof: We prove this lemma by showing how, given a state $x \in L^3_{VEmu[alg]}$, we can construct a state y of *VLayer[alg]* such that $y \lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}$ and $x \mathcal{R}_{Emu[alg]} y$. This construction is relatively trivial given the manner in which $\mathcal{R}_{Emu[alg]}$ is defined; the relation mostly describes what the state y will be. The only components in state y for which the relation does not dictate the state values exactly are as follows:

- $VW.last$: We require that for each $u \in U$, $last(u)$ is no older than the most recent of the GPSupdates that occurred or the last time that a round was d old.
- $Vbcast.vbcastq$: Property 4 of the simulation relation constrains the messages sent no more than d before $x(now)$. We have $vbcastq$ contain no messages before that time. This obviously satisfies property 4.

It is not difficult to check that such a state y is one where x is related to y and $y \lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}$.

We conclude that for any state x in $L^3_{VEmu[alg]}$, there is some state y of *VLayer[alg]* such that $x \mathcal{R}_{Emu[alg]} y$ and $y \lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}$. ■

Lemma 11.14 and Theorem 11.11 immediately imply:

Lemma 11.15 $Start(\mathsf{ActHide}(H_{VEmu}, VEmu[alg]), L^3_{VEmu[alg]}) \leq Start(\mathsf{ActHide}(H_{VL}, VLayer[alg]), \{x \in Q_{VLayer[alg]} \mid x \lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}\})$.

11.3.3 Self-stabilization

We've seen that $L^3_{VEmu[alg]}$ is a legal set for the emulation, and that each state in $L^3_{VEmu[alg]}$ is related to some desirable state of *VLayer[alg]*. Here we show that for any $alg \in VAlgs$, $VEmu[alg]$ started in any state x such that the $LeadSpec$ component states are in $Inv_{LeadSpec}$ and the $TOBSpec$ component states are in $Inv_{TOBSpec}$ stabilizes to execution

fragments whose states are in $L^3_{VEmu[alg]}$ (Lemma 11.19). This is done in phases, corresponding to each legal set: we show that we stabilize to each set from the one before it. After we show this stabilization result, we conclude that after an execution of *VEmu[alg]* has stabilized, the trace fragment from the point of stabilization is a trace of a fragment of $VLayer[alg]$, with certain actions hidden and with the centralized components started in a somewhat consistent state (Theorem 11.21).

The first lemma describes the first phase of stabilization, for legal set $L^1_{VEmu[alg]}$. Recall that this legal set is one that is arrived at after GPSupdate actions have occurred at each process. It is easy to check that $frags^{\{x\in Q_{VEmu[alg]}|x\lceil X_{LeadSpec}\in Inv_{LeadSpec}\wedge x\lceil X_{TOBspec}\in Inv_{TOBspec}\}}_{VEmu[alg]}$ stabilizes to $frags^{L^1_{VEmu[alg]}}_{VEmu[alg]}$ in time t^1_{vestab}, where t^1_{vestab} is any t such that $t > \epsilon_{sample}$. (To see this stabilization result, just consider the moment after each node has a received a GPSupdate, which takes at most ϵ_{sample} time to happen.)

Lemma 11.16 *Let alg be in $VAlgs$ and t^1_{vestab} be any t such that $t > \epsilon_{sample}$. $frags^{\{x\in Q_{VEmu[alg]}|x\lceil X_{LeadSpec}\in Inv_{LeadSpec}\wedge x\lceil X_{TOBspec}\in Inv_{TOBspec}\}}_{VEmu[alg]}$ stabilizes to $frags^{L^1_{VEmu[alg]}}_{VEmu[alg]}$ in time t^1_{vestab}.*

We now show that execution fragments starting in $L^1_{VEmu[alg]}$ stabilize to execution fragments starting in $L^2_{VEmu[alg]}$. Recall that $L^2_{VEmu[alg]}$ describes states that satisfy certain properties with respect to the relationship between the leader election service state and the emulation algorithm state. The proof of this lemma takes advantage of the fact that when a round is more than d old, a large number of the properties of $L^2_{VEmu[alg]}$ are trivially satisfied.

Lemma 11.17 *Let alg be in $VAlgs$ and t^2_{vestab} be any t such that $t > d$. Then $frags^{L^1_{VEmu[alg]}}_{VEmu[alg]}$ stabilizes to $frags^{L^2_{VEmu[alg]}}_{VEmu[alg]}$ in time t^2_{vestab}.*

Proof: By Lemma 3.21, we just need to show that for any length-t^2_{vestab} prefix α of an element of $frags^{L^1_{VEmu[alg]}}_{VEmu[alg]}$, $\alpha.lstate$ is in $L^2_{VEmu[alg]}$. We examine each property of $L^2_{VEmu[alg]}$.

By Lemma 11.4, since the first state of α is in $L^1_{VEmu[alg]}$, we know that property 1 of $L^2_{VEmu[alg]}$ holds in each state of α. That property 2(a) and the second conjunct of property

2(b) hold after d time passes is immediately obvious. It is also easy to check that these properties do not affect the other properties, and so can be stabilized independently.

For the remaining properties, consider a state x in α such that $x(now) \mod t_{slice} > d$. Such a state must exist in α since α is of length $t^2_{vestab} > d$. We just need to show that the remaining properties hold in state x and we are done. The crux of this part of the proof is that when $x(now) \mod t_{slice} > d$, the properties of $L^1_{VEmu[alg]}$ make many of the remaining cases trivially satisfied. Properties 2(d), 4, and 5 trivially hold in state x.

For property 2(b)'s first conjunction, if $pref_p$ is true at a non-failed process then property 6(b) of $L^1_{VEmu[alg]}$ implies that either $x(now) \mod t_{slice} = 0$ or $participated_p$ is true. Since we are assuming that $x(now) \mod t_{slice} > d$, then $participated_p$ is true, which by property 7(a) of $Inv_{LeadSpec}$ implies that $x(now) \mod t_{slice} \leq d$. Hence, we know that property 2(b)'s first conjunct is trivially true. Property 7(a) of $Inv_{Leadspec}$ also implies that property 2(c) trivially holds.

For property 2(e), notice that by property 2(b) of $L^1_{VEmu[alg]}$, if $leader_p$ is true then $x(now) \mod t_{slice} = d$, so property 2(e) also trivially holds in state x.

Finally, for property 3, notice that true-tagged vmsg messages and vstate messages are only sent by a process for which $leader_p$ is true. As just established, this does not hold for any process in state x. Any such messages that were previously in the queue will be removed before time passes.

We conclude that $\alpha.lstate$ is in $L^2_{VEmu[alg]}$. ■

We now show that execution fragments starting in $L^2_{VEmu[alg]}$ stabilize to execution fragments starting in the final set of legal states, $L^3_{VEmu[alg]}$. Recall that $L^3_{VEmu[alg]}$ describes states that can be related to certain states of the VSA layer. The proof of this lemma takes advantage of the fact that when a round is more than 0 old, but less than d old, many of the properties of $L^3_{VEmu[alg]}$ are satisfied.

Lemma 11.18 *Let alg be in $VAlgs$ and t^3_{vestab} be any t such that $t > t_{slice} - d$. Then $frags^{L^2_{VEmu[alg]}}_{VEmu[alg]}$ stabilizes to $frags^{L^3_{VEmu[alg]}}_{VEmu[alg]}$ in time t^3_{vestab}.*

Proof: By Lemma 3.21, we just need to show that for any length-t^3_{vestab} prefix α of an element of $frags^{L^2_{VEmu[alg]}}_{VEmu[alg]}$, $\alpha.lstate$ is in $L^3_{VEmu[alg]}$. We examine each property of

$L^3_{VEmu[alg]}$.

By Lemma 11.6, since the first state of α is in $L^2_{VEmu[alg]}$, we know that property 1 of $L^3_{VEmu[alg]}$ holds in each state of α.

For the remaining properties, consider a state x in α such that $x(now) \mod t_{slice} \in (0, d)$. Such a state must exist in α since $t^3_{vestab} > t_{slice} - d$. We just need to show that all the properties hold in state x and we are done. Properties 2(a) and 2(d) trivially hold in x.

For property 2(b), notice that properties 4 and 5 of $Inv_{TOBspec}$ imply that no such vstate message could exist, since the timestamp on the message would be from the future. Hence, property 2(b) is trivially satisfied.

For property 2(c), property 3(b) of $L^1_{VEmu[alg]}$ implies $part_p$ is true, making property 2(c) trivially satisfied.

We conclude that $\alpha.lstate$ is in $L^3_{VEmu[alg]}$. ■

We've shown that executions of $VEmu[alg]$ started in a consistent leader election and totally ordered broadcast state stabilize to executions of $VEmu[alg]$ started in $L^1_{VEmu[alg]}$, which stabilize to executions started in $L^2_{VEmu[alg]}$, which in turn stabilize to executions started in $L^3_{VEmu[alg]}$. Now we can combine these stabilization results to conclude that executions of $VEmu[alg]$ started in consistent leader election and totally ordered broadcast states stabilize to executions of $VEmu[alg]$ started in $L^3_{VEmu[alg]}$ in time t_{vestab}, where t_{vestab} is any t such that $t > \epsilon_{sample} + t_{slice}$.

Lemma 11.19 *Let alg be an element of $VAlgs$, and t_{vestab} be any t such that $t > \epsilon_{sample} + t_{slice}$.*
Then $frags_{VEmu[alg]}^{\{x \in Q_{VEmu[alg]} | x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x \lceil X_{TOBspec} \in Inv_{TOBspec}\}}$ stabilizes to $frags_{VEmu[alg]}^{L^3_{VEmu[alg]}}$ in time t_{vestab}.

Proof: This result follows as a direct application of Lemma 3.7 to Lemmas 11.16, 11.17, and 11.18. Let $t^1_{vestab} = \epsilon_{sample} + (t_{vestab} - t_{slice} - \epsilon_{sample})/3$, $t^2_{vestab} = d + (t_{vestab} - t_{slice} - \epsilon_{sample})/3$, and $t^3_{vestab} = t_{slice} - d + (t_{vestab} - t_{slice} - \epsilon_{sample})/3$.

Let B_0 be $frags_{VEmu[alg]}^{\{x \in Q_{VEmu[alg]} | x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x \lceil X_{TOBspec} \in Inv_{TOBspec}\}}$, B_1 be $frags_{VEmu[alg]}^{L^1_{VEmu[alg]}}$, B_2 be $frags_{VEmu[alg]}^{L^2_{VEmu[alg]}}$, and B_3 be $frags_{VEmu[alg]}^{L^3_{VEmu[alg]}}$ in Lemma 3.7. Let t_1 be t^1_{vestab}, t_2 be t^2_{vestab}, and t_3 be t^3_{vestab} in Lemma 3.7. Then by Lemma 3.7 and Lemmas

11.16-11.18, we have that $frags_{VEmu[alg]}^{\{x \in Q_{VEmu[alg]} | x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x\lceil X_{TOBspec} \in Inv_{TOBspec}\}}$ stabilizes in time $t_{vestab}^1 + t_{vestab}^2 + t_{vestab}^3$ to $frags_{VEmu[alg]}^{L^3_{VEmu[alg]}}$.

Since $t_{vestab} = t_{vestab}^1 + t_{vestab}^2 + t_{vestab}^3$, we conclude that $frags_{VEmu[alg]}^{\{x \in Q_{VEmu[alg]} | x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x\lceil X_{TOBspec} \in Inv_{TOBspec}\}}$ stabilizes to $frags_{VEmu[alg]}^{L^3_{VEmu[alg]}}$ in time t_{vestab}. ■

We can now conclude from Lemma 11.19 and Lemma 11.15 that an execution of $VEmu[alg]$ eventually reaches a point such that the trace of the execution from that point on is the same as the trace of an execution fragment of $VLayer[alg]$ starting an arbitrary state of its nodes, both after some action hiding.

Theorem 11.20 *Let alg be an element of $VAlgs$, and t_{vestab} be any t such that $t > \epsilon_{sample} + t_{slice}$.*
Then $tracefrags_{\mathsf{ActHide}(H_{VEmu}, VEmu[alg])}^{\{x \in Q_{VEmu[alg]} | x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x\lceil X_{TOBspec} \in Inv_{TOBspec}\}}$ stabilizes in time t_{vestab} to $traces_{\mathsf{ActHide}(H_{VL}, U(VLNodes[alg]) \| R(RW \| VW \| Vbcast))}$.

As promised at the beginning of Section 11.3.3, we can actually conclude even more than the above result; we can conclude that an execution of $VEmu[alg]$ eventually reaches a point such that the trace of the execution from that point on is the same as the constrained trace of certain execution fragments of $VLayer[alg]$, both after some action hiding.

Theorem 11.21 *Let alg be an element of $VAlgs$, and t_{vestab} be any t such that $t > \epsilon_{sample} + t_{slice}$.*
Then $tracefrags_{\mathsf{ActHide}(H_{VEmu}, VEmu[alg])}^{\{x \in Q_{VEmu[alg]} | x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x\lceil X_{TOBspec} \in Inv_{TOBspec}\}}$ stabilizes in time t_{vestab} to $\{trace(\alpha) \mid \alpha \in S[VLNodes[alg]] \cap execs_{\mathsf{ActHide}(H_{VL}, U(VLNodes[alg]) \| R(RW \| VW \| Vbcast))}\}$.

Proof: By Theorem 11.19, we know that $frags_{VEmu[alg]}^{\{x \in Q_{VEmu[alg]} | x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x\lceil X_{TOBspec} \in Inv_{TOBspec}\}}$ stabilizes in time t_{vestab} to $frags_{VEmu[alg]}^{L^3_{VEmu[alg]}}$. By Lemmas 3.5 and 3.10, this implies that $traces_{Start(VEmu[alg], \{x \in Q_{VEmu[alg]} | x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x\lceil X_{TOBspec} \in Inv_{TOBspec}\})}$ stabilizes in time t_{vestab} to $tracefrags_{VEmu[alg]}^{L^3_{VEmu[alg]}}$.

Since Lemmas 11.13 and 11.15 imply that $tracefrags_{\mathsf{ActHide}(H_{VEmu}, VEmu[alg])}^{L^3_{VEmu[alg]}} \subseteq \{trace(\alpha) \mid \alpha \in S[VLNodes[alg]] \cap frags_{\mathsf{ActHide}(H_{VL}, U(VLNodes[alg]) \| R(RW \| VW \| Vbcast))}\}$,

we conclude that the traces of $\mathsf{ActHide}(H_{VEmu}, Start(VEmu[alg], \{x \in Q_{VEmu[alg]} | x \lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x \lceil X_{TOBspec} \in Inv_{TOBspec}\}))$ stabilize in time t_{vestab} to $\{trace(\alpha) \mid \alpha \in S[VLNodes[alg]] \cap execs_{\mathsf{ActHide}(H_{VL}, U(VLNodes[alg]) \| R(RW \| VW \| Vbcast))}\}$. ■

11.3.4 Stabilizing emulations

Now we finally tie all this back to the concept of VSA layer emulations and stabilizing VSA layer emulations. We've described $VEmu[alg]$, which is a system that emulates the VSA layer for any VSA layer algorithm alg. However, a VSA layer emulation (Definition 8.3) is concerned with physical layer programs, which don't include leader election services or totally ordered broadcast services, that emulate virtual layer programs. Here we relate our emulation algorithm to the implementations of the leader election and totally ordered broadcast services, which allows us to talk about an implementation of the VSA layer using the physical layer. We do this by defining our VSA layer emulation algorithm based on our implementations of leader election and totally ordered broadcast, together with $VSAE[alg]$ for each $alg \in VAlgs$; we replace the leader election and totally ordered broadcast specification automata ($TObcast, LeadMain$, and $LeadCL_p, TOBDelay_p$, and $TOBFilter_p$ for each p in P) in $VEmu[alg]$ with the physical layer implementations ($TOBImpler_p$ and $Leader_p$ for each p in P) of these automata (Lemma 11.22). We then show that this also defines a stabilizing VSA layer emulation algorithm (Theorem 11.24).

Lemma 11.22 • *Let $amap : VAlgs \to PAlgs$ be defined as follows:*

For each $alg \in VAlgs$, $amap[alg]$ is the function from $P \to PProgram_p$ such that for each $p \in P$, $amap[alg](p) = \mathsf{ActHide}(H_{VEmu}, TOBImpler_p \| Leader_p \| CE[alg]_p \| VSAE[alg]_p)$.

This describes the mapping of VSA layer algorithms to physical layer algorithms that map each process to the composition of its totally ordered broadcast and leader election implementation pieces and the CE and $VSAE$ pieces for the particular VSA algorithm.

- *Let t_{stab} be any t such that $t > 2d + 2\epsilon_{sample} + t_{slice}$.*

- *Let $\mathcal{B}$ be* $\{PLNodes[amap[alg]] \mid alg \in VAlgs\}$.

  These are programmable components of the emulating system, namely the physical nodes.

- *Let $\mathcal{C}$ be* $\{VLNodes[alg] \mid alg \in VAlgs\}$.

  These are programmable components of the emulated system, namely the virtual nodes and client nodes.

- *Let emu be the function of type $\mathcal{C} \rightarrow \mathcal{B}$ such that for each $alg \in VAlgs$,* $emu(VLNodes[alg]) = PLNodes[amap[alg]]$.

- *Let S be the function in Definition 11.12.*

Then $amap$ is an S-constrained VSA layer emulation algorithm. (In other words, for each $alg \in VAlgs$, having each process run the $Fail$-transform of $amap[alg](p)$ together with the RW and $Pbcast$ produces traces that look like traces of executions of the virtual layer running alg and in S, after some action hiding.)

Proof: By Definition 8.3 of a VSA layer emulation algorithm, we must show that $(\mathcal{B}, RW\|Pbcast, H_{PL})$ emulates $(\mathcal{C}, RW\|VW\|Vbcast, H_{VL})$ constrained to S with emu. By Definition 4.1 of emulation, this means that we must show that for each $C \in \mathcal{C}$, $traces_{\mathsf{ActHide}(H_{PL}, emu(C)\|RW\|Pbcast)} \subseteq \{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(H_{VL}, C\|RW\|VW\|Vbcast)}\}$. Substituting for the components in this expression, we must show that for each $alg \in VAlgs$ and each α in $execs_{\mathsf{ActHide}(H_{VEmu} \cup H_{PL}, \prod_{p \in P} Fail(TOBImpler_p\|Leader_p\|CE[alg]_p\|VSAE[alg]_p)\|RW\|Pbcast)}$, there exists an α' in $execs_{\mathsf{ActHide}(H_{VL}, VLayer[alg])}$ such that:

1. $trace(\alpha) = trace(\alpha')$.

2. $\alpha' \in S(VLNodes[alg])$.

(In other words, we must show that for each VSA layer algorithm alg, an execution of the emulation algorithm at the physical layer shares the same trace as that of an execution of the virtual layer that also satisfied the properties of S, after some action hiding.)

Consider execution α. We first show how to break down the execution into component executions that are related to executions of components of $VEmu[alg]$, rather than the physical layer. We then paste these executions together to arrive at an execution of $VEmu[alg]$, which we have shown (Lemma 11.13) to behave as desired executions of the virtual layer.

By Lemma 2.14 and Theorem 5.3, we know that $\alpha\lceil(A_{TOBImpl}, V_{TOBImpl})$ is an execution of ***TOBImpl***. We also know, by Lemma 9.15, that there must exist an initial state y_{TOB} of ***TOBspec*** such that $\alpha.fstate\lceil X_{TOBimpl}\mathcal{R}_{TOB}y_{TOB}$. By Lemma 9.17, this implies that there exists some execution α_{TOB} of ***TOBspec*** that starts in state y_{TOB} such that:

- $trace(\alpha_{TOB}) = trace(\alpha\lceil(A_{TOBimpl}, V_{TOBimpl}))$.
- $\alpha_{TOB}\lceil(A_{RW}, V_{RW}) = \alpha\lceil(A_{RW}, V_{RW})$.
- For each $p \in P$, $\alpha\lceil(\{\mathsf{fail}_p, \mathsf{restart}_p\}, \{failed_p\}) = \alpha_{TOB}\lceil(\{\mathsf{fail}_p, \mathsf{restart}_p\}, \{failed_p\})$.

Similar reasoning for $LeadImpl$ and $LeadSpec$ gives us an execution α_{Lead} of ***LeadSpec*** such that:

- $trace(\alpha_{Lead}) = trace(\alpha\lceil(A_{LeadImpl}, V_{LeadImpl}))$.
- $\alpha_{Lead}\lceil(A_{RW}, V_{RW}) = \alpha\lceil(A_{RW}, V_{RW})$.
- For each $p \in P$, $\alpha\lceil(\{\mathsf{fail}_p, \mathsf{restart}_p\}, \{failed_p\}) = \alpha_{Lead}\lceil(\{\mathsf{fail}_p, \mathsf{restart}_p\}, \{failed_p\})$.

Consider executions $\alpha_p^{TOB} = \alpha_{TOB}\lceil(A_{Fail(TOBDelay_p\|TOBFilter_p)}, V_{Fail(TOBDelay_p\|TOBFilter_p)})$, $\alpha_p^{Lead} = \alpha_{Lead}\lceil(A_{Fail(LeadCl_p)}, V_{Fail(LeadCl_p)})$, and $\alpha_p^{CV} = \alpha\lceil(A_{Fail(CE[alg]_p\|VSAE[alg]_p)}, V_{Fail(CE[alg]_p\|VSAE[alg]_p)})$. Since each of these executions begins with the same value of the $failed_p$ variable, we have that Theorem 5.4 implies that for each $p \in P$ there exists an execution α_p of $Fail(TOBFilter_p\|TOBDelay_p\|LeadCl_p\|CE[alg]_p\|VSAE[alg]_p)$ that is the result of pasting the α_p^{TOB}, α_p^{Lead}, and α_p^{CV} component executions. (This follows from two

applications of Theorem 5.4.) This and Corollary 2.17 then imply that there exists an execution α'' of $VEmu[alg]$ such that $trace(\alpha) = trace(\alpha'')$.

Lemma 11.10 implies that there exists some initial state y of $VLayer[alg]$ such that $\alpha''.fstate\mathcal{R}_{Emu[alg]}y$, and Lemma 11.13 implies that there exists some α' in $execs_{\mathsf{ActHide}(H_{VL},VLayer[alg])}$ such that $trace(\alpha) = trace(\alpha')$ and $\alpha' \in S(VLNodes[alg])$. ■

Now we have shown that we have a VSA layer emulation. Before we can use this result to show that we have a *stabilizing* VSA layer emulation (Theorem 11.24), we need to also show the following result, which says that our low-level physical layer algorithm stabilizes to a point after which it looks like $VEmu[alg]$ started from a legal state. This connects the states of the implementation of $VEmu[alg]$ with the legal states $L^3_{VEmu[alg]}$ of $VEmu[alg]$. Since we have results showing that fragments of $VEmu[alg]$ starting in $L^3_{VEmu[alg]}$ are related to desirable execution fragments of $VLayer[alg]$ (Lemma 11.13), this will allow us to conclude the final stabilizing VSA layer emulation result. (It is worth noting that this proof would be improved if a general stabilizing composition result that takes into account $Fail$-transforms was available. I discuss this point in the Conclusions (Chapter 16).)

Lemma 11.23 *Let alg be an element of $VAlgs$.*
Let $Impler[alg]$ be $\prod_{p\in P} Fail(TOBImpler_p \| Leader_p \| CE[alg]_p \| VSAE[alg]_p)$.
Let $L[alg]$ be the set of states $x \in Q_{Impler[alg]\|RW\|Pbcast}$ such that $\exists y \in L^3_{VEmu[alg]}$:

1. $x\lceil X_{TOBImpl}\mathcal{R}_{TOB}y\lceil X_{TOBspec}$.
2. $x\lceil X_{LeadImpl}\mathcal{R}_{Leader}y\lceil X_{LeadSpec}$.
3. *For each $p \in P$, $x\lceil X_{CE[alg]_p\|VSAE[alg]_p} = y\lceil X_{CE[alg]_p\|VSAE[alg]_p}$.*

Then $Impler[alg]$ self-stabilizes to $L[alg]$ relative to $R(RW\|Pbcast)$ in time t_{stab}.

Proof: Consider any execution $\alpha_{PL} = \alpha^1_{PL}\alpha^2_{PL}\alpha^3_{PL}$ of the emulation algorithm at the physical layer such that $\alpha^1_{PL}.lstate = \alpha^2_{PL}.fstate$, $\alpha^2_{PL}.lstate = \alpha^3_{PL}.fstate$, $\alpha^1_{PL}.ltime = 2d + \epsilon_{sample} + (t_{stab} - 2d - 2\epsilon_{sample} - t_{slice})/2$, and $\alpha^2_{PL}.ltime = t_{stab} -$

$\alpha^1_{PL}.ltime$. Notice that this makes α^3_{PL} a state-matched t_{stab}-suffix of α_{PL}. We must show that $\alpha^3_{PL}.fstate$ is in $L[alg]$. The proof proceeds by showing that $\alpha^2_{PL}\alpha^3_{Pl}$ (the execution after the underlying leader election and totally ordered broadcast service implementations have stabilized) is related to an execution $\alpha^1_{VEmu[alg]}\alpha^2_{VEmu[alg]}$ of $VEmu[alg]$ started in invariant states of the leader election specification and the totally ordered broadcast specification. It then shows that $\alpha^2_{VEmu[alg]}$ (the execution of $VEmu[alg]$ after it has stabilized) is related to an execution of the virtual layer starting from a state with reachable states of $RW\|VW\|Vbcast$.

By Lemma 2.14, Corollary 2.17, and Theorem 5.3 (projection and pasting lemmas), we have the trivial result that $\alpha^1_{PL}\lceil(A_{TOBImpl}, V_{TOBImpl})$ $\alpha^2_{PL}\lceil(A_{TOBImpl}, V_{TOBImpl})$ $\alpha^3_{PL}\lceil(A_{TOBImpl}, V_{TOBImpl})$ is an execution of $U(TOBimpler)\|R(RW\|Pbcast)$. Since $\alpha^1_{PL}.ltime > 2d + \epsilon_{sample}$, Theorem 9.24 implies that $\alpha^2_{PL}.fstate\lceil X_{TOBImpl}$ is in $L_{TOBimpl}$. Lemma 9.18 implies there exists some reachable state of $TOBspec$ such that $\alpha^2_{PL}.fstate\lceil X_{TOBimpl}$ is related to it. Lemma 9.17 then implies that there exists an execution $\alpha^1_{TOBspec}\alpha^2_{TOBspec}$ of $R(TOBspec)$ such that:

1. $\alpha^2_{TOBspec}.fstate = \alpha^1_{TOBspec}.lstate$.
2. $\alpha^2_{PL}.fstate \mathcal{R}_{TOB} \alpha^1_{TOBspec}.fstate$ and $\alpha^3_{PL}.fstate \mathcal{R}_{TOB} \alpha^2_{TOBspec}.fstate$.
3. $\alpha^1_{TOBspec}.fstate \in reachable_{TOBspec}$ and $\alpha^2_{TOBspec}.fstate \in reachable_{TOBspec}$.
4. $trace(\alpha^1_{TOBspec}) = trace(\alpha^2_{PL}\lceil(A_{TOBImpl}, V_{TOBImpl})$ and $trace(\alpha^1_{TOBspec}) = trace(\alpha^2_{PL}\lceil(A_{TOBImpl}, V_{TOBImpl})$.

Lemma 2.14 then implies that there exists executions $\alpha^1_{RW}\alpha^2_{RW}$ of RW, $\alpha^1_{TObcast}\alpha^2_{TObcast}$ of $TObcast$, and $\alpha^{p,1}_{TOBFilDel}\alpha^{p,2}_{TOBFilDel}$ of $Fail(TOBFilter_p\|TOBDelay_p)$ for each $p \in P$ such that:

1. $\alpha^1_{RW} = \alpha^1_{TOBspec}\lceil(A_{RW}, V_{RW}) = \alpha^2_{PL}\lceil(A_{RW}, V_{RW})$, and $\alpha^2_{RW} = \alpha^2_{TOBspec}\lceil(A_{RW}, V_{RW}) = \alpha^3_{PL}\lceil(A_{RW}, V_{RW})$.
2. $\alpha^1_{TObcast} = \alpha^1_{TOBspec}\lceil(A_{TObcast}, V_{TObcast})$ and $\alpha^2_{TObcast} = \alpha^2_{TOBspec}\lceil(A_{TObcast}, V_{TObcast})$.

3. For each $p \in P$, $\alpha^{p,1}_{TOBFilDel} = \alpha^1_{TOBspec}\lceil(A_{Fail(TOBFilter_p\|TOBDelay_p)}, V_{Fail(TOBFilter_p\|TOBDelay_p)})$ and $\alpha^1_{TOBspec}.fstate(failed_p) = \alpha^2_{PL}.fstate(failed_p)$.

4. For each $p \in P$, $\alpha^{p,2}_{TOBFilDel} = \alpha^2_{TOBspec}\lceil(A_{Fail(TOBFilter_p\|TOBDelay_p)}, V_{Fail(TOBFilter_p\|TOBDelay_p)})$ and $\alpha^2_{TOBspec}.fstate(failed_p) = \alpha^3_{PL}.fstate(failed_p)$.

Similar reasoning for $LeadSpec$ and $LeadImpl$ tells us that there exist executions $\alpha^1_{LeadSpec}\alpha^2_{LeadSpec}$ of ***LeadSpec*** and executions $\alpha^1_{LeadMain}\alpha^2_{LeadMain}$ of ***LeadMain*** and $\alpha^{p,1}_{LeadCl}\alpha^{p,2}_{LeadCl}$ of $Fail(LeadCl_p)$ for each $p \in P$ such that:

1. $\alpha^1_{LeadSpec}\lceil(A_{RW}, V_{RW}) = \alpha^2_{PL}\lceil(A_{RW}, V_{RW})$, and $\alpha^2_{Leadspec}\lceil(A_{RW}, V_{RW}) = \alpha^3_{PL}\lceil(A_{RW}, V_{RW})$.

2. $\alpha^1_{LeadMain} = \alpha^1_{Leadspec}\lceil(A_{LeadMain}, V_{LeadMain})$ and $\alpha^2_{LeadMain} = \alpha^2_{Leadspec}\lceil(A_{LeadMain}, V_{LeadMain})$.

3. For each $p \in P$, $\alpha^{p,1}_{LeadCl} = \alpha^1_{Leadspec}\lceil(A_{Fail(LeadCl_p)}, V_{Fail(LeadCl_p)})$ and $\alpha^1_{Leadspec}.fstate(failed_p) = \alpha^2_{PL}.fstate(failed_p)$.

4. For each $p \in P$, $\alpha^{p,2}_{LeadCl} = \alpha^2_{Leadspec}\lceil(A_{Fail(LeadCl_p)}, V_{Fail(LeadCl_p)})$ and $\alpha^2_{Leadspec}.fstate(failed_p) = \alpha^3_{PL}.fstate(failed_p)$.

Since for each $p \in P$ and $i \in \{1, 2\}$, the value of $failed_p$ is the same in the first state of $\alpha^{p,i}_{TOBFilDel}$, $\alpha^{p,i}_{LeadCl}$, and $\alpha^{i+1}_p\lceil(A_{Fail(CE[alg]_p\|VSAE[alg]_p)}, V_{Fail(CE[alg]_p\|VSAE[alg]_p)})$, Theorem 5.4 (applied twice) implies that for each $p \in P$ there exists an execution fragment α^1_p of $Fail(LeadCl_p\|TOBFilter_p\|TOBDelay_p\|CE[alg]_p\|VSAE[alg]_p)$ such that $\alpha^1_p\lceil(A_{Fail(TOBFilter_p\|TOBDelay_p)}, V_{Fail(TOBFilter_p\|TOBDelay_p)}) = \alpha^{p,1}_{TOBFilDel}$, $\alpha^1_p\lceil(A_{Fail(LeadCl_p)}, V_{Fail(LeadCl_p)}) = \alpha^{p,1}_{LeadCl}$, and $\alpha^1_p\lceil(A_{Fail(CE[alg]_p\|VSAE[alg]_p)}, V_{Fail(CE[alg]_p\|VSAE[alg]_p)}) = \alpha^2_{PL}\lceil(A_{Fail(CE[alg]_p\|VSAE[alg]_p)}, V_{Fail(Ce[alg]_p\|VSAE[alg]_p)})$. Corollary 2.17 then implies that there exists an execution fragment $\alpha^1_{VEmu[alg]}$ of $VEmu[alg]$ that is the result of pasting executions α^1_p, α^1_{RW}, $\alpha^1_{LeadMain}$, and $\alpha^1_{TObcast}$ and hiding actions in H_{PL}. We arrive at $\alpha^2_{VEmu[alg]}$ similarly.

Notice that $\alpha^1_{VEmu[alg]}\alpha^2_{VEmu[alg]}$ is in the set of executions of $Start(VEmu[alg], \{x \in Q_{VEmu[alg]} \mid x\lceil X_{LeadSpec} \in Inv_{LeadSpec} \wedge x\lceil X_{TOBspec} \in Inv_{TOBspec}\})$, and that $\alpha^2_{VEmu[alg]}$ is the state-matched $t_{stab} - \alpha^1_{PL}.ltime$-suffix of $\alpha^1_{VEmu[alg]}\alpha^2_{VEmu[alg]}$. Since $t_{stab} - \alpha^1_{PL}.ltime > \epsilon_{sample} + t_{slice}$, Lemma 11.19 implies that $\alpha^2_{VEmu[alg]}$ is in the set of execution fragments of $VEmu[alg]$ starting in a state in $L^3_{VEmu[alg]}$.

Set y to be $\alpha^2_{VEmu[alg]}.fstate$. All that remains to show is that the three conditions of the lemma hold between $\alpha^3_{PL}.fstate$ and y. This follows immediately from construction of $\alpha^2_{VEmu[alg]}$. ■

Now we can conclude the final result, namely that $amap$ is an S-constrained t-stabilizing VSA layer emulation algorithm. The proof is a direct consequence of Lemmas 11.22, 11.23, and 11.13– Since we have already shown that $amap$ is an S-constrained VSA layer emulation algorithm (Lemma 11.22), Definition 4.4 of an S-constrained t-stabilizing VSA layer emulation algorithm implies that all that remains is to show that the traces of the emulation algorithm at the physical layer stabilizes to traces of execution fragments of the virtual layer that both satisfy the properties of S and that start in reachable states of $RW\|VW\|Vbcast$. Lemma 11.23 gives that the executions of the emulation algorithm at the physical layer stabilize to executions beginning in states in $L[alg]$, which we show to be related to executions of $VEmu[alg]$ with the same trace and that begin in states in $L^3_{VEmu[alg]}$. Lemma 11.13 gives that these executions are in turn related to executions of the virtual layer with the same trace and that start in reachable states of $RW\|VW\|Vbcast$.

Theorem 11.24 *$amap$ is an S-constrained t_{stab}-stabilizing VSA layer emulation algorithm.*

Proof: By Definition 8.3 of a stabilizing VSA layer emulation algorithm, we must show that $(\mathcal{B}, RW\|Pbcast, H_{PL})$ emulation stabilizes in time t_{stab} to $(\mathcal{C}, RW\|VW\|Vbcast, H_{VL})$ constrained to S with emu. By Definition 4.4, this means that we must show that $(\mathcal{B}, RW\|Pbcast, H_{PL})$ emulates $(\mathcal{C}, RW\|VW\|Vbcast, H_{VL})$ constrained to S with emu (which we have already shown in Lemma 11.22) and that $traces_{\mathsf{ActHide}(H_{PL}, U(emu(C))\|R(RW\|Pbcast))}$ stabilizes in time t_{stab} to $\{trace(\alpha) \mid \alpha \in S(C) \cap execs_{\mathsf{ActHide}(H_{VL}, U(C)\|R(RW\|VW\|Vbcast))}\}$.

By Lemma 11.23 we know that $execs_{\mathsf{ActHide}(H_{PL},U(emu(C))\|R(RW\|Pbcast))}$ stabilizes in time t_{stab} to $frags^{L[alg]}_{\mathsf{ActHide}(H_{PL},emu(C)\|RW\|Pbcast)}$. By reasoning similar to that in the proof of Lemma 11.22, following the same tedious process of breaking down executions of the physical layer algorithm into component executions that can be related to executions of $VEmu[alg]$, we know that for any $\alpha' \in frags^{L[alg]}_{\mathsf{ActHide}(H_{PL},emu(C)\|RW\|Pbcast)}$, there exists some $\alpha'' \in frags^{L^3_{VEmu[alg]}}_{\mathsf{ActHide}(H_{VEmu},VEmu[alg])}$ with the same trace as α'. Because $\alpha'' \in frags^{L^3_{VEmu[alg]}}_{\mathsf{ActHide}(H_{VEmu},VEmu[alg])}$, Lemma 11.13 implies that there exists some $\alpha \in S(C) \cap execs_{\mathsf{ActHide}(H_{VL},U(C)\|R(RW\|VW\|Vbcast))}$ such that $trace(\alpha) = trace(\alpha')$. ■

In other words, consider any VSA layer program alg, and the physical nodes running their emulation of the VSA layer running alg (consisting of totally ordered broadcast and leader election implementations and the main emulation programs for VSAs and their local clients). Traces of this system where the physical nodes start in an arbitrary state and are run with $RW\|Pbcast$ in a reachable state stabilize in time t_{stab} to traces of execution fragments of the VSA layer running alg (and satisfying properties of S), only from arbitrary states of the clients and VSAs.

Given this result, an application programmer can now write programs for the VSA layer without reasoning about the implementation of the VSA layer.

11.3.5 Message complexity

The message overhead introduced by this algorithm consists of the extra messaging generated for the leader election service (one message per process), and the one $|vstate|$-sized message communicated every t_{slice} time.

11.4 Extending the implementation to allow more failures

Rather than considering a VSA failed immediately after a vstate message fails to be sent by a leader, we can extend the emulation to allow some finite number k of such rounds to pass before failing the VSA. This extension potentially makes the VSA more fault-tolerant, though it does introduce some additional complication. If a leader is supposed to

perform broadcasts on the VSA's behalf, but fails or leaves before sending them, the next leader needs to transmit the messages. Since emulators store outgoing VSA messages in a local outgoing queue but clears that queue at the beginning of a new round, an extended algorithm must allow all emulators to carry their outgoing queue forward into subsequent rounds. A new leader then just transmits any messages stored in its outgoing queue and removes them. To prevent messages from being rebroadcast by future leaders, emulators that receive a VSA message broadcast by the leader remove it from their own outgoing queues.

Stabilization of an extended algorithm would also take about k times the amount of time of the original algorithm.

Part III

VSA layer applications

Part III of this thesis describes applications that we implement using the VSA layer. In the thesis, each implementation, whether of the VSA programming layer or of applications built on the layer, is proved correct using the TIOA formal framework. The first three chapters describe a suite of three algorithms that together define a program for the VSA layer that offers end-to-end routing; Chapters 12 and 13 describe geocast and location management automata that are parts of a larger end-to-end routing automaton at each region. The last chapter describes a motion coordination application.

In Chapter 12, I describe the first piece of the end-to-end routing application, a stabilizing region-to-region communication service. The algorithm is based on a shortest path procedure. When a region receives a geocast message it has not previously seen from region u to region v for which it is on a shortest path from u to v, it forwards the message closer to region v.

Chapter 13 describes the second piece of the end-to-end routing application, a location management service built over the geocast service of Chapter 12. The solution is based on the concept of *home location servers*, where each mobile client identifier hashes to a home location, a region of the network that is periodically updated with the location of the client and that is responsible for answering queries about the client's location. The periodic location updates and the forwarding of queries and responses are done using the geocast service of Chapter 12.

In Chapter 14, I describe a simple self-stabilizing program for the VSA layer to provide a mobile client end-to-end routing service. A client sends a message to another client by forwarding the message to its local VSA, which then uses the home location service to discover the destination client's region and forwards the message to that region using the geocast service.

Finally, in Chapter 15, we study how the VSA layer can help us solve the problem of coordinating the behavior of a set of autonomous mobile robots (physical nodes) in the presence of changes in the underlying communication network as well as changes in the set of participating robots. Each VSA must decide based on its own local information which robots to keep in its own region, and which to assign to neighboring regions; for each robot that remains, the VSA determines where on the curve the robot should reside. Unlike in the

prior three algorithms (Geocast, location management, and end-to-end communication), the client motion in the motion coordination protocol is controllable by the client, allowing the client to change its motion trajectory based on instructions from a VSA.

Chapter 12

GeoCast

In this chapter, we describe a self-stabilizing algorithm that uses $RW, VW, Vbcast$, and $VBDelay_u, u \in U$ automata to provide geographic routing between regions of the network, allowing communication between regions of the virtual infrastructure. In order to route location information between geographic regions, we use a shortest path algorithm.

GeoCast algorithms [14,73], GOAFR [59], and algorithms for "routing on a curve" [72] route messages based on the location of the source and destination, using geography to delivery messages efficiently. GPSR [57], AFR [60], GOAFR+ [59], polygonal broadcast [35], and the asymptotically optimal algorithm [60] are algorithms based on greedy geographic routing algorithms, forwarding messages to the neighbor that is geographically closest to the destination. The algorithms also address "local minimum situations", where the greedy decision cannot be made. GPSR, GOAFR+, and AFR achieve, under reasonable network behavior, a linear order expected cost in the distance between the sender and the receiver.

In [37], we used a variant of the VSA layer to simplify the implementation of the geocast routing service. There we used a simple variant of a greedy depth first search algorithm to communicate messages between VSAs. Here we implement the geocast portion of a larger VSA program (the end-to-end routing program described in Chapter 14) using a simple shortest path routing algorithm that runs on top of the VSA layer's fixed infrastructure.

In the rest of this chapter, we describe the service (Section 12.1), then describe a set

```
1  Signature:
     Input time(t)_u, t ∈ ℝ^≥0
3    Input geocast(m, v)_u, m ∈ Msg, v ∈ U
     Input vrcv(⟨geocast, m, w, v, t⟩)_u, m∈ Msg,w,v∈ U,t∈ ℝ^≥0
5    Output vcast(⟨geocast, m, w, v, t⟩)_u, m∈ Msg,w,v∈ U,t∈ ℝ^≥0
     Output georcv(m)_u, m ∈ Msg
7    Internal ledgerClean(⟨m,w,v,t⟩)_u, m∈ Msg,w,v∈ U,t∈ ℝ^≥0

9  State:
     analog clock: ℝ^≥0 ∪ {⊥}, initially ⊥
11   ledger: (Msg × U × U × ℝ^≥0) → Bool, initially null

13 Trajectories:
     evolve
15     d(clock) = 1
     stop when
17     ∃m: Msg, ∃w, v: U, ∃t: ℝ^≥0: [ledger(⟨m, w, v, t⟩) ≠ null
         ∧ (ledger(⟨m, w, v, t⟩)= false ∨ [u ≠ w ∧ clock = t]
19          ∨ clock < t ∨ t+(e+d)dist(w, u)< clock-ε
            ∨ dist(w, v) ≠ dist(w, u) + dist(u, v)) ]
21
   Transitions:
23   Input time(t)_u
     Effect:
25     if clock ≠ t then
         ledger ← null
27     clock ← t

     Input geocast(m, v)_u
     Effect:                                                            30
       if (ledger(m, u, v, clock) = null∨ u= v)∧ clock ≠ ⊥then
         ledger(m, u, v, clock) ← false                                 32

     Output vcast(⟨geocast, m, w, v, t⟩)_u                              34
     Precondition:
       ledger(⟨m, w, v, t⟩) = false ∧ v ≠ u                             36
     Effect:
       ledger(⟨m, u, v, t⟩) ← true                                      38

     Input vrcv(⟨geocast, m, w, v, t⟩)_u                                40
     Effect:
       if ledger(⟨m,w,v,t⟩)= null ∧ t+(e+d)dist(w, u)≥ clock            42
           ∧ t< clock∧ dist(w, v)= dist(w, u)+dist(u, v)
                              ∧ v ≠ w ≠ u then                          44
         ledger(⟨m, w, v, now⟩) ← false
                                                                        46
     Output georcv(m)_u
     Local: v: U, t: ℝ^≥0                                               48
     Precondition:
       ledger(⟨m, v, u, t⟩) = false                                     50
     Effect:
       ledger(⟨m, v, u, t⟩) ← true                                      52

     Internal ledgerClean(⟨m, w, v, t⟩)_u                               54
     Precondition:
       t + (e + d) dist(w, u) < clock ∨ (u ≠ w ∧ clock = t)             56
       ∨ clock < t ∨ dist(w, v) ≠ dist(w, u) + dist(u, v)
     Effect:                                                            58
       ledger(⟨m, w, v, t⟩) ← null
```

Figure 12-1: VSA geocast automaton at region u, V_u^{Geo}.

of legal states of the service and properties of executions starting in legal states (Section 12.3), and finally argue that our service is self-stabilizing (Section 12.4).

12.1 Specification

Our geocast service allows a region u to broadcast a message m to region v via $\mathsf{geocast}(m, v)_u$. It allows a region to receive such a broadcast message via $\mathsf{georcv}(m)_v$, under certain conditions. The TIOA specification algorithm for individual regions is in Figure 12-1. The complete service, $GeoCast$, is the composition of $\prod_{u \in U} Fail(V_u^{Geo} \| VBDelay_u)$ with $RW \| VW \| Vbcast$. In other words, the service consists of a $Fail$-transformed automata at each region of the geocast machine and $VBDelay$ machine for that region, as well as $RW \| VW \| Vbcast$.

VSA-to-VSA communication is based on a shortest path procedure. We assume that each VSA can calculate its hop count distance in the static region graph to other VSAs.

When a VSA receives a geocast message it has not previously seen from region u to region v for which it is on a shortest path from u to v, it forwards the message, tagged with a geocast label, via a vcast output. Whenever the destination VSA receives such a message it performs a georcv of the message.

Note: Notice that for each $u \in U$, V_u^{Geo} is technically not a valid VSA since its external interface contains non-vcast, vrcv, time actions. However, we will later (in Chapters 13 and 14) be composing this automaton with other automata and hiding these actions to produce new automata that are VSAs. In the meantime, we may refer to these almost-VSAs as VSAs, with the understanding that this technical detail will be resolved later. None of the results in this chapter require that V_u^{Geo} be a VSA.

In this thesis, V_u^{Geo} happens to be part of a specific VSA that is the composition of V_u^{Geo} with specific other automata, namely a location management automaton and an end-to-end routing automaton. However, the V_u^{Geo} automaton can be part of other VSAs as well, as long as it is composed with automata that allow us to hide the geocast and georcv actions. For example, consider the following variant of the geocast service, $C - Geocast$: The $C - Geocast$ service allows a client C_p to broadcast a message m to clients in region v via C-geocast$(m, v)_p$. It allows each client C_q in region v to receive such a broadcast message via C-georcv$(m)_q$, under certain conditions.

The $C - Geocast$ application can be implemented using the VSA layer in the following way: Each region's VSA is composed of two subprograms, V_u^{Geo} and a new automaton V_u^{C-Geo} that interacts with V_u^{Geo} and has an external interface consisting only of time, vbcast, vrcv, geocast, and georcv actions; the geocast and georcv actions are hidden in the composition of V_u^{Geo} and V_u^{C-Geo}, resulting in a new machine that is a valid VSA. Whenever a client in a region u receives a C-geocast$(m, v)_p$ input, it vcasts a $\langle$C-geocast$, m, v, u\rangle$ message to its local VSA. When the local VSA's V_u^{C-Geo} subprogram vrcvs such a message from a client in its region, it submits a geocast$(m, v)_p$ input to the local V_u^{Geo} VSA subprogram. When such a georcv$(m)_v$ later occurs at the V_v^{Geo} VSA subprogram in region v, the output goes to the local VSA's V_v^{C-Geo} subprogram. This subprogram then performs a vcast$(\langle$C-georcv$, m, v\rangle)_v$ output. Any client C_q in region v that receives this message through vrcv performs a C-georcv$(m)_q$ output.

Detailed VSA code description

The following code description refers to the TIOA code for the machine at region u, V_u^{Geo}, in Figure 12-1.

The state variable $ledger$ keeps track of information with respect to each non-expired geocast-tagged message (one for which V_u^{Geo} might still receive messages) that the VSA has heard of. The message is stored in $ledger$ together with its source, destination, and timestamp. For each such unique tuple of message information, the table stores a Boolean indicating whether the region has yet processed the message, either by forwarding it in a geocast broadcast or by receiving it. If the Boolean is false, then the VSA has not yet processed the message.

When V_u^{Geo} receives a time(t) input (line 23, supplied by the virtual time service VW), it checks its local $clock$ to see if it matches t. If not (line 25), V_u^{Geo} resets its $ledger$ values (line 26). Either way, V_u^{Geo} sets its $clock$ to t (line 27). (Notice that in normal operation, once an alive VSA has received its first time input its $clock$ should always be equal to the real time since its $clock$ variable advances at the same rate as real time.)

When V_u^{Geo} receives a geocast$(m, v)_u$ input at some time t and either it is the first occurrence of geocast$(m, v)_u$ at time t or $u = v$ (lines 29-31), V_u^{Geo} sets $ledger(\langle m, u, v, clock\rangle)$ to false (line 32), indicating the geocast tuple must be processed so that the message can be forwarded to region v.

Whenever any V_u^{Geo} has a false $ledger$ entry for some tuple $\langle m, w, v, t\rangle$ where $u = v$, the message has reached its destination, and V_u^{Geo} performs a georcv$(m)_u$ output (lines 47-50) and sets the $ledger$ entry to true (line 52). If, on the other hand, $u \neq v$ (line 36, meaning V_u^{Geo} has heard of a particular geocast it should forward but has not yet done anything about it), V_u^{Geo} sends a message consisting of a geocast tag and the tuple via vcast (line 34), and sets the $ledger$ entry to true (line 38).

Whenever V_u^{Geo} receives a $\langle \text{geocast}, m, w, v, t\rangle$ message (line 40), it checks the following in lines 42-44: (1) it does not yet have a non-null $ledger$ entry for the tuple, (2) u is on some shortest path between w and v (equivalent to saying that $dist(w, v) = dist(w, u) + dist(u, v)$), and (3) the current time $clock$ is not more than $t + (e+d)dist(w, u)$

(meaning that V_u^{Geo} received the message no later than the maximum amount of time a shortest region path trip from w would have taken to reach u). If the three conditions hold then V_u^{Geo} sets $ledger(\langle m, w, v, t\rangle)$ to true (line 45).

The internal action $\mathsf{ledgerClean}(\langle m, w, v, t\rangle)_u$ (line 54) serves to clean $ledger$ of tuples that correspond to geocasts that V_u^{Geo} no longer will be involved with (line 59). In particular it clears entries for which $t + (e + d)dist(w, u) < clock$ (line 56), corresponding to geocasts that are too old for V_u^{Geo} to forward. This action is also used for local correction, removing $ledger$ entries for geocast messages between regions that region u is not on a shortest path between and for geocast messages that are timestamped in the future (lines 56-57). Self-stabilization of the system as a whole is then accomplished by the clear-out of older geocast records based on their timestamps, and by the screening of incoming messages in lines 42-44. Too old forwarded messages are eliminated from the system and newer forwarded messages do not impact the treatment of the older ones.

12.2 Properties of executions of the geocast service

We say that a geocast by a region u to a region v, at time t is *serviceable*, if there exists at least one shortest path from u to v of regions that are nonfailed and have $clock$ values equal to the real-time for the entire interval $[t, t + (e + d)dist(u, v)]$.

With this definition, we can show the following result:

Lemma 12.1 *The service guarantees that in each execution α of $GeoCast$, there exists a function mapping each* $\mathsf{georcv}(m)_v$ *event to the* $\mathsf{geocast}(m, v)_u$ *event that caused it such that the following hold:*

1. Integrity: *If a* $\mathsf{georcv}(m)_v$ *event π is mapped to a* $\mathsf{geocast}(m, v)_u$ *event π', then π' occurs before π.*

2. Same-time self-delivery: *If a* $\mathsf{georcv}(m)_v$ *event π is mapped to a* $\mathsf{geocast}(m, v)_v$ *event π' where π' occurs at time t, then event π occurs at time t.*

3. Bounded-time delivery: *If a* $\mathsf{georcv}(m)_v$ *event π is mapped to a* $\mathsf{geocast}(m, v)_u$

event π' where π' occurs at time t and $u \neq v$, then event π occurs in the interval $(t, t + (e + d)dist(u, v)]$.

4. Reliable self-delivery: *This guarantees that a geocast will be received if sent to itself and no failures occur: If a* $\mathsf{geocast}(m, v)_v$ *event π' occurs at time t, $\alpha.ltime > t$, and region v does not fail at time t, then there exists a* $\mathsf{geocv}(m)_v$ *event π such that π is mapped to some* $\mathsf{geocast}(m, v)_v$ *event (not necessarily π') at time t.*

5. Reliable serviceable delivery: *This guarantees that a geocast will be received if it is serviceable: If a* $\mathsf{geocast}(m, v)_u$ *event π' occurs at time t, $\alpha.ltime > t + (e + d)dist(u, v)$, and π' is serviceable, then there exists a* $\mathsf{georcv}(m)_v$ *event π such that π is mapped to some* $\mathsf{geocast}(m, v)_u$ *event (not necessarily π') at time t.*

Proof sketch: It is easy to define the mapping from georcv to geocast events described above as follows: For each $\mathsf{georcv}(m)_u$ event, there is some region v and time t where the tuple $\langle m, v, u, t\rangle$ is in a $ledger$ that changes from being mapped to false to being mapped to true (lines 50-52). We map the georcv event to the first $\mathsf{geocast}(m, u)_v$ event that occurs at time t.

It is easy to see that most of the properties hold. We show here that the most interesting properties, Bounded-time delivery and Reliable serviceable delivery, hold. To see that Bounded-time delivery holds, notice that for a $\mathsf{georcv}(m)_v$ to happen, there must be some $u \in U$ and $t \in \mathbb{R}^{\geq 0}$ such that $ledger(\langle m, u, v, t\rangle) =$ false. This can only occur if a $\mathsf{geocast}(m, v)_v$ occurred (trivially satisfying the property), or if a $\mathsf{vrcv}(\langle \mathsf{geocast}, m, u, v, t\rangle)_v$ occurred at some time t' to set the $ledger$ entry to false. For the second case, by the conditional on lines 42-43, the $ledger$ entry is only changed if $t + (e + d)dist(w, v) \leq t'$. By the stopping conditions on line 18, the $\mathsf{georcv}(m)_v$ must have occur at time t' as well, giving the result.

To see that the more interesting Reliable serviceable delivery property holds, assume that a $\mathsf{geocast}(m, v)_u$ event π' occurs at time t and π' is serviceable. Let one of the shortest paths of VSAs that satisfy the serviceability definition be $u_1, \cdots, u_{dist(u,v)-1}, v$, where u_1 is a neighbor of u and each region in the sequence neighbors the regions that precede or follow it in the sequence. We argue that there exists a $\mathsf{geocv}(m)_v$ event π such that π is

mapped to the first geocast$(m, v)_u$ event at time t. Since the first such geocast$(m, v)_u$ event occurs at an alive process that does not fail at time t, it will immediately vcast a geocast-tagged $\langle m, u, v, t\rangle$ message. Such a message takes more than 0, but no more than $e + d$ time to be delivered at neighboring regions, one of which is u_1. $V_{u_1}^{Geo}$ will then immediately vcast a geocast-tagged $\langle m, u, v, t\rangle$ message, since the conditional on lines 42-43 will hold. Such a message takes more than 0, but no more than $e + d$ time to be delivered at neighboring regions, one of which is u_2. Either the same case as for u_1 holds or u_2 received the earlier transmission and immediately transmitted or is about to transmit. This argument is repeated until a geocast-tagged $\langle m, u, v, t\rangle$ message is received at region v. This process will then immediately perform a geocast$(m)_v$ event. This event is mapped to the first geocast$(m, v)_u$ event at time t, and we are done. ■

12.3 Legal sets

Here we describe a legal set of $GeoCast$ by describing two legal sets, the second a subset of the first. Recall from Lemma 3.13 that a legal set of states for a TIOA is one where each closed execution fragment starting in a state in the set ends in a state in the set. We break the definition of the legal set up into two legal sets in order to simplify the proof reasoning and more easily prove stabilization later, in Section 12.4. At the end of this section, we discuss properties of execution fragments of $GeoCast$ that start in our set of legal states.

12.3.1 Legal set L_{geo}^1

The first set of legal states describes some properties that are locally checkable at a region and that become true at an alive process at the time of the first time input for the process and possibly a ledgerClean action.

Definition 12.2 *Let L_{geo}^1 be the set of states x of $GeoCast$ where all of the following hold:*

1. $x \lceil X_{RW\|VW\|Vbcast} \in Inv_{RW\|VW\|Vbcast}$.
   This says that the state restricted to the variables of the composition of RW, VW, and $Vbcast$ are reachable states of their composition.

2. *For each* $u \in U : [\neg failed_u \Rightarrow \forall \langle m, t\rangle \in to_send_u : rtimer - t \in [0, e]]$.

   This says that $VBDelay$ *messages queued for a region have been waiting in the buffer at least 0 and at most* e *time.*

3. *For each* $u \in U : (\neg failed_u \wedge clock_u = \bot)$*:*

   (a) *There are no* geocast *tuples in* $VBDelay_u.to_send$.

   This says that non-failed regions that have not yet received a time *input do not have any* geocast *messages queued up for sending.*

   (b) *For each* $\langle m, w, v, t\rangle : ledger(\langle m, w, v, t\rangle) \neq false$.

   This says that non-failed regions that have not yet received a time *input do not have any* $ledger$ *entries that need to be processed.*

4. *For each* $u \in U : (\neg failed_u \wedge clock_u \neq \bot)$*:*

   (a) $clock_u = now$.

   This says that non-failed regions that have a non-$\bot$ $clock$ *have a clock time that is the same as the actual time.*

   (b) *For each* $\langle m, w, v, t\rangle : ledger_u(\langle m, w, v, t\rangle) \neq null$ *(For each non-failed region with a non-*$\bot$ $clock$*, each non-null* $ledger$ *entry satisfies the following):*

   i. $t + (e + d)dist(w, u) \geq clock_u - \epsilon \wedge (t + (e + d)dist(w, u) \geq clock_u \vee ledger_u(\langle m, w, v, t\rangle) = true)$.

   This says the entry has not expired too long ago– if we add the maximum amount of time for a message to follow a shortest path from w *to our region to the time when the geocast message originated, the result is no less than* ϵ *before the current time. Also, if the tuple's expiration point has passed then* $ledger$ *maps it to true.*

   ii. $clock_u \neq t \vee u = w$.

   This says that if t *is equal to the current time, then the source of the geocast message must be the current region. (Recall that* vcast*s, such as of* geocast*-tagged messages, take non-0 time to be delivered, implying that the only current-time* $ledger$ *entries must be from self-*geocast*s.)*

iii. $(clock_u > t \wedge u = w) \Rightarrow ledger_u(\langle m, w, v, t\rangle) = true$.

This says that self-geocasts are processed at the time they occur.

iv. $clock_u \geq t$.

This says that entries in $ledger$ can't be for geocast messages sent in the future.

v. $dist(w, v) = dist(w, u) + dist(u, v)$.

This says that u must be on a shortest path between the sender of the geocast and the destination.

It is trivial to check that L^1_{geo} is a legal set for $GeoCast$:

Lemma 12.3 *L^1_{geo} is a legal set for $GeoCast$.*

12.3.2 Legal set L^2_{geo}

The next legal set, L^2_{geo}, is a subset of L^1_{geo} that satisfies additional properties with respect to the state of each V^{geo}_u and $Vbcast$. The properties are concerned with geocast tuples, whether they are in a region's $ledger$ or in transit in the communication service.

Definition 12.4 *Let L^2_{geo} be the set of states x of $GeoCast$ where all of the following hold:*

1. $x \in L^1_{geo}$.

   This says that L^2_{geo} is a subset of L^1_{geo}.

2. *For each $u \in U : (\neg failed_u \wedge clock_u \neq \bot)$, and for each $\langle m, w, v, t\rangle$: $ledger_u(\langle m, w, v, t\rangle) \neq null$:*

   (a) $[u \neq v \wedge ledger_u(\langle m, w, v, t\rangle) = true] \Rightarrow \exists t' \in \mathbb{R}^{\geq 0} : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, t'\rangle \in to_send_u \vee \exists t'' \geq t : \exists P' \subseteq P \cup U : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, u, t'', P'\rangle \in vbcastq$.

   This says that if the $ledger$ of an alive region with non-$\bot$ $clock_u$ maps a tuple $\langle m, w, v, t\rangle$ to true and u is not the destination, then the tuple tagged with geocast *is either in $VBDelay_u$ or in $vbcastq$. (Recall that $vbcastq$ contains a record of all previously* vcast *messages.)*

(b) $u \neq w \Rightarrow \exists t' \in [t, t+e] : \exists P' \subset P \cup U : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t', P'\rangle \in vbcastq$.

This says that if a non-source's $ledger$ *maps the tuple to a non-null value, then there exists a record of the original broadcast of the* geocast *tuple in* $vbcastq$ *within* e *time of the tuple's timestamp.*

3. *For each* $u \in U : \neg failed_u$*:* $\exists\langle\langle \mathsf{geocast}, m, w, v, t\rangle, t'\rangle \in to_send_u \Rightarrow [t + (e+d)dist(w,u) \geq now - rtimer_u + t' \wedge \exists t'' \in [t, t+e] : \exists P' \subset P \cup U : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t'', P'\rangle \in vbcastq]$.

   This says that if a nonfailed region's $VBDelay$ *queue contains a* geocast *message, then the timestamp on the message is such that it was sent by the region before it expired, and there exists a record of the original broadcast of the* geocast *tuple in* $vbcastq$ *within* e *time of the tuple's timestamp.*

4. *For each* $\langle\langle \mathsf{geocast}, m, w, v, t\rangle, u, t', P'\rangle \in vbcastq : [P' \neq \emptyset \Rightarrow \exists t'' \in [t, t+e] : \exists P'' \subset P : \langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t'', P''\rangle \in vbcastq]$.

   This says that if a geocast *tuple with timestamp* t *is in transit in* $Vbcast$ *(meaning the tuple has yet to be either delivered or dropped by each process), then a* vcast *of the tuple happened between time* t *and time* $t+e$ *and was either received or dropped by at least one process. (In other words, if a* geocast *tuple is still in transit, then there exists a record of the original broadcast of the* geocast *tuple in* $vbcastq$ *within* e *time of the tuple's timestamp.)*

Next we check that L^2_{geo} is a legal set for $GeoCast$.

Lemma 12.5 L^2_{geo} *is a legal set for* $GeoCast$*.*

Proof: Let x be any state in L^2_{geo}. By Definition 3.12 of a legal set, we must verify two things for state x:

- For each state x' of $GeoCast$ and action a of $GeoCast$ such that (x, a, x') is in the set of discrete transitions of $GeoCast$, state x' is in $L^1_{TOBimpl}$.

- For each state x' and closed trajectory τ of $GeoCast$ such that $\tau.fstate = x$ and $\tau.lstate = x'$, state x' is in L^2_{Geo}.

By Lemma 12.3, we know that if x satisfies the first property of L^2_{geo}, then any discrete transition of $GeoCast$ will lead to a state x' that still satisfies the first property, and any closed trajectory starting with state x will end in some state that satisfies the first property. This implies that we just need to check that in the two cases of the legal set definition, the state x' satisfies all parts of the second property of L^2_{geo}.

For the first case of the legal set definition, we consider each action:

- GPSupdate$(l, t)_p$, drop(n, j), fail$_u$, restart$_u$, geocast$(m, v)_u$, georcv$(m)_u$, ledgerClean$(\langle m, w, v, t\rangle)_u$: These are trivial to verify.

- time$(t)_u$: If $failed_u$ holds in state x, then none of the properties are affected. Let's consider the case where $\neg failed_u$. Since property 4(a) holds in state x, either $t = clock_u$, meaning all properties still hold since no changes to region u's state occur, or $clock_u = \bot$ and the action initializes $ledger_u$. In the second case, property 2 becomes trivially true, and property 4 is not affected. Since property 3(a) of L^1_{geo} holds in state x, we know that no geocast tuples are in to_send_u, making property 3 of L^2_{geo} trivially true.

- vrcv$(\langle \text{geocast}, m, w, v, t\rangle)_u$: The only non-trivial property to verify is property 2(b). Assume that $u \neq w$, meaning that the region receiving the message is not the region that received the associated geocast. We must show that there exists some $t' \in [t, t+e] : \exists P' \subset P \cup U$ such that the received tuple, tagged with w, t', and P' is in $vbcastq$. By the precondition for this action, we know that there exists some $\langle\langle \text{geocast}, m, w, v, t\rangle, w', t'', P''\rangle$ in $x(vbcastq)$ such that P'' is non-empty. Since state x satisfies property 4, we know that there exists some $t' \in [t, t+e]$ and P' a proper subset of $P \cup U$ such that $\langle\langle \text{geocast}, m, w, v, t\rangle, w, t', P'\rangle$is in $vbcastq$, showing the property.

- vcast$(\langle \text{geocast}, m, w, v, t\rangle)_u$: The only non-trivial properties to verify are properties 2(a) and 3. To check property 2(a) we consider two cases: one where $u = w$ and one where it does not. If $u \neq w$, then property 2(a) follows from the fact that property 2(b) held in state x. Otherwise, it follows from the fact that an effect of

the action is the addition of an appropriate tuple to to_send_u. To check property 3 we need to check that the tuple added to to_send_u has a timestamp t such that $t + (e + d)dist(w, u) \geq now$ and there is a record of the original broadcast of the geocast tuple. The first follows from the fact that property 4(b)i. of L^1_{geo} holds in state x. The second follows from the fact that property 2(a) holds in state x.

- $\mathsf{vcast}'(\langle \mathsf{geocast}, m, w, v, t\rangle, true)_u$: The only non-trivial properties to verify are properties 2(a) and 4. Property 2(a) is easy to see since an effect of this action is moving a tuple from to_send_u into $vbcastq$. To check property 4, we need to show that there is a $\langle\langle \mathsf{geocast}, m, w, v, t\rangle, w, t'', P''\rangle$ in $x(vbcastq) = x'(vbcastq)$, where $t'' \in [t, t + e]$, which follows from the fact that property 3 held in state x.

For the second case of the legal set definition, we now consider any closed trajectory τ such that $x = \tau.fstate$. Let x' be $\tau.lstate$. We must show that $x' \in L^2_{geo}$, by verifying that each property of L^2_{geo} holds. It is easy to see that because the only evolving variables referenced in the properties are $clock_u, rtimer_u$, and now which evolve at the same rate, properties 2 and 4 hold.

The only interesting property to check is property 3. In particular, the only thing of interest to check is that if a region u is not failed and its $VBDelay$ buffer contains a geocast tuple from region w with timestamp t and $VBDelay$ timer tag t', then $t + (e + d)dist(w, u) \geq now - rtimer_u + t'$. However, since now and $rtimer$ evolve at the same rate, the value on the right of the inequality remains the same over a trajectory. The values on the left of the inequality remain the same over a trajectory because they are discrete variables. ■

Properties of execution fragments starting in L^2_{geo}

One thing to note is that execution fragments of $GeoCast$ that begin in a state in L^2_{geo} satisfy a set of properties very close to the ones described for executions in Section 12.2. Recall that in Section 12.2, we showed that $GeoCast$ guarantees that for every execution there exists a function mapping each $\mathsf{georcv}(m)_v$ event to the $\mathsf{geocast}(m, v)_u$ event that caused it such that five properties (Integrity, Same-time self-delivery, Bounded-time deliv-

ery, Reliable self-delivery, and Reliable serviceable delivery) hold.

Now we consider execution fragments of $GeoCast$ rather than executions and show that properties similar to those in Section 12.2 still hold. The first property basically says that the properties of an execution of $GeoCast$ also hold for execution fragments of $GeoCast$ that begin in a state in L^2_{geo}, provided that we are allowed to consider a function that maps only a subset of georcv events in α. The second property constrains the set of georcv events that we don't map to be ones that occur early enough in the execution fragment that there is not required to be a corresponding geocast event.

Lemma 12.6 *$GeoCast$ guarantees that for every execution fragment α beginning in a state in L^2_{geo} there exists a subset Π of the* georcv$(m)_v$ *events in α such that:*

1. *There exists a function mapping each* georcv$(m)_v$ *event in Π to the* geocast(m, v) *event that caused it such that the five properties (Integrity, Same-time self-delivery, Bounded-time delivery, Reliable self-delivery, and Reliable serviceable delivery) hold.*

2. *For every* georcv$(m)_v$ *event π not in Π where π occurs at some time t, it must be the case that $t - \alpha.fstate(now) \leq (e + d) * \max_{u \in U} dist(u, v)$.*

Proof sketch: The two properties together say that execution fragments of $GeoCast$ that begin in a state in L^2_{geo} demonstrate behavior similar to that of executions of $GeoCast$, modulo several orphan georcvs that can be viewed as events that would have been mapped to geocast events that occur before the start of α. In particular, consider the same mapping described in Section 12.2. We can show the same results as in Section 12.2 for geocasts and those georcvs that are mapped to geocasts. Now consider each georcv$(m)_v$ that occurs at some t time after the start of the execution fragment and is not mapped to a geocast. We just need to show that there exists some region u such that $t \leq (e + d)dist(u, v)$, implying that the georcv could be viewed as being mapped to a geocast$(m, v)_u$ that occurs before the start of the execution fragment. Each such georcv corresponds to a $ledger$ entry that satisfies property 4(b) of L^1_{geo}. Taking the source region in the entry as u, we know that the associated timestamp t' satisfied the property that it was no more than

$(e+d)dist(u,v)$ old when the georcv occurred. Since this tuple must have been in the system (either in transit or in a $ledger$) at the beginning of the execution fragment, this implies that $t \leq (e+d)dist(u,v)$. ■

12.4 Self-stabilization

We've seen that L^2_{geo} is a legal set for $GeoCast$. Here we show that $\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u)$ self-stabilizes to L^2_{geo} relative to $R(RW\|VW\|Vbcast)$ (Theorem 12.9), meaning that if certain program portions of the implementation are started in an arbitrary state and run with $R(RW\|VW\|Vbcast)$, the resulting execution eventually gets into a state in L^2_{geo}. This is done in two phases, corresponding to the legal sets L^1_{geo} and L^2_{geo}.

Using Theorem 12.9, we then conclude that after an execution of $GeoCast$ has stabilized, the execution fragment from the point of stabilization on satisfies the properties described in Section 12.3.2.

The first lemma describes the first phase of stabilization, for legal set L^1_{geo}. It says that $\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u)$ self-stabilizes in time t^1_{geo} to L^1_{geo} relative to $R(RW\|VW\|Vbcast)$, where t^1_{geo} is any time greater than ϵ_{sample}:

Lemma 12.7 *Let t^1_{geo} be any t such that $t > \epsilon_{sample}$.*
$\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u)$ self-stabilizes in time t^1_{geo} to L^1_{geo} relative to $R(RW\|VW\|Vbcast)$.

Proof sketch: To see this result, just consider any time after each node has received a time input, which takes at most ϵ_{sample} time to happen. ■

The next lemma shows that starting from a state in L^1_{geo}, $GeoCast$ ends up in a state in L^2_{geo} within t^2_{geo} time, where t^2_{geo} is any time greater than $\epsilon + (e+d)(D+1)$. (Recall that D is the network diameter in region hops.) This result takes advantage of the timestamping of geocast tuples as a way to prevent data from being too old.

Lemma 12.8 *Let t^2_{geo} be any t such that $t > \epsilon + (e+d)(D+1)$.*
$frags^{L^1_{geo}}_{GeoCast}$ stabilizes in time t^2_{geo} to $frags^{L^2_{geo}}_{GeoCast}$.

Proof: By Lemma 3.21, we just need to show that for any length-t^2_{geo} prefix α of an element of $frags^{L^1_{geo}}_{GeoCast}$, $\alpha.lstate$ is in L^2_{geo}. We examine each property of L^2_{geo}.

By Lemma 12.7, since the first state of α is in L^1_{geo}, we know that property 1 of L^2_{geo} holds in each state of α.

For property 2(a) it is plain that for any state in α, any new tuple added to a region u's $ledger$ will satisfy the property since the tuple will initially map to false, making the property trivially hold with respect to that tuple. Also, any tuple that maps to false will continue to satisfy the property even when it changes to being mapped to true, since such a change only occurs when the geocast-tagged tuple is added to to_send. The tuple is then only removed from to_send if the process fails or a similar tuple is added to $vbcastq$, either or which would have property 2(a) continue to hold.

This leaves tuples with a non-u destination that a region u's $ledger$ maps to true in the first state of α. Since $\alpha.fstate \in L^1_{tob}$ and hence satisfies property 4(b)i., we know that such a tuple will have a timestamp no smaller than $now - \epsilon - (e + d)D$. This means that in $\alpha.lstate$, the entry will have been removed, giving us that the algorithm stabilizes to satisfy the property.

For property 3, consider what happens when a nonfailed region has a geocast tuple in its to_send buffer. The first thing we would like to show is that the tuple's timestamp is consistent with what it would have been if the tuple were broadcast before it expired. Since $\alpha.fstate \in L^1_{tob}$ and hence satisfies property 4(b)i., we know that any new messages added to to_send will satisfy this requirement. This leaves only problematic tuples that were present in to_send in $\alpha.fstate$. However, we know that each tuple in to_send spends at most e time there. Since this is less than t^2_{geo} we are done with this portion of property 3.

The remainder of property 3, together with property 2(b) and property 4 are very similar in their proof obligations. Hence, we only discuss the proof of property 4 here.

For property 4, notice that for each geocast tuple added for the first time in the system to a to_send queue and then propagated within e time to $vbcastq$, the property will hold and continue to hold as the message makes its way through the system. The only thing we need to consider are the tuples throughout the system in $\alpha.fstate$. Consider the worst case of a "bad" tuple in a to_send queue. The tuple could, at worst, take maximum time

to be propagated to $vbcastq$ and delivered at a client (which works out to $e + d$ time), and could contain a timestamp just under $e + d$ ahead of real-time in $\alpha.fstate$. The tuple will eventually stop being forwarded when it stops being accepted for $ledger$ entries, up to $(e+d)(D-1)$ later. Its entries in $ledger$s can take up to an additional $e+d+\epsilon$ time before being removed by ledgerClean actions. This total time of $\epsilon + (e + d)(D + 1)$ is less than t^2_{geo}, and we are done. ■

Now we can combine our stabilization results to conclude that $Fail(VBDelay_u \| V^{Geo}_u)$ components started in an arbitrary state and run with $R(RW \| VW \| Vbcast)$ stabilizes to L^2_{geo} in time t_{geo}, where t_{geo} is any t such that $t > \epsilon_{sample} + \epsilon + (e+d)(D+1)$. The result is a simple application of the transitivity of stabilization (Lemma 3.6) to the prior two results.

Theorem 12.9 *Let t_{geo} be any t such that $t > \epsilon_{sample} + \epsilon + (e + d)(D + 1)$. $\prod_{u \in U} Fail(VBDelay_u \| V^{Geo}_u)$ self-stabilizes in time t_{geo} to L^2_{geo} relative to $R(RW \| VW \| Vbcast)$.*

Proof: We must show that $execs_{U(\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u)) \| R(RW\|VW\|Vbcast)}$ stabilizes in time t_{geo} to $frags^{L^2_{geo}}_{\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u)) \| R(RW\|VW\|Vbcast)}$. By Corollary 3.11, $frags^{L^2_{geo}}_{\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u) \| R(RW\|VW\|Vbcast)}$ is the same as $frags^{L^2_{geo}}_{GeoCast}$. The result follows from the application of transitivity of stabilization (Lemma 3.6) on the two lemmas (Lemmas 12.7 and 12.8) above. Let $t^1_{geo} = \epsilon_{sample} + (t_{geo} - \epsilon_{sample} - \epsilon - (e+d)(D+1))/2$ and $t^2_{geo} = \epsilon + (e+d)(D+1) + (t_{geo} - \epsilon_{sample} - \epsilon - (e+d)(D+1))/2$. (These terms are chosen so as to satisfy the constraints that $t^1_{geo} > \epsilon_{sample}$ and $t^2_{geo} > \epsilon + (e+d)(D+1)$, as well as the constraint that $t^1_{geo} + t^2_{geo} = t_{geo}$.)

First, let B be $execs_{U(\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u)) \| R(RW\|VW\|Vbcast)}$, C be $frags^{L^1_{geo}}_{GeoCast}$, and D be $frags^{L^2_{geo}}_{GeoCast}$ in Lemma 3.6. Then by Lemma 3.6 and Lemmas 12.7 and 12.8, we have that $execs_{U(\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u)) \| R(RW\|VW\|Vbcast)}$ stabilizes in time $t^1_{geo} + t^2_{geo}$ to $frags^{L^2_{GeoCast}}_{GeoCast}$.

Since $t_{geo} = t^1_{geo} + t^2_{geo}$, we conclude that $\prod_{u\in U} Fail(VBDelay_u \| V^{Geo}_u)$ self-stabilizes in time t_{geo} to L^2_{geo} relative to $R(RW\|VW\|Vbcast)$. ■

With Lemma 12.6, this allows us to conclude that after an execution of $GeoCast$ has

stabilized, the execution fragment from that point on satisfies the properties in Section 12.3.2:

Lemma 12.10 *Let t_{geo} be any t such that $t > \epsilon_{sample} + \epsilon + (e+d)(D+1)$.*
Then $execs_{U(\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo})) \| R(RW \| VW \| Vbcast)}$ stabilizes in time t_{geo} to a set $\mathcal{A}$ of execution fragments such that for each $\alpha \in \mathcal{A}$, there exists a subset Π of the $\mathsf{georcv}(m)_v$ events in α such that:

1. *There exists a function mapping each $\mathsf{georcv}(m)_v$ event in Π to the $\mathsf{geocast}(m, v)$ event that caused it such that the five properties (Integrity, Same-time self-delivery, Bounded-time delivery, Reliable self-delivery, and Reliable serviceable delivery) hold.*

2. *For every $\mathsf{georcv}(m)_v$ event π not in Π where π occurs at some time t, it must be the case that $t - \alpha.fstate(now) \leq (e+d) * \max_{u \in U} dist(u, v)$.*

Chapter 13

Location Management

In this chapter, we describe a self-stabilizing algorithm for the location management part of the end-to-end routing service in Chapter 14. The algorithm is built on the Geocast service and the VSA layer and provides a location service that allows VSAs in the network to find relatively recent information about the region locations of clients. Each mobile client identifier hashes to a home location, a region of the network that is periodically updated with the location of the client, and that is responsible for answering queries about the client's location.

Finding the location of a moving client in an ad-hoc network is difficult, much more so than in cellular mobile networks where a fixed infrastructure of wired support stations exist (as in [54]), or in sensor networks where some approximation of a fixed infrastructure may exist [6]. A *location service* in ad-hoc networks is a service that allows any client to discover the location of any other client using only its identifier. The basic paradigm for location services that we use here is that of a *home location service*: Hosts called *home location servers* are responsible for storing and maintaining the location of other hosts in the network [1, 48, 62]. Several ways to determine the sets of home location servers, both in the cellular and entirely ad-hoc settings, have been suggested.

The locality aware location service (LLS) in [1] for ad-hoc networks is based on a hierarchy of lattice points for destination nodes, published with locations of associated nodes. Lattice points can be queried for the desired location, with a query traversing a spiral path of lattice nodes increasingly distant from the source until it reaches the destination. An-

other way of choosing location servers is based on quorums. A set of hosts is chosen to be a $write$ quorum for a mobile client and is updated with the client's location. Another set is chosen to be a $read$ quorum and queried for the desired client location. Each $write$ and $read$ quorum has a nonempty intersection, guaranteeing that if a $read$ quorum is queried, the results will include the latest location of the client written to a $write$ quorum. In [48], a uniform quorum system is suggested, based on a virtual backbone of quorum representatives.

Location servers can also be chosen using a hash table. Some papers [51, 62, 82] use geographic locations as a repository for data. These use a hash to associate each piece of data with a region of the network and store the data at certain nodes in the region. This data can then be used for routing or other applications. The Grid location service (GLS) [62] maps each client C_p's id to some geographic coordinates x_p. A client C_p's location is then saved by clients located closest to the coordinates x_p.

The location managment scheme we present here is based on the hash table concept and built on top of the VSA layer and the Geocast service. VSAs and mobile clients are programmed to form a self-stabilizing, fault-tolerant distributed data structure for location management, where VSAs serve as home locations for mobile clients. Each client's id hashes to a VSA region, the client's home location, whose VSA is responsible for maintaining the location of the client. Whenever a VSA wants to locate a client node C_p, the VSA computes the home location of C_p by applying a predefined global hash function to C_p's id, and queries the region represented by the result of that hash for C_p's location.

In the rest of this chapter, we describe the service (Section 13.1) and properties of the service (Section 13.2), then describe a set of legal states of the service and properties of executions starting in those legal states (Section 13.3), and finally argue that our service is self-stabilizing (Section 13.4). As a wrap-up we also mention some possible extensions to this work.

```
  Signature:
2   Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^{≥0}
    Output vcast(⟨update, p, u, t⟩)_p, u ∈ U, t ∈ ℝ^{≥0}
4
  State:
6   analog clock ∈ ℝ^{≥0} ∪ {⊥}, initially ⊥
    reg ∈ U ∪ {⊥}, the current region, initially ⊥
8   hbTO ∈ ℕ, initially 0

10 Trajectories:
    evolve
12    d(clock) = 1
    stop when
14    Any precondition is satisfied.

Transitions:                                              16
Input GPSupdate(l, t)_p
Effect:                                                   18
  if reg ≠ region(l) ∨ clock ≠ t then
    clock ← t                                             20
    reg ← region(l)
    hbTO ← 0                                              22

Output vcast(⟨update, p, u, t⟩)_p                          24
Precondition:
  t = clock ∧ u = reg ≠ ⊥                                  26
  hbTO *ttl_hb ≤ clock ∨ hbTO *ttl_hb > clock + ttl_hb
Effect:                                                   28
  hbTO ← ⌊clock/ttl_hb⌋ + 1
```

Figure 13-1: Client $C^{HL}[ttl_{hb}]_p$ periodically sends region updates to its local VSA.

13.1 Location service specification

Our location service allows a VSA u to submit a query for a recent region of a client node p via a $\mathsf{HLquery}(p)_u$ action. It allows the region to receive a reply to this query indicating that p was recently in a region v though a $\mathsf{HLreply}(p, v)_u$ action, under certain conditions. In our implementation, called the *Home Location Service (HLS)*, we accomplish this using *home locations*. Recall that the home location of a client node p is the region whose VSA is periodically (at least every ttl_{hb} time) updated with p's region. The home locations are calculated with a hash function h, mapping a client's id to a VSA region, and is known to all VSAs. These home location VSAs can then be queried by other VSAs to determine a recent region of p.

The HLS implementation consists of two parts: a client-side portion and a VSA-side portion. C_p^{HL} is a subautomaton of client p that interacts with VSAs to provide HLS. It is responsible for notifying VSAs in its current and neighboring regions which region it is in.

For the VSA-side, V_u^{HL} is a subprogram of the VSA at region u that takes a request for some client node p's region, calculates p's home location using the hash function, and then sends location queries to the home location using Geocast. The home location subprogram at the receiving VSA responds with the region information it has for p, which is then output by V_u^{HL}. V_u^{HL} also is responsible both for informing the home location of each client p located in its region of p's region, and maintaining and answering queries for the regions of clients for which it is a home location.

1 **Signature**:
Input time$(t)_u$, $t \in \mathbb{R}^{\geq 0}$
3 **Input** vrcv$(\langle$update, $p, v, t\rangle)_u$, $p \in P$, $v \in U$, $t \in \mathbb{R}^{\geq 0}$
Input HLQuery$(p)_u$
5 **Input** georcv$(m)_u$, $m \in (\{$hlquery$\} \times P \times U)$
$\cup$ ({update, hlreply} $\times P \times U \times \mathbb{R}^{\geq 0}$)
7 **Output** geocast$(m, v)_u$, $v \in U$, $m \in (\{$hlquery$\} \times P \times \{u\})$
$\cup$ ({update, hlreply} $\times P \times U \times \mathbb{R}^{\geq 0}$)
9 **Output** HLreply$(p, v)_u$, $p \in P$, $v \in U$
Internal clean$_u$
11
State:
13 **analog** *clock*: $\mathbb{R}^{\geq 0} \cup \{\bot\}$, initially $\bot$
local, *lastreq*: $P \rightarrow \mathbb{R}^{\geq 0} \cup \{\bot\}$, initially $\bot$
15 *dir*, *lastLoc*: $P \rightarrow U \times \mathbb{R}^{\geq 0}$, initially null
req: $P \rightarrow Bool$, initially false
17 *answer*: $P \rightarrow 2^U$, initially $\emptyset$

19 **Trajectories:**
evolve
21 $\mathbf{d}(clock) = 1$
stop when
23 Any output precondition is satisfied
$\vee \exists p \in P$: $[lastreq(p) \leq clock$ -2(e+d) $dist(u, h(p))$ -ϵ
25 $\vee \exists \langle v, t\rangle = dir(p)$: $t \leq clock$ -ttl_{hb} -d -$(e + d)$ $dist(v', u)$ -ϵ
$\vee \exists \langle v, t\rangle = lastLoc(p)$: $t \leq clock$ -ttl_{hb} -d
27 -$(e + d)$ $(dist(v, h(p)) + dist(h(p), u))$ -ϵ]

29 **Transitions**:
Input time$(t)_u$
31 **Effect**:
if $clock \neq t \vee \exists p \in P$: $(local(p) \notin [clock$ -d, $clock) \cup \{\bot\}$
33 $\vee$ $lastreq(p) > clock \vee [req(p) \wedge lastreq(p) = \bot]$
$\vee$ $[\exists \langle v, t\rangle \in \{dir(p), lastLoc(p)\}$: $t \geq clock]$
35 $\vee$ $[\neg \exists \langle v, t\rangle = dir(p)$: $t \geq clock$ -ttl_{hb} -d -
$(e + d)$ $dist(v', u)$
$\wedge\, answer(p) \neq \emptyset] \vee [h(p) \neq u \wedge dir(p) \neq \bot]$)**then**
37 $clock \leftarrow t$
for each $p \in P$
39 $local(p)$, $lastreq(p) \leftarrow \bot$
$dir(p) \leftarrow null$
41 $req(p) \leftarrow$ **false**
$answer(p) \leftarrow \emptyset$
43
Input vrcv$(\langle$update, $p, v, t\rangle)_u$
45 **Effect**:
if $v = u \wedge t \in [clock$ -d, $clock)$ **then**
47 $local(p) \leftarrow t$

49 **Output** geocast$(\langle$update, $p, u, t\rangle, v)_u$
Precondition:
51 $local(p) \in [clock$ -d, $clock) \wedge v = h(p)$
Effect:
53 $local(p) \leftarrow \bot$

Input georcv$(\langle$update, $p, v, t\rangle)_u$
Effect: 56
if $h(p) = u \wedge t \in [clock$ -d -$(d + e)$ $dist(u, v)$, $clock)$
$\wedge$ $(dir(p) = null \vee [dir(p) = \langle v', t'\rangle \wedge t' < t])$ **then** 58
$dir(p) \leftarrow \langle v, t\rangle$
60
Input HLQuery$(p)_u$
Effect: 62
if $clock \neq \bot$**then**
$lastreq(p) \leftarrow clock$ 64
$req(p) \leftarrow$ **true**
66
Output geocast$(\langle$hlquery, $p, u\rangle, v)_u$
Precondition: 68
$clock \neq \bot \wedge req(p) =$ **true** $\wedge\, v = h(p)$
Effect: 70
$req(p) \leftarrow$ **false**
72
Input georcv$(\langle$hlquery, $p, v\rangle)_u$
Effect: 74
if $h(p) = u \wedge \exists \langle v', t\rangle = dir(p)$:
$t \in [clock$ -ttl_{hb} -d -$(e + d)$ $dist(v', u)$, $clock)$ **then** 76
$answer(p) \leftarrow answer(p) \cup \{v\}$
78
Output geocast$(\langle$hlreply, $p, v, t\rangle, v')_u$
Precondition: 80
$clock \neq \bot \wedge v' \in answer(p) \wedge u = h(p) \wedge dir(p) = \langle v, t\rangle$
Effect: 82
$answer(p) \leftarrow answer(p) - \{v'\}$
84
Input georcv$(\langle$hlreply, $p, v, t\rangle)_u$
Effect: 86
if $t \in [clock$-ttl_{hb}-d-$(e$+$d)(dist(v,h(p))$+$dist(h(p),u))$,$clock)$
$\wedge$ $[(\exists v' \in U$: $lastLoc(p) = \langle v', t'\rangle \wedge t' < t)$ 88
$\vee$ $lastLoc(p) = null]$ **then**
$lastLoc(p) \leftarrow \langle v, t\rangle$ 90

Output HLreply$(p, v)_u$ 92
Precondition:
$[\exists t \in [clock$-ttl_{hb}-d-$(e$+$d)(dist(v,h(p))$+$dist(h(p),u))$,$clock)$: 94
$lastLoc(p) = \langle v,t\rangle] \wedge lastreq(p) \geq clock-2(e$+$d)dist(u,h(p))$
Effect: 96
$lastreq(p) \leftarrow \bot$
98
Internal clean$_u$
Precondition: 100
$\exists p \in P$: $[lastreq(p) < clock$ -2$(e$+$d)$ $dist(u, h(p))$
$\vee \exists \langle v, t\rangle = dir(p)$: $t < clock$ -ttl_{hb} -d -$(e + d)$ $dist(v', u)$ 102
$\vee \exists \langle v, t\rangle = lastLoc(p)$: $t <$
$clock$ -ttl_{hb} -d -$(e + d)$ $(dist(v, h(p)) + dist(h(p), u))]$ 104
Effect:
for each $p \in P$ 106
if $lastreq(p) < clock$ -2$(e$+$d)$ $dist(u, h(p))$ **then**
$lastreq(p) \leftarrow \bot$ 108
if $\exists \langle v,t\rangle = dir(p)$:$t < clock$-$ttl_{hb}$-$d$-$(e$+$d)dist(v',u)$ **then**
$dir(p) \leftarrow \bot$ 110
if $\exists \langle v, t\rangle = lastLoc(p)$: $t < clock$ -ttl_{hb} -d
-$(e + d)$ $(dist(v, h(p)) + dist(h(p), u))$ **then** 112
$lastLoc(p) \leftarrow \bot$

Figure 13-2: VSA $V^{HL}[ttl_{hb}, h : P \rightarrow U]_u$ automaton.

The TIOA specification for the the individual clients is in Figure 13-1. The specification for the individual regions is in Figure 13-2. The complete service, HLS, is the composition of $\prod_{u \in U} Fail(V_u^{HL} \| V_u^{Geo} \| VBDelay_u)$, $\prod_{p \in P} Fail(C_p^{HL} \| VBDelay_p)$, and $RW \| VW \| Vbcast$. In other words, the service consists of a $Fail$-transformed automata at each region of the home location machine, geocast machines and $VBDelay$ machine; a $Fail$-transformed automata at each client of the geocast machine and $VBDelay$ machine; and $RW \| VW \| Vbcast$.

Just as with the geocast automata V_u^{Geo} in Chapter 12, we note that for each $u \in U$, $V_u^{HL} \| V_u^{Geo}$ is not technically a valid VSA since its external interface consists of non-vcast, vrcv, and time actions. However, we will later (in Chapter 14) compose this automaton with other automata and hide these actions to produce new automata that are VSAs. In the meantime we map refer to these almost-VSAs as VSAs, with the understanding that the technical detail will be resolved later.

Again, just as with the geocast service, the V_u^{HL} subprograms can be used in other VSA layer programs, as long as each V_u^{HL} is composed with other VSA subprograms that allow us to hide the HLQuery and HLreply actions. For example, we could define a $C - HLS$ service that allows clients to query for the region of other clients, and to subsequently receive replies. We could implement this service in the same way as we implemented the $C - Geocast$ service at the end of Section 12.1: have clients broadcast queries to and receive replies from their local region u's VSA subprogram for $C - HLS$, which in turn interacts with the region's V_u^{HL} subprogram to have those queries answered.

We now describe the pieces of the HLS service in more detail.

13.1.1 Client algorithm

The code executed by client p's C_p^{HL} is in Figure 13-1.

Clients receive GPSupdates every ϵ_{sample} time from the GPS automaton (lines 17-22), making them aware of their current region and the time. If a client's region or local clock changes as a result, the variable $hbTO$ is set to 0 (line 22), forcing the immediate send of an update message, with its id, current time and region information (lines 24-29). The client

also periodically (at every multiple of ttl_{hb} time) reminds its current VSA of its region by broadcasting an additional update message.

13.1.2 VSA algorithm

The code for automaton V_u^{HL} appears in Figure 13-2.

First, the VSA knows which clients are in its or neighboring regions through update messages. If a VSA vrcvs an update message from a client p claiming to be in its region (lines 44-47), the VSA sends an update message for p, with p's heartbeat timestamp and region, through Geocast to $h(p)$, the VSA home location of client p (lines 49-53).

When a VSA receives one of these update messages for a client p, it stores both the region indicated in the message as p's current region and the attached heartbeat timestamp in its dir table (lines 55-59). This location information for p is refreshed each time the VSA receives an update for client p with a newer heartbeat timestamp (line 58). Since a client sends an update message every ttl_{hb} time, which can take up to d time to arrive at and trigger its local VSA u to send an update message through Geocast, which can take $(e + d)dist(u, h(p))$ time to be delivered at the home location, an entry for client p indicating the client was in region u is erased by its home location if its timestamp is older than $ttl_{hb} + d + (e + d)dist(u, h(p))$ (lines 102 and 109-110).

The other responsibility of the VSA is to receive and respond to requests for location information on clients. A request for a client p's location comes in to region u via a HLquery$(p)_u$ input (line 61). This sets $lastreq(p)$, the time of the last query for p's location (used later to clean up expired queries), to the current time, and updates the flag $req(p)$ to true, indicating that a query should be sent to p's home location (lines 63-65). This triggers the geocast of a $\langle \text{hlquery}, p, u \rangle$ message to p's home location (lines 67-71). Any home location that receives such a message and has an unexpired entry for p's region responds with a hlreply to the querying VSA with the region and the timestamp of the information (lines 79-83).

If the querying VSA at u receives a hlreply for a client p with newer information than it currently has, it stores the attached region, v, and timestamp in $lastLoc(p)$ (lines 84-

90). This information stays in $lastLoc(p)$ until replaced with newer information or until the entry's timestamp is older than the maximum time for a client to have sent the next update, had the update received by its local VSA, and had the information propagated to its home location and from the home location to VSA u (lines 99, 103-104, and 111-113).

If there is an outstanding request for p's location (indicated by the condition that $lastreq(p) \geq clock - 2(e+d)dist(u, h(p))$ in line 95), the VSA performs a HLreply$(p, v)_u$ output and clears $lastreq(p)$, indicating that all outstanding queries for p's location are satisfied (lines 92-97). If, however, $2(e + d)dist(u, h(p))$ time passes since a request for p's region was received and there is no entry for p's region, $lastreq(q)$ is just erased (lines 99, 101, and 107-108), indicating that the query has expired.

13.2 Properties of executions of the location service

A location service answers queries for the locations of clients. A VSA u can submit a query for a recent region of client node p via a HLquery$(p)_u$ action. If p's home location can be communicated with and p has been in the system for a sufficient amount of time, the service responds within bounded time with a recent region location v of p through a HLreply$(p, v)_u$ action.

More formally, we say that a process p is *findable* at a time t if there exists a time t_{sent} such that:

1. $t_{sent} \mod ttl_{hb} = 0$ and process p has been alive since time $t_{sent} - \epsilon_{sample}$.
2. For each $u \in \{reg^-(p, t_{sent}), reg^+(p, t_{sent})\}$, $t_{sent} + d + (e + d)dist(u, h(p)) < t$.
3. For each $t' \in [t_{sent}, t]$ and $v \in \{reg^-(p, t'), reg^+(p, t')\}$, there exists at least one shortest path from v to $h(p)$ of regions that are nonfailed and have $clock$ values equal to the real-time for the interval $[t', t' + (e + d)dist(v, h(p))]$.

(Notice that this amounts to saying that a process is findable if we can be assured that its home location will have some information on the whereabouts of the process.)

We say that a HLQuery by a region u for a process p at time t is *serviceable* if:

1. Process p is findable at time t' for each $t' \in [t, t + (e + d)dist(u, h(p))]$.

2. There exists at least one shortest path from u to $h(p)$ of regions that are nonfailed and have $clock$ values equal to the real-time for the interval $[t, t + 2(e + d)dist(u, h(p))]$.

Then we can show the following result:

Lemma 13.1 *The HLS service guarantees that in each execution α of HLS, there exists a function mapping each* HLreply$(p, v)_u$ *event to a* HLQuery$(p)_u$ *event such that the following hold:*

1. Integrity*: If a* HLreply$(p, v)_u$ *event π is mapped to a* HLQuery$(p)_u$ *event π', then π' occurs before π.*

2. Bounded-time reply*: If a* HLreply$(p, v)_u$ *event π is mapped to a* HLQuery$(p)_u$ *event π' where π' occurs at time t, then event π occurs in the interval $[t, t + 2(e + d)dist(u, h(p))]$.*

3. Reliable reply*: This guarantees that a query will be answered if it is serviceable: If a* HLQuery$(p)_u$ *event π' occurs at time t, $\alpha.ltime > t + 2(e + d)dist(u, h(p))$, and π' is serviceable, then there exists a* HLreply$(p, v)_u$ *event π such that π occurs at some time $t' \in [t, t + 2(e + d)dist(u, h(p))]$.*

4. Reliable information*: If a* HLreply$(p, v)_u$ *event occurs at some time t, then there exists a time $t' \in [t - ttl_{hb} - d - (e + d)(dist(v, h(p)) + dist(h(p), u)), t]$ such that $v \in \{reg^-(p, t'), reg^+(p, t')\}$.*

Proof sketch: It is easy to define the mapping from HLQuery to HLreply events described above as follows: For each HLreply$(p, v)_u$ event, there is some time $t \neq \perp$ such that $t = lastreq(p)_u$ (line 95). We map the HLreply event to the first HLQuery$(p)_u$ event that occurs at time t.

It is very easy to check that the first two properties hold. To see that Reliable reply holds, we note that for a HLQuery$(p)_u$ event the properties of the underlying Geocast service make the property easy to check. (Due to properties of Geocast, the only thing

that really needs checking is that if p is findable, then when any $\langle \mathsf{hlquery}, p, u\rangle$ message sent because of the HLQuery is received by p's home location, the home location will have information on p's location. We can see that this holds because if p is findable, the properties of Geocast ensure that some recent-enough update message about p will have been received by p's home location.)

To see that the Reliable information property holds, assume that a $\mathsf{HLreply}(p, v)_u$ event π occurs at some time t. We must show that there exists a time $t' \in [t - ttl_{hb} - d - (e + d)(dist(v, h(p)) + dist(h(p), u)), t]$ such that $v \in \{reg^-(p, t'), reg^+(p, t')\}$. By the precondition for the HLreply event on lines 94-95, we know that there exists a pair $\langle v, t''\rangle$ equal to $lastLoc(p)$ such that $t'' \geq t - ttl_{hb} - d - (e + d)(dist(v, h(p)) + dist(h(p), u))$. We now argue that t'' satisfies the properties of the t' we are looking for. The only way that $lastLoc(p)$ is set to $\langle v, t''\rangle$ is by the receipt of a $\langle \mathsf{hlreply}, p, v, t''\rangle$ message (lines 85-90). Such a message is only sent by p's home location if the home location's $dir(p)$ is set to $\langle v, t''\rangle$ (lines 79-81). The home location's $dir(p)$ is only set to $\langle v, t''\rangle$ by the receipt of an $\langle \mathsf{update}, p, v, t''\rangle$ tuple (lines 55-59). Such an update tuple is only sent by the region v if its $local(p)$ is set to t'' (lines 49-51). Its $local(p)$ is only set to t'' if it received an $\langle \mathsf{update}, p, v, t''\rangle$ message through the Vbcast service (lines 44-47). Such a message must have been sent by a process p at time t. Since the message is only sent by the process p if its latest region update by time t was for region v, we have our result. ■

13.3 Legal sets

Here we describe a legal set of HLS by describing a sequence of five legal sets, each a subset of the prior. Recall from Lemma 3.13 that a legal set of states for a TIOA is one where each closed execution fragment starting in a state in the set ends in a state in the set. We break the definition of the legal set up into multiple legal sets in order to simplify the proof reasoning and more easily prove stabilization later, in Section 13.4. Because the proofs in this section are routine, we omit them. At the end of this section, we discuss properties of execution fragments of HLS that start in our set of legal states.

13.3.1 Legal set L^1_{hls}

The first set of legal states describes some properties that are locally checkable at a region or client and that become true at an alive VSA at the time of the first time input for the VSA and GPSupdate input at a client, assuming the underlying $GeoCast$ system is in a legal state.

Definition 13.2 *Let L^1_{hls} be the set of states x of HLS where all of the following hold:*

1. $x\lceil X_{GeoCast} \in L^2_{geo}$.
   This says that the state restricted to the variables of $GeoCast$ is a legal state of $GeoCast$.

2. *For each $p \in P : \neg failed_p$ (nonfailed client):*

   (a) $clock_p \neq \bot \Rightarrow [clock_p = now \wedge reg_p = reg(p)]$.
   This says that if the local clock is not $\bot$, then it is set to the current real-time and reg_p is p's current region.

   (b) $[hbTO * ttl_{hb} = clock_p + ttl_{hb} \wedge \langle \text{update}, p, reg_p, clock_p\rangle \notin to_send^-_p to_send^+_p] \Rightarrow \langle\langle \text{update}, p, reg_p, clock_p\rangle, reg_p, clock_p, P \cup U\rangle \in vbcastq$.
   This says that if $hbTO$ indicates that the client should have just sent an update and there is no such message in the client's $VBDelay$, then the update has already been propagated to $Vbcast$.

   (c) $[\exists q \in P, u \in U, t \in \mathbb{R}^{\geq 0} : \langle \text{update}, q, u, t\rangle \in to_send^-_p to_send^+_p] \Rightarrow [q = p \wedge t = now \wedge u \in \{reg^-(p, now), reg^+(p, now)\}]$.
   This says that if an update message is in one of a client's $VBDelay$ queues, then the message correctly indicates a region that the client has been in at this time.

3. *For each $u \in U : \neg failed_u \wedge clock_u \neq \bot$ (nonfailed VSA that has received a* time *input):*

(a) $clock_u = now$.

This says that the local clock should be equal to the real-time.

(b) $\neg\exists p \in P : (local(p) \notin [clock - d, clock) \cup \bot \vee lastreq(p) > clock \vee [req(p) \wedge lastreq(p) = \bot] \vee [\exists\langle v, t\rangle \in \{dir(p), lastLoc(p)\} : t \geq clock] \vee [\ \exists\langle v, t\rangle = dir(p) : t \geq clock - ttl_{hb} - d - (e + d)dist(v', u) \wedge answer(p) \neq \emptyset] \vee [h(p) \neq u \wedge dir(p) \neq \bot])$.

This just says that a non-failed VSA's state must satisfy a litany of local consistency conditions, none of which is very interesting.

It is trivial to check that L^1_{hls} is a legal set for HLS.

Lemma 13.3 *L^1_{hls} is a legal set for HLS.*

13.3.2 Legal set L^2_{hls}

The second set of legal states describes some properties that hold after any spurious VSA messages are broadcast and spurious $Vbcast$ messages are delivered.

Definition 13.4 *Let L^2_{hls} be the set of states x of HLS where all of the following hold:*

1. $x \in L^1_{hls}$.

   This says that L^2_{hls} is a subset of L^1_{hls}.

2. *For each* $\langle\langle \mathsf{update}, p, u, t\rangle, q, v, t', P'\rangle \in vbcastq$:

   $t' \geq now - d \Rightarrow [q = p \wedge t = t' \wedge u \in \{reg^-(p, t), reg^+(p, t)\}]$.

   This says that any update *tuple in $vbcastq$ sent in the last d time must correctly indicate a region of the sender at the time the message was sent.*

3. *For each $u \in U : \neg failed_u$ (nonfailed VSA):*

   (a) $\not\exists\langle\langle \mathsf{update}, p, v, t\rangle, t'\rangle \in to_send_u$.

   This says that a VSA should not vcast *an* update *tuple. (VSAs only* geocast update *tuples.)*

(b) For each $p \in P : local(p) = t \neq \bot \Rightarrow u \in \{reg^-(p,t), reg^+(p,t)\}$*.*

This says that if $local(p)$ *is set to* t*, then the VSA's region is a region of the process* p *at time* t*.*

(c) For each $v, v' \in U, p \in P, t \in \mathbb{R}^{\geq 0} : [ledger(\langle\langle \mathsf{update}, p, v, t\rangle, u, v', now\rangle) \neq null \vee \langle\langle \mathsf{geocast}, \langle \mathsf{update}, p, v, t\rangle, u, v', now\rangle, rtimer_u\rangle \in to_send_u] \Rightarrow [u = v \wedge v' = h(p) \wedge u \in \{reg^-(p,t), reg^+(p,t)\}]$*.*

This says that if an update *message for* p *has been* geocast *but not yet been turned over to* $Vbcast$*, then it is being* geocast *to the home location of the process and correctly indicates one of the regions of* p *at the time* t *included in the message.*

(d) For each $p \in P, \langle v, t\rangle = lastLoc(p) : t \geq clock_u - d : \exists\langle\langle \mathsf{geocast}, \langle \mathsf{hlreply}, p, v, t\rangle, v', u, t'\rangle, v'', t'', P'\rangle \in vbcastq : t'' \geq t$*.*

This says that if $lastLoc(p)$ *is set to some* $\langle v, t\rangle$ *where* $t \geq now - d$*, then there exists a* geocast *of an* hlreply *tuple no older than* t *that indicates that* v *is a region of* p *at time* t*.*

4. *For each* $\langle\langle \mathsf{geocast}, \langle \mathsf{update}, p, v, t\rangle, u, v', t'\rangle, u', now, P \cup U\rangle$ *in* $vbcastq$*:*

$$[t' \in (t, t+d] \wedge u = v = u' \wedge v' = h(p) \wedge u \in \{reg^-(p,t), reg^+(p,t)\}].$$

This says that any update *tuple for a process* p *and time* t *that has just been* geocast *and whose record is in* $Vbcast$ *correctly indicates a region of the process* p *at time* t*. It also says that the message is being* geocast *to the process's home location.*

For the sake of brevity and reader sanity, we do not include the proof of the following lemma here. The proof is a tedious but not difficult case analysis, based on the actions and trajectories of the HLS system.

Lemma 13.5 L^2_{hls} *is a legal set for* HLS*.*

13.3.3 Legal set L^3_{hls}

The third set of legal states describes some properties that hold after any spurious geocast messages are delivered.

Definition 13.6 *Let L^3_{hls} be the set of states x of HLS where all of the following hold:*

1. $x \in L^2_{hls}$.
   This says that L^3_{hls} is a subset of L^2_{hls}.

2. *For each* $\langle \text{geocast}, \langle\langle \text{update}, p, v, t\rangle, u, v', t'\rangle, u', t'', P'\rangle$ *in* $vbcastq$*:* $t'' \geq now - (e + d)D \Rightarrow [t' \in (t, t + d] \wedge u = v = u' \wedge v' = h(p) \wedge u \in \{reg^-(p, t), reg^+(p, t)\}]$.
   This says that a geocast *of an* update *for a process p at time t that was passed to $Vbcast$ at some time $t'' \geq now - (e + d)D$ was sent to p's home location by the VSA at a region of the process at time t.*

This lemma is also easy to check:

Lemma 13.7 *L^3_{hls} is a legal set for HLS.*

13.3.4 Legal set L^4_{hls}

The fourth set of legal states describes some properties that hold after any bad location information stored at home locations of processes is cleaned up.

Definition 13.8 *Let L^4_{hls} be the set of states x of HLS where all of the following hold:*

1. $x \in L^3_{hls}$.
   This says that L^4_{hls} is a subset of L^3_{hls}.

2. *For each* $\langle \text{geocast}, \langle\langle \text{update}, p, v, t\rangle, u, v', t'\rangle, u, t'', P'\rangle$ *in* $vbcastq : t'' \geq now - d - ttl_{hb} - 2(e+d)D$*:* $[t' \in (t, t+d] \wedge u = v \wedge v' = h(p) \wedge u \in \{reg^-(p, t), reg^+(p, t)\}]$.
   This is similar to property 2 of L^3_{hls}, only extended for $t'' \geq now - d - ttl_{hb} - 2(e+d)D$.

3. *For each* $u \in U : \neg failed_u$*:* $\forall p \in P : \forall \langle v, t\rangle = dir(p) : t \geq clock_u - ttl_{hb} - d - (e+d)dist(v, u) \Rightarrow \exists \langle \text{geocast}, \langle\langle \text{update}, p, v, t\rangle, v, u, t'\rangle, v, t'', P'\rangle \in vbcastq : t'' \geq now - d - ttl_{hb} - (e + d)D$.
   This says that at a nonfailed VSA, if the VSA is storing the location of a process p as region v at time t, then if $t \geq clock_u - ttl_{hb} - d - (e + d)dist(v, u)$, there was a geocast *of an* update *tuple indicating the same region and time information.*

4. *For each* $u \in U : \neg failed_u$*,* $v, v' \in U, p \in P, t \in \mathbb{R}^{\geq 0} : [ledger(\langle\langle \mathsf{hlreply}, p, v, t\rangle, u, v', now\rangle) \neq null \vee \langle\langle \mathsf{geocast}, \langle \mathsf{hlreply}, p, v, t\rangle, u, v', now\rangle, rtimer_u\rangle \in to_send_u] \Rightarrow [u = h(p) \wedge v \in \{reg^-(p,t), reg^+(p,t)\}]$.

   This says that if an hlreply *message for a process* p *has been* geocast *but not yet turned over to* $Vbcast$*, then the VSA is the home location for* p *and the attached region* v *is a region of* p *at time* t*.*

5. *For each* $\langle \mathsf{geocast}, \langle\langle \mathsf{hlreply}, p, v, t\rangle, u, v', t'\rangle, u', now, P \cup U\rangle$ *in* $vbcastq$*:* $[u = h(p) \wedge v \in \{reg^-(p,t), reg^+(p,t)\}]$.

   This says that any geocast *of an* hlreply *that has just been turned over to* $Vbcast$ *correctly names a region that a process* p *was in at a time* t *and was sent by* p*'s home location.*

6. *For each* $u \in U : \neg failed_u$*, for each* $p \in P$*,* $\langle v, t\rangle = lastLoc(p)$*:*
   $t \geq clock_u - d - ttl_{hb} - (e + d)D \Rightarrow \exists \langle \mathsf{geocast}, \langle\langle \mathsf{hlreply}, p, v, t\rangle, h(p), u, t'\rangle, h(p), t'', P'\rangle \in vbcastq : [t'' \geq t \wedge v \in \{reg^-(p,t), reg^+(p,t)\}]$.

   This says that if $lastLoc(p)$ *is set to some* $\langle v, t\rangle$ *where* $t \geq now - d - ttl_{hb} - (e+d)D$*, then there exists a* geocast *of an* hlreply *tuple no older than* t *that indicates that* v *is a region of* p *at time* t*. In addition,* v *was a region of* p *at time* t*.*

The proof of the following lemma is again omitted because it is routine.

Lemma 13.9 *L^4_{hls} is a legal set for HLS.*

13.3.5 Legal set L^5_{hls}

The fifth set of legal states describes some properties that hold after any bad location information stored at location queriers is cleaned up.

Definition 13.10 *Let L^5_{hls} be the set of states x of HLS where all of the following hold:*

1. $x \in L^4_{hls}$.

   This says that L^5_{hls} is a subset of L^4_{hls}.

2. *For each* $\langle \mathsf{geocast}, \langle\langle \mathsf{hlreply}, p, v, t\rangle, u, v', t'\rangle, u, t'', P'\rangle$ *in* $vbcastq$*:* $t'' \geq clock_u - (e+d)D \Rightarrow v \in \{reg^-(p,t), reg^+(p,t)\}$.

   This is similar to property 5 of L^4_{hls}*, only extended for* $t'' \geq clock_u - (e+d)D$*, rather than just* $t'' = now$*.*

3. *For each* $u \in U : \neg failed_u$ *and for each* $p \in P : \langle v, t\rangle = lastLoc_u(p) \wedge t \geq clock_u - ttl_{hb} - d - 2(e+d)D : \exists \langle \mathsf{geocast}, \langle\langle \mathsf{hlreply}, p, v, t\rangle, h(p), u, t'\rangle, h(p), t'', P'\rangle \in vbcastq : [t'' \geq t \wedge v \in \{reg^-(p,t), reg^+(p,t)\}]$.

   This is similar to property 6 of L^4_{hls}*, only extended for* $t'' \geq clock_u ttl_{hb} - d - 2(e+d)D$*.*

It is trivial to see that since the second two properties are simply properties of L^4_{hls} observed for longer periods of time, the following result will follow:

Lemma 13.11 L^5_{hls} *is a legal set for* HLS*.*

Properties of execution fragments starting in L^5_{hls}

As with the Geocast service, we can describe properties of execution fragments of HLS that start in L^5_{hls} as properties of executions of HLS, as described in Section 13.2. As before, the difference is in the mapping of some subset of HLreply events that occur towards the beginning of the execution fragment.

More formally, we can say the following:

Lemma 13.12 HLS *guarantees that for an execution fragment* α *starting in* L^5_{hls}*, there exists a subset* Π *of the* HLreply *events in* α *such that:*

1. *There exists a function mapping each* HLreply *event in* Π *to a* HLquery *event such that the four properties (Integrity, Bounded time reply, Reliable reply, and Reliable information) hold.*

2. *For every* $\mathsf{HLreply}(p)_u$ *event* π *not in* Π *where* π *occurs at some time* t*, it must be the case that* $t - \alpha.fstate(now) \leq 2(e+d)dist(u, h(p))$*.*

This concept and proof is similar to the material in Section 12.3.2, where we described the properties of execution fragments of Geocast as a variant of the properties of executions of Geocast, adjusting for a subset of receive events towards the beginning of a fragment.

13.4 Self-stabilization

We've seen that L^5_{hls} is a legal set for HLS. Here we show that $\prod_{u \in U} Fail(VBDelay_u \| V^{Geo}_u \| V^{HL}_u) \prod_{p \in P} Fail(VBDelay_p \| C^{HL}_p)$ self-stabilizes to L^5_{hls} relative to $R(RW \| VW \| Vbcast)$ (Theorem 13.19), meaning that if certain program portions of the implementation are started in an arbitrary state and run with $R(RW \| VW \| Vbcast)$, the resulting execution eventually gets into a state in L^5_{hls}. Using Theorem 13.19, we then conclude that after an execution of HLS has stabilized, the execution fragment from the point of stabilization on satisfies the properties described in Section 13.3.5.

The proof of the main stabilization result for the chapter, Theorem 13.19, breaks stabilization down into two large phases, corresponding to stabilization of the lower level Geocast service, followed by stabilization of the HLS service assuming that Geocast has stabilized. We have seen that $GeoCast$ stabilizes to the set of legal states L^2_{geo} in Section 12.4. What we need to show for Theorem 13.19 is that, starting from a set of states where $GeoCast$ is already stabilized, HLS stabilizes to L^5_{hls} (Lemma 13.18). We do this in five stages, one for each of the legal sets described in Section 13.3. The first stage starts from a state where $GeoCast$ is already stabilized and ends up in the first legal set. The second stage starts in the first legal set and ends up in the second, etc.

The first lemma describes the first stage of HLS stabilization, to legal set L^1_{hls}. It says that within t^1_{hls} time of $GeoCast$ stabilizing, where $t^1_{hls} > \epsilon_{sample}$, the system ends up in a state in L^1_{hls}.

Lemma 13.13 *Let t^1_{hls} be any t such that $t > \epsilon_{sample}$. $frags_{HLS}^{\{x | x \lceil X_{GeoCast} \in L^2_{geo}\}}$ stabilizes in time t^1_{hls} to $frags_{HLS}^{L^1_{hls}}$.*

Proof sketch: To see this result, just consider the first time after each node has received a

time or GPSupdate input, which takes at most ϵ_{sample} time to happen. ■

The next lemma describes the second stage of HLS stabilization. It shows that starting from a state in L^1_{hls}, HLS ends up in a state in L^2_{hls} within t^2_{hls} time, where t^2_{hls} is any time greater than $2e + d$.

Lemma 13.14 *Let t^2_{hls} be any t such that $t > 2e + d$. $frags^{L^1_{hls}}_{HLS}$ stabilizes in time t^2_{hls} to $frags^{L^2_{hls}}_{HLS}$.*

Proof: By Lemma 3.21, we just need to show that for any length-t^2_{hls} prefix α of an element of $frags^{L^1_{hls}}_{HLS}$, $\alpha.lstate$ is in L^2_{hls}. We examine each property of L^2_{hls}.

By Lemma 13.13, since the first state of α is in L^1_{hls}, we know that property 1 of L^2_{hls} holds in each state of α.

For property 2 notice that for each update message added for the first time to one of a client's to_send queue and then propagated to $Vbcast$, the property will hold and will continue to hold thereafter. Hence, the only thing we need to worry about are the messages already in a to_send queue or already in $Vbcast$ in $\alpha.fstate$. However, after d time elapses from the start of α, the property will be trivially true.

For property 3, we consider each part. Property 3(a) will hold after at most e time, the time it takes for any such errant messages in $\alpha.fstate$ to be propagated out to $Vbcast$. Property 3(b) will hold after at most d time after property 3(a) holds (giving any messages with bad location information to be received and then removed from $local$ through the geocast of an update). Property 3(c) will hold within any non-0 time after property 3(b) holds, as each new geocast of an update will use location information that is correct. Property 3(d)

For property 4 notice that for each geocast tuple of an update message added for the first time to a to_send queue after property 3(b) holds (which takes up to $e + d$ time) and then propagated within e time to $vbcastq$, the property will hold and continue to hold as the message makes its way through the system. The only thing we need to consider are the tuples that are already in a to_send queue in $\alpha.fstate$. In the worst case, such a tuple takes e time to be placed in $vbcastq$, and any non-0 time afterwards to have its $Vbcast$ timestamp no longer be the current time. ■

For the third stage of HLS stabilization, the next lemma shows that starting from a state in L^2_{hls}, HLS ends up in a state in L^3_{hls} within t^3_{hls} time, where t^3_{hls} is any time greater than $(e + d)D$.

Lemma 13.15 *Let t^3_{hls} be any t such that $t > (e+d)D$. (Recall D is the hop count diameter of the network.) $frags^{L^2_{hls}}_{HLS}$ stabilizes in time t^3_{hls} to $frags^{L^3_{hls}}_{HLS}$.*

Proof: By Lemma 3.21, we just need to show that for any length-t^3_{hls} prefix α of an element of $frags^{L^2_{hls}}_{HLS}$, $\alpha.lstate$ is in L^3_{hls}. We examine each property of L^3_{hls}.

By Lemma 13.14, since the first state of α is in L^2_{hls}, we know that property 1 of L^3_{hls} holds in each state of α.

For property 2, notice that by property 4 of L^2_{hls} we have that all geocast tuples of update messages added to $vbcastq$ in α will satisfy the property and continue to do so. After $(e + d)D$ time has passed, we will have that the property holds for all such tuples broadcast within the prior $(e + d)D$ time. ■

The next lemma, for the fourth stage of HLS stabilization, shows that starting from a state in L^3_{hls}, HLS ends up in a state in L^4_{hls} within t^4_{hls} time, where t^4_{hls} is any time greater than $d + ttl_{hb} + (e + d)D$.

Lemma 13.16 *Let t^4_{hls} be any t such that $t > d + ttl_{hb} + (e + d)D$. $frags^{L^3_{hls}}_{HLS}$ stabilizes in time t^4_{hls} to $frags^{L^4_{hls}}_{HLS}$.*

Proof: By Lemma 3.21, we just need to show that for any length-t^4_{hls} prefix α of an element of $frags^{L^3_{hls}}_{HLS}$, $\alpha.lstate$ is in L^4_{hls}. We examine each property of L^4_{hls}.

By Lemma 13.15, since the first state of α is in L^3_{hls}, we know that property 1 of L^4_{hls} holds in each state of α. Property 2 is easy to see due to its similarity to property 2 of L^3_{hls}.

For property 3, notice that at the beginning of α, the newest values of t in a dir tuple is less than $\alpha.fstate(now)$. After t^4_{hls} time passes, these entries will be expired and won't affect the property. This means that all we have to check is that whenever a dir entry is updated in α, it satisfies the property. This is obvious since such an update only occurs through the georcv of an update message, which can only happen if property 3 holds.

For property 4, notice that any new hlreply tuple that is added to the $ledger$ or added to $VBDelay$ after property 3 holds will satisfy property 4. Similarly, for property 5, any new hlreply tuple added to $vbcastq$ after property 4 holds will satisfy property 5.

For property 6, notice that at the beginning of α, the newest values of t in a $lastLoc$ tuple is less than $\alpha.fstate(now)$. After t^4_{hls} time passes, those entries still in $lastLoc$ will be timestamped with values less than those of concern to the property. This means that all we have to check is that any additions or updates to $lastLoc$ satisfy the property. Since such changes only occur through the georcv of an hlreply, we just need to verify that any such message that arrives with the wrong region for p at some time has a timestamp that is older than t^4_{hls}. This follows from the fact that any hlreply sent in α with bad information must be using information timestamped from before α (by property 2 of L^3_{hls}). ■

For the fifth stage of HLS implementation, the next lemma shows that starting from a state in L^4_{hls}, HLS ends up in a state in L^5_{hls} within t^5_{hls} time, where t^5_{hls} is any time greater than $(e + d)D$.

Lemma 13.17 *Let t^5_{hls} be any t such that $t > (e + d)D$. $frags^{L^4_{hls}}_{HLS}$ stabilizes in time t^5_{hls} to $frags^{L^5_{hls}}_{HLS}$.*

The proof of this lemma is simple for the same reason that the proof that L^5_{hls} is trivial; the property is a longer-interval version of properties that we already know hold.

We now have all of the pieces of reasoning for the five stages of the second phase of HLS stabilization. (Recall that the second phase of HLS stabilization occurs after $GeoCast$ has stabilized, corresponding to $GeoCast$ state being in the set L^2_{geo}.) We then combine this reasoning from Lemmas 13.13-13.17 to show that the second phase of stabilization of HLS takes t'_{hls} time, $t'_{hls} > \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e + d)D$, to stabilize:

Lemma 13.18 *Let t'_{hls} be any t such that $t > \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e + d)D$. Then $frags^{\{x | x\lceil X_{GeoCast} \in L^2_{geo}\}}_{HLS}$ stabilizes in time t'_{hls} to $frags^{L^5_{hls}}_{HLS}$.*

Proof: The result follows from the application of Lemma 3.7 on the five lemmas (Lemmas 13.13-13.17) above.

Let t' be $(t'_{hls} - (\epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D))/5$. Then let t^1_{hls} be $t' + \epsilon_{sample}$, t^2_{hls} be $t' + 2e + d$, t^3_{hls} be $t' + (e + D)D$, t^4_{hls} be $t' + d + ttl_{hb} + (e + d)D$, and t^5_{hls} be $t' + (e + d)D$. (These terms are chosen so as to satisfy the constraints that $t^1_{hls} > \epsilon_{sample}$, $t^2_{hls} > 2e + d$, etc.)

Let B_0 be $frags_{HLS}^{\{x|x\lceil X_{GeoCast} \in L^2_{geo}\}}$, B_1 be $frags_{HLS}^{L^1_{hls}}$, B_2 be $frags_{HLS}^{L^2_{hls}}$, B_3 be $frags_{HLS}^{L^3_{hls}}$, B_4 be $frags_{HLS}^{L^4_{hls}}$, and B_5 be $frags_{HLS}^{L^5_{hls}}$ in Lemma 3.7. Let t_1 be t^1_{hls}, t_2 be t^2_{hls}, t_3 be t^3_{hls}, t_4 be t^4_{hls}, and t_5 be t^5_{hls} in Lemma 3.7. Then by Lemma 3.7 and Lemmas 13.13-13.17, we have that $frags_{HLS}^{\{x|x\lceil X_{GeoCast} \in L^2_{geo}\}}$ stabilizes in time $t^1_{hls} + t^2_{hls} + t^3_{hls} + t^4_{hls} + t^5_{hls}$ to $frags_{HLS}^{L^5_{hls}}$.

Since $t'_{hls} = t^1_{hls} + t^2_{hls} + t^3_{hls} + t^4_{hls} + t^5_{hls}$, we conclude that $frags_{HLS}^{\{x|x\lceil X_{GeoCast} \in L^2_{geo}\}}$ stabilizes in time t'_{hls} to $frags_{HLS}^{L^5_{hls}}$. ■

Using this and our prior result on $GeoCast$ stabilization (Theorem 12.9) we can now finally show the main stabilization result of this chapter. The proof of the result breaks down the self-stabilization of HLS into two phases, the first being where $GeoCast$ stabilizes, and the second being where the remaining pieces of HLS stabilize.

Theorem 13.19 *$\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL}) \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL})$ self-stabilizes in t_{hls} time, $t_{hls} > t_{geo} + \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e + d)D$, to L^5_{hls} relative to $R(RW \| VW \| Vbcast)$.*

Proof: For brevity, we will use $execs_{U-HLS}$ to refer to $execs_{U(\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL}) \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL})) \| R(RW \| VW \| Vbcast)}$. We must show that $execs_{U-HLS}$ stabilizes in time t_{hls} to $frags^{L^5_{hls}}_{\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL}) \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL}) \| R(RW \| VW \| Vbcast)}$. By Corollary 3.11, $frags^{L^5_{hls}}_{\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL}) \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL}) \| R(RW \| VW \| Vbcast)}$ is the same as $frags_{HLS}^{L^5_{hls}}$. This means that we must show that $execs_{U-HLS}$ stabilizes in time t_{hls} to $frags_{HLS}^{L^5_{hls}}$. The result follows from the application of transitivity of stabilization (Lemma 3.6) on the two phases of HLS stabilization.

For the first phase, we note that by Theorem 12.9, $execs_{U-HLS}$ stabilizes in time t_{geo} to $frags_{HLS}^{\{x|x\lceil X_{GeoCast} \in L^2_{geo}\}}$.

For the second phase, let t'_{hls} be $t_{hls} - t_{geo}$. Since $t_{hls} > t_{geo} + \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$, this implies that $t'_{hls} > \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$. By Lemma 13.18, we have that $frags_{HLS}^{\{x|x\lceil X_{GeoCast} \in L^2_{geo}\}}$ stabilizes in time t'_{hls} to $frags_{HLS}^{L^5_{hls}}$.

Taking B to be $execs_{U-HLS}$, C to be $frags_{HLS}^{\{x|x\lceil X_{GeoCast} \in L^2_{geo}\}}$, and D to be $frags_{HLS}^{L^5_{hls}}$ in Lemma 3.6, we have that $execs_{U-HLS}$ stabilizes in time $t_{geo} + t'_{hls}$ to $frags_{HLS}^{L^5_{hls}}$.

Since $t_{hls} = t_{geo} + t'_{hls}$, we conclude that $\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL}) \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL})$ self-stabilizes in t_{hls} time to L^5_{hls} relative to $R(RW \| VW \| Vbcast)$. ■

With Lemma 13.12, this allows us to conclude that after an execution of HLS has stabilized, the execution fragment from that point on satisfies the properties in Section 13.3.5:

Lemma 13.20 *Let t_{hls} be any t such that $t > t_{geo} + \epsilon_{sample} + ttl_{hb} + 2e + 2d + 3(e+d)D$. Then $execs_{U(\prod_{u \in U} Fail(VBDelay_u \| V_u^{Geo} \| V_u^{HL}) \prod_{p \in P} Fail(VBDelay_p \| C_p^{HL})) \| R(RW \| VW \| Vbcast)}$ stabilizes in time t_{hls} to a set $\mathcal{A}$ of execution fragments such that for each $\alpha \in \mathcal{A}$, there exists a subset Π of the* HLreply *events in α such that:*

1. *There exists a function mapping each* HLreply *event in Π to a* HLquery *event such that the four properties (Integrity, Bounded time reply, Reliable reply, and Reliable information) hold.*

2. *For every* HLreply$(p)_u$ *event π not in Π where π occurs at some time t, it must be the case that $t - \alpha.fstate(now) \leq 2(e+d)dist(u, h(p))$.*

13.5 Extensions

Here we briefly describe some possible extensions to our HLS algorithm:

Multiple home locations: In order for our scheme to tolerate crash failures of a limited number of VSAs, each mobile client id could map to a set of VSA home locations; the hash function would return a sequence of region ids as the home locations. We could use any hash function that provides a sequence of region identifiers; one possibility is a *permutation*

hash function, where permutations of region ids are lexicographically ordered and indexed by client id. A version of the home location service was presented in [37] that used this idea.

Randomized asymmetric quorums: It is possible to have asymmetric updates and queries, such as with local updates to close-by VSAs and uniformly selected VSAs or vice versa (the expected number of VSAs that are required to be updated and queried is small, as proved in [68]). Instead of using a predefined set to query, one might use a randomized scheme based on [68], where a random set of regions is chosen for updating and inquiring about the location of a client node. Moreover, we could enhance the scheme in [68] by using a predefined set for location updates (such as the close-by regions) and random set for location queries (or vice versa).

Attribute queries: There are scenarios in which one would like to query for client nodes with certain attributes in a geographic area (e.g., a search for a medical doctor that is currently near by). Our scheme supports such queries in a natural way: Attributes can hash to home locations that store tables of clients with the attribute, and their locations. Clients searching for another nearby client with some attribute could then have a local VSA query home locations for the attribute, and select a nearby client from the list that is returned.

Chapter 14

End-to-end Routing

One basic, but often difficult to provide, service in mobile networks is end-to-end routing. We describe a self-stabilizing algorithm over the VSA layer to provide a mobile client end-to-end routing service. This service is built on prior geocast and location management services in such a way that the resulting application remains self-stabilizing.

Our self-stabilizing implementation of a mobile client end-to-end communication service is simple, given the geocast and home location services. A client sends a message to another client by forwarding the message to its local VSA, which then uses the home location service to discover the destination client's region and forwards the message to that region using the geocast service.

In the rest of this chapter, we describe the service (Section 14.1) and some of its properties (Section 14.2), then describe a set of legal states of the service and properties of execution starting in those legal states (Section 14.3), and finally argue that our service is self-stabilizing (Section 14.4).

14.1 Client end-to-end routing specification

End-to-end routing is an important application for ad-hoc networks. End-to-end routing ($E2E$) is a service that allows arbitrary clients to communicate: a client p sends a message m to client q using the $\mathsf{esend}(m, q)_p$ action. The message may then be received by q through the $\mathsf{ercv}(m)_q$ action.

Our implementation of the end-to-end routing service, $E2E$, uses the home location service to discover a recent region location of a destination client node and then uses this location in conjunction with Geocast to deliver messages. As in the implementation of the Home Location Service, there are two parts to the end-to-end routing implementation: the client-side portion and the VSA-side portion.

The client-side portion C_p^{E2E} takes a request to send a message m to a client q and transmits it to its local VSA for forwarding. It also listens for $Vbcast$ messages originating at other clients and addressed to it, and delivers them.

The VSA V_u^{E2E} portion is very simple. A client may send it a message to be forwarded to a client. It looks up a somewhat recent location of the destination client using HLS and then sends the message via geocast to the reported region.

The TIOA specification for the individual clients is in Figure 14-1. The specification for the individual regions is in Figure 14-2. The complete service, $E2E$ is the composition of $\prod_{u\in U} Fail(V_u^{E2E} \| V_u^{Geo} \| V_u^{HL} \| VBDelay_u)$, $\prod_{p\in P} Fail(C_p^{E2E} \| C_p^{HL} \| VBDelay_p)$, and $RW \| VW \| Vbcast$. In other words, the service consists of a *Fail*-transformed automaton at each region of the composition of the end-to-end, home location, geocast, and $VBDelay$ machines; a *Fail*-transformed automaton at each client of the composition of the end-to-end, home location, and $VBDelay$ machines; and $RW \| VW \| Vbcast$.

Recall that in the Geocast (Chapter 12) and Location Management (Chapter 13) chapters, we noted that for each $u \in U$, the various geocast and home location automata at the regions were not technically VSAs since their external interfaces included more than just the allowed vcast, vrcv, and time actions. Here we can finally resolve this issue. For each $u \in U$, the VSA at region u is the composition $V_u^{E2E} \| V_u^{Geo} \| V_u^{HL}$, with all geocast, georcv, HLQuery and HLreply actions hidden. The resulting machine satisfies the conditions for being a VSA.

We now describe the pieces of the $E2E$ service in more detail.

14.1.1 Client algorithm

The signature, state, and transitions of C_p^{E2E} are in Figure 14-1.

Signature:
2 **Input** GPSupdate$(l, t)_p, l \in R, t \in \mathbb{R}^{\geq 0}$
Input esend$(m, q)_p, m \in Msg, q \in P$
4 **Input** vrcv$(\langle \text{rdata}, m, p \rangle)_p$, $m \in Msg$
Output vcast$(\langle \text{sdata}, m, q \rangle)_p$, $m \in Msg$, $q \in P$
6 **Output** ercv$(m)_p$, $m \in Msg$

8 **State:**
analog $clock \in \mathbb{R}^{\geq 0} \cup \{\perp\}$, initially $\perp$
10 $reg \in U \cup \{\perp\}$, initially $\perp$
$sdataq \in (Msg \times P)^*$, initially λ
12 $deliverq \in Msg^*$, initially λ

14 **Trajectories**:
evolve
16 $\mathbf{d}(clock) = 1$
stop when
18 Any precondition is satisfied.

20 **Transitions:**
Input GPSupdate$(l, t)_p$
22 **Effect**:
if $clock \neq t \vee reg = \perp$ **then**
24 $sdataq, deliverq \leftarrow \lambda$
$clock \leftarrow t$
26 $reg \leftarrow region(l)$

Input esend$(m, q)_p$ 28
Effect:
$sdataq \leftarrow$ **append**$(sdataq, \langle m, q \rangle)$ 30

Output vcast$(\langle \text{sdata}, m, q, reg \rangle)_p$ 32
Precondition:
$\langle m, q \rangle =$ **head**$(sdataq) \wedge clock \neq \perp \wedge reg \neq \perp$ 34
Effect:
$sdataq \leftarrow$ **tail**$(sdataq)$ 36

Input vrcv$(\langle \text{rdata}, m, p \rangle)_p$ 38
Effect:
$deliverq \leftarrow$ **append**$(deliverq, m)$ 40

Output ercv$(m)_p$ 42
Precondition:
$m =$ **head**$(deliverq) \wedge clock \neq \perp \wedge reg \neq \perp$ 44
Effect:
$deliverq \leftarrow$ **tail**$(deliverq)$ 46

Figure 14-1: Client C_p^{E2E} automaton.

The two main variables, $sdataq$ and $deliverq$, are queues. Variable $sdataq$ stores pairs $\langle m, q \rangle$ of esend requests that have not yet been forwarded to a VSA, where m is a message and q the intended recipient. Variable $deliverq$ stores messages intended for receipt by the client, but not yet ercv'ed.

The GPSupdate$(l, t)_p$ action (line 21) results in an update of the client's reg variable to the region $region(l)$ and a reset of the local clock to time t (lines 25-26). If the $clock$ variable was not t when the action occurred or if reg was $\perp$, then the $sdataq$ and $deliverq$ queues are also cleared (lines 23-24); this corresponds to a resetting of the queues either because the client has just started or because the client had incorrect local state.

A message m is sent to another client q via an esend$(m, q)_p$ input (line 28), which adds the pair $\langle m, q \rangle$ to $sdataq$ (line 30). This results in the forwarding of the information to p's current region's VSA through vcast$(\langle \mathtt{sdata}, m, q, reg \rangle)_p$ and the removal of the pair from $sdataq$ (lines 32-36).

Information about a message m for client p from other clients can be forwarded and ultimately received through a vrcv$(\langle \mathtt{rdata}, m, p \rangle)_p$ input (line 38). This adds the message m to $deliverq$ (line 40). The message m is subsequently delivered through the output

```
1  Signature:
     Input time(t)_u, t ∈ ℝ^≥0
3    Input vrcv(⟨sdata, m, q, u⟩)_u, m ∈ Msg, q ∈ P
     Input HLreply(p, v)_u, p ∈ P, v ∈ U
5    Input georcv(⟨fdata, m, p⟩)_u, m ∈ Msg, p ∈ P
     Output HLQuery(p)_u, p ∈ P
7    Output vcast(⟨rdata, m, p⟩)_u, m ∈ Msg, p ∈ P
     Output geocast(⟨fdata, m, p⟩, v)_u,
9              m ∈ Msg, p ∈ P, v ∈ U

11 State:
     analog clock ∈ ℝ^≥0 ∪ {⊥}, initially ⊥
13   bcastq ∈ 2^(Msg×P), initially ∅
     tosend ∈ P → 2^(Msg×(ℝ^≥0 ∪⊥)), initially ∅
15   findreg ∈ P → U ∪ {⊥}, initially ⊥

17 Trajectories:
     evolve
19     d(clock) = 1
     stop when
21     Any output precondition is satisfied
       ∨ ∃p ∈ P: [findreg(p) ≠ ⊥∧ tosend(p) = ∅]
23     ∨ ∃p ∈ P, m ∈ Msg, t ∈ ℝ^≥0: (⟨m, t⟩ ∈ tosend(p)
         ∧ [t> clock ∨ t ≤ q clock -2(e+d)dist(u, h(p)) -ε])
25
   Transitions:
27   Input time(t)_u
     Effect:
29     if clock ≠ t then
         clock ← t
31       bcastq ← ∅
         for each p ∈ P
33           tosend(p) ← ∅
             findreg(p) ← ⊥
35
     Input vrcv(⟨sdata, m, p, u⟩)_u
37   Effect:
       tosend(p) ← tosend(p) ∪ {⟨m, ⊥⟩}

   Output HLQuery(p)_u                                               40
   Local: m ∈ Msg
   Precondition:                                                     42
     clock ≠ ⊥∧ ⟨m, ⊥⟩∈ tosend(p)
   Effect:                                                           44
     tosend(p) ← tosend(p) − {⟨m, ⊥⟩} ∪ {⟨m, clock⟩}
                                                                     46
   Input HLreply(p, v)_u
   Effect:                                                           48
     findreg(p) ← v
                                                                     50
   Output geocast(⟨fdata, m, p⟩, v)_u
   Precondition:                                                     52
     clock ≠ ⊥∧findreg(p) = v ≠ ⊥
     ∃t:(⟨m, t⟩∈ tosend(p)∧ [t = ⊥∨ t≤ clock-2(e+d) dist(u, h(p))])  54
   Effect:
     tosend(p) ← tosend(p) − {⟨m′, t⟩ |m′ = m}                       56

   Internal cleanFind(p)_u                                           58
   Precondition:
     findreg(p) ≠ ⊥∧ tosend(p) = ∅                                    60
   Effect:
     findreg(p) ← ⊥                                                  62

   Internal cleanSend(p)_u                                           64
   Precondition:
     ∃⟨m, t⟩∈ tosend(p): [t > clock ∨ t < clock-2(e+d) dist(u, h(p))]  66
   Effect:
     tosend(p) ← tosend(p)                                           68
         − {⟨m, t⟩ |t > clock ∨ t < clock -2(e+d) dist(u, h(p))}
                                                                     70
   Input georcv(⟨fdata, m, p⟩)_u
   Effect:                                                           72
     bcastq ← bcastq ∪ {⟨m, p⟩}
                                                                     74
   Output vcast(⟨rdata, m, p⟩)_u
   Precondition:                                                     76
     clock ≠ ⊥∧ ⟨m, p⟩ ∈ bcastq
   Effect:                                                           78
     bcastq ← bcastq − {⟨m, p⟩}
```

Figure 14-2: VSA $V^{E2E}[ttl_{hb}, h]_u$ automaton.

ercv$(m)_p$ action (lines 42-46).

14.1.2 VSA algorithm

The signature, state, and transitions of V_u^{E2E} are in Figure 14-2.

There are three main variables in the $V^{E2E}[ttl_{hb}, h]_u$ automaton. The variable $bcastq$ is a set of pairs of messages and process ids; each pair corresponds to a mesasge that the VSA is about to broadcast locally for receipt by some client. The variable $tosend$ maps each process id p to a set of messages that local clients have asked the VSA to forward to p, tagged either with a timestamp indicating when it arrived at the VSA or $\perp$, indicating the message has just arrived but the location of p has not yet been queried. The variable

$findreg$ maps each process id either to a region corresponding to a recent location of the process, or $\perp$.

The VSA at a region u is told by a local client of their esends of message m to a client p via the receipt of a $\langle \mathsf{sdata}, m, p, u\rangle$ action (line 36). This adds the pair $\langle m, \perp\rangle$ to $tosend(p)$ (line 38), indicating that m is to be sent to p and that the VSA needs to look up p's region. This results in an $\mathsf{HLQuery}(p)_u$ to look up the region, resulting in the update of the pair $\langle m, \perp\rangle$ to $\langle m, clock\rangle$ (lines 40-45). Whenever a response in the form $\mathsf{HLreply}(p, v)_u$ occurs (line 47), the variable $findreg(p)$ is updated to v (line 49), indicating p was in region v recently.

For each pair $\langle m, t\rangle$ in $tosend(p)$, if $findreg(p)$ is not $\perp$, meaning that the VSA has a relatively recent location for p, the VSA forwards the message information to p's location and removes the message record from $tosend$. This is done through a $\mathsf{geocast}(\langle \mathsf{fdata}, m, p\rangle)_u$ output (lines 51-56). If there are no tuples in $tosend(p)$, meaning there are no messages that need to be forwarded to p outstanding, then $findreg(p)$ is cleared (lines 58-62).

When a $\langle \mathsf{fdata}, m, p\rangle$ message is received from the geocast service, indicating that there is a message m intended for some client p that should be nearby, the VSA adds the pair $\langle m, p\rangle$ to its $bcastq$ (lines 71-73). This results in the local broadcast via $\mathsf{vcast}(\langle \mathsf{rdata}, m, p\rangle)_u$ (lines 75-79) to inform the client p of the message m.

If a tuple $\langle m, t\rangle$ is in $tosend(p)$ but the timestamp t is either from the future (the result of corruption) or from longer than $2(e + d)dist(u, h(p))$ ago (meaning that the HLQuery for p's location timed out), then $\langle m, t\rangle$ is considered to be expired and is removed from $tosend(p)$ (lines 64-69).

14.2 Properties of executions of the end-to-end routing service

The end-to-end communication service allows clients to send messages to other clients. A client p can send a message m to another client q through the $\mathsf{esend}(m, q)_p$ action. If client

q can be found at an alive VSA and q does not move too far for a sufficient amount of time, the message will then be received by client q through the $\mathsf{ercv}(m)_q$ action.

More formally, we say that a process p is *hosted by region* u at a time t if:

1. For each $t' \in [t, t + 3(e + d)D + e + d]$, u is not failed.

2. For each $t' \in [t - ttl_{hb} - d - (e+d)D, t + (e+d)D + d]$, $reg^-(p, t') = reg^+(p, t') = u$.

3. For each $t' \in [t + (e+d)D + d, t + 3(e+d)D + e + 2d]$, $\{reg^-(p, t') = reg^+(p, t')\} \subseteq nbrs^+(u)$ and p is not failed.

This amounts to saying that a proces sis hosted by a region u at time t if: (1) region u is not failed from time t until d before what will be the deadline for message delivery in the end-to-end communication service; (2) region u has been the region of p long enough that any location information stored at p's home location from t until any home location query started at time t can complete will indicate that p is either in u or some newer region; and (3) process p stays in u or a neighboring region of u until any end-to-end communication started at t can complete.

We say that a $\mathsf{esend}(m, q)_p$ at a time t is *receivable* if there exists some region u such that:

1. Process p is not failed at time t.

2. Process q is hosted by region u at time t.

3. For each $t' \in [t, t + d]$ and each $v \in \{reg^-(p, t), reg^+(p, t)\}$, an $\mathsf{HLquery}(q)_v$ at time t' is serviceable.

4. For each $v \in \{reg^-(p, t), reg^+(p, t)\}$, there exists at least one shortest path from v to u of regions that are nonfailed and have $clock$ values equal to the real-time for the interval $[t, t + (e + d)(2dist(v, h(p)) + dist(v, u))]$.

Then we can show the following result:

Lemma 14.1 *The $E2E$ service guarantees that in each execution α of $E2E$, there exists a function mapping each $\mathsf{ercv}(m)_q$ event to a $\mathsf{esend}(m, q)_p$ event such that the following hold:*

1. Integrity*: If an* ercv$(m)_q$ *event* π *is mapped to an* esend$(m, q)_p$ *event* π'*, then* π' *occurs before* π*.*

2. Bounded-time delivery*: If an* ercv$(m)_q$ *event* π *is mapped to an* esend$(m, q)_p$ *event* π' *where* π' *occurs at time* t*, then event* π *occurs in the interval* $(t, t + 3(e + d)D + e + 2d]$*.*

3. Reliable receivable delivery*: This guarantees that a message that is end-to-end sent will be received if it is receivable: If an* esend$(m, q)_p$ *event* π' *occurs at time* t*,* $\alpha.ltime > t + 3(e + d)D + e + 2d$*, and* π' *is receivable, then there exists a* ercv$(m)_q$ *event* π *such that* π *occurs in the interval* $(t, t + 3(e + d)D + e + 2d]$*.*

Proof sketch: It is easy to define the mapping from ercv to esend events described above by reasoning about the chain of actions connecting a ercv and esend event: For each ercv$(m)_q$ event, m must have been removed from $deliverq$ (line 44). Such an m is added to $deliverq$ through the receipt of a rdata message containing m (lines 38-40), which in turn was sent by a VSA based on one of its local $bcastq$ tuples (lines 75-79). Such a tuple in $bcastq$ came from the receipt of an fdata message (lines 71-73), which was geocast by some VSA based on its local $tosend$ and $findreg$ variables (lines 51-56). Such values in $tosend$ queues are added based on receipt of an sdata message (lines 36-38) which are only sent by a client in response to an esend. Hence, for each ercv$(m)_q$ event there must have been an esend$(m, q)_p$ event that occurred before. The mapping selects the latest such one.

The two interesting properties to check are Bounded-time delivery and Reliable receivable delivery. Bounded-time delivery is guaranteed by the fact that in the reasoning above, there is an upper bound on the amount of time each step can take. The receipt of the rdata message sent by a VSA can take up to $e + d$ time. The receipt of the fdata message at the VSA that caused the rdata message can take up to $(e + d)D$ time, the maximum time for a geocast to complete. The VSA that geocast that fdata message only did so if its $findreg$ indicated a location for the end-to-end message recipient; this can take up to $2D(e+d)$ time for the VSA to discover (the time is the maximum time for an HLQuery for the location

to complete). This is all after the VSA that geocast that fdata message received an sdata message sent from a client up to d time before. The sum of these times is $3D(e+d)+e+2d$.

For Reliable receivable delivery, we note that the properties of the underlying HLS and $Geocast$ services make the property easy to check. Consider a receivable $\mathsf{esend}(m, q)_p$ event π' occurs at time t. We need to show that an $\mathsf{ercv}(m)_q$ event π occurs within $3D(e + d) + e + 2d$ time. By property 1 of *receivable*, we know that p doesn't fail at time t. This means that it will transmit an sdata message to its VSA at time t. By property 3 of *receivable*, a local VSA will receive this sdata message by time $t + d$ and either already have a listed location u for q or will HLQuery for one. If it must perform an HLQuery, we know it will receive a reply by time $t+d+2D(e+d)$, or $2D(e+d)$ later. This then prompts the VSA to geocast an fdata message to u. Since property 4 of *receivable* holds, we know that the geocast will arrive at region u at most $(e + d)D$ later, by time $t + d + 3D(e + d)$. By property 1 of our definition of hosting, we know that region u will be alive to receive the message. It then takes region u up to e time to vcast a rdata message to q, and a further d time for the message to arrive at q. By property 3 of hosting, we know that q is alive and will vrcv the rdata message, causing it to immediately ercv the message embedded in the rdata message. This happens by at time at most $t + 3D(e + d) + e + 2d$. ■

14.3 Legal sets

Here we describe a legal set of $E2E$ by describing a sequence of four legal sets, each a subset of the prior. Recall from Lemma 3.13 that a legal set of states for a TIOA is one where each closed execution fragment starting in a state in the set ends in a state in the set. We break the definition of the legal set up into multiple legal sets in order to simplify the proof reasoning and more easily prove stabilization later, in Section 14.4. Because the proofs in this section are routine, we omit them. At the end of this section, we discuss properties of execution fragments of $E2E$ that start in our set of legal states.

14.3.1 Legal set L^1_{e2e}

The first set of legal states describes some properties that are locally checkable at a region or client and that become true at an alive VSA at the time of the first time input for the VSA and GPSupdate input at a client, assuming the underlying HLS system is in a legal state.

Definition 14.2 *Let L^1_{e2e} be the set of states x of $E2E$ where all of the following hold:*

1. $x\lceil X_{HLS} \in L^5_{hls}$.
   This says that the state restricted to the variables of HLS is a legal state of HLS.
2. *For each $p \in P : \neg failed_p$ (nonfailed client):*
   (a) $clock_p \neq \bot \Rightarrow [clock_p = now \wedge reg_p = reg(p)]$.
   This says that if the local clock is not $\bot$, then it is set to the current real-time and reg_p is p's current region.
   (b) *For each $u \in U$, $[\exists\langle \mathsf{sdata}, m, q, u\rangle \in to_send^-_p to_send^+_p] \Rightarrow u \in \{reg^-(p, now), reg^+(p, now)\}$.*
   This says that if an sdata message is in one of a client's $VBDelay$ queues, then the message correctly indicates a region that the client has been in at this time.
   (c) *For each $m \in deliverq_p$, $\exists\langle\langle \mathsf{rdata}, m, p\rangle, u, t, P'\rangle \in vbcastq$:*
   $t \geq now - d \wedge p \notin P'$.
   This says that each message sitting in $deliverq$ was sent in an rdata message to p within the last d time.
3. *For each $u \in U : \neg failed_u \wedge clock_u \neq \bot$ (nonfailed VSA that received a time input):*
   (a) $clock_u = now$.
   This says that the local clock should be equal to the real-time.
   (b) *For each $p \in P$ and $\langle m, t\rangle \in tosend_u(p) : t \leq clock_u$.*
   This just says that any records of messages that are waiting to be geocast to another region do not have timestamps from the future.

(c) *For each* $p \in P, v \in U$, $findreg_u(p) = v \Rightarrow \exists t \in [now - ttl_{hb} - d - (e + d)(dist(v, h(p)) + dist(h(p), u)), now] : v \in \{reg^+(p, t), reg^-(p, t)\}$.

This says that if the VSA's $findreg$ *indicates that a process* p *was recently located at region* v*, then process* p *was in that region within the last* $ttl_{hb} + d + (e + d)(dist(v, h(p)) + dist(h(p), u))$ *time.*

(d) *For each* $\langle m, p\rangle \in bcastq_u$,

$\exists\langle\langle \mathsf{geocast}, \langle \mathsf{fdata}, m, p\rangle, w, u, t\rangle, w, t', P'\rangle \in vbcastq : t \geq now-(e+d)D$.

This says that any pair in a VSA's $bcastq$ *was part of an* fdata *message that was* geocast *to* u *within the last* $(e + d)D$ *time.*

Lemma 14.3 L^1_{e2e} *is a legal set for* $E2E$.

14.3.2 Legal set L^2_{e2e}

The second set of legal states describes some properties that hold after any spurious VSA messages are broadcast and spurious $Vbcast$ messages are delivered.

Definition 14.4 *Let* L^2_{e2e} *be the set of states* x *of* $E2E$ *where all of the following hold:*

1. $x \in L^1_{e2e}$.

   This says that L^2_{e2e} *is a subset of* L^1_{e2e}.

2. *For each* $\langle\langle \mathsf{sdata}, m, q, reg\rangle, u, t, P'\rangle \in vbcastq$,

   $t \geq now - d \Rightarrow reg \in \{reg^-(p, t), reg^+(p, t)\}$.

   This says that for any sdata *transmission made within the last* d *time, the* sdata *message was sent by a process to a local VSA.*

3. *For each* $u \in U : \neg failed_u$ *(nonfailed VSA):*

   (a) $\nexists\langle\langle \mathsf{sdata}, m, q, v\rangle, t\rangle \in to_send_u$.

   This says that a VSA cannot be in the process of transmitting an sdata *message.*

   (b) *For each* $\langle\langle \mathsf{rdata}, m, p\rangle, t\rangle \in to_send_u$: $\exists\langle\langle \mathsf{geocast}, \langle \mathsf{fdata}, m, p\rangle, w, u, t'\rangle, v, t'', P'\rangle \in vbcastq : t' + (e + d)D + e \geq$

$t + now - rtimer_u$.

This says that any rdata *message in $VBDelay_u$ can be matched to an* fdata *transmission to region u made within the last $(e+d)D+e$ time.*

4. *For each $\langle\langle \mathsf{rdata}, m, p\rangle, u, t, P'\rangle \in vbcastq$, $t \geq now - d \Rightarrow \exists\langle\langle \mathsf{geocast}, \langle \mathsf{fdata}, m, p\rangle, w, u, t'\rangle, v, t'', P'\rangle \in vbcastq : t' + (e+d)D + e \geq t$.*
   This says that any rdata *transmission in $Vbcast$ from the last d time can be matched to an* fdata *transmission to region u made up to $(e+d)D+e$ time before the* rdata *transmission.*

Lemma 14.5 *L^2_{e2e} is a legal set for $E2E$.*

14.3.3 Legal set L^3_{e2e}

The third set of legal states describes some properties that hold after any VSA records that could cause the forwarding of spurious end-to-end messages are removed.

Definition 14.6 *Let L^3_{e2e} be the set of states x of $E2E$ where all of the following hold:*

1. *$x \in L^2_{e2e}$.*
   This says that L^3_{e2e} is a subset of L^2_{e2e}.

2. *For each $u \in U : \neg failed_u$, for each $p \in P, [(\exists v \in U, m \in Msg : ledger_u(\langle\langle \mathsf{fdata}, m, p\rangle, u, v, now\rangle) \neq null) \vee \exists\langle m, t\rangle \in tosend_u(p) : t \geq now - 2D(e+d)] \Rightarrow \exists\langle\langle \mathsf{sdata}, m, p, u\rangle, v, t', P'\rangle \in vbcastq : [u \notin P' \wedge t' \geq now - d \wedge (t \neq \perp \Rightarrow t' \geq t - d)]$.*
   This says that any record in $tosend$ or any fdata *message that was just geocast can be matched to an* sdata *transmission to the region made no more than d ago and d before the record's timestamp if a non-$\perp$ timestamp exists.*

Lemma 14.7 *L^3_{e2e} is a legal set for $E2E$.*

14.3.4 Legal set L^4_{e2e}

The fourth set of legal states describes some properties that hold after any bad forwards of end-to-end messages are removed.

Definition 14.8 *Let L^4_{e2e} be the set of states x of $E2E$ where all of the following hold:*

1. $x \in L^3_{e2e}$.
   This says that L^4_{e2e} is a subset of L^3_{e2e}.

2. *For each* $\langle\langle \mathsf{geocast}, \langle \mathsf{fdata}, m, p\rangle, u, v, t\rangle, w, t', P'\rangle \in vbcastq$*:* $t \geq now - (D(e+d) + e + d) \Rightarrow [(\exists \langle\langle \mathsf{sdata}, m, p, u\rangle, v, t'', P'\rangle \in vbcastq : t'' + d + 2(e+d)dist(u, h(p)) \geq t) \wedge \exists t^* \in [t - ttl_{hb} - d - (e+d)(dist(v, h(p)) + dist(h(p), u)), t] : v \in \{reg^-(p, t^*), reg^+(p, t^*)\}]$.
   This says that any fdata *transmission from within the last $(e+d)D+e+d$ time can be matched to an* sdata *transmission that occurred no more than $2(e+d)dist(u, h(p)) + d$ time before the timestamp of the* fdata *geocast. In addition, the* fdata *message is being* geocast *to a region v that contained the intended end-to-end recipient at some time in the $ttl_{hb} + d + (e+d)(dist(v, h(p)) + dist(h(p), u))$ interval leading up to the time of the* fdata *transmission.*

Lemma 14.9 *L^4_{e2e} is a legal set for $E2E$.*

Properties of execution fragments starting in L^4_{e2e}

As in the location management service, we can describe the properties of execution fragments of $E2E$ that start in L^4_{e2e} as properties of executions of $E2E$, as described in Section 14.2. As before, the difference is in the mapping of some subset of ercv events that occur towards the beginning of the execution fragment.

More formally, we can say the following:

Lemma 14.10 *$E2E$ guarantees that for an execution fragment α starting in L^4_{e2e}, there exists a subset Π of the* ercv *events in α such that:*

1. *There exists a function mapping each* ercv *event in* Π *to an* esend *event such that the three properties (Integrity, Bounded-time delivery, and Reliable receivable delivery) hold.*

2. *For every* $\mathsf{ercv}(m)_q$ *event* π *not in* Π *where* π *occurs at some time* t*, it must be the case that* $t - \alpha.fstate(now) \leq 3D(e+d) + e + 2d$*.*

This concept and proof is similar to the material in Section 13.3.5, where we described the properties of execution fragments of HLS as a variant of the properties of executions of HLS, adjusting for a subset of reply events towards the beginning of a fragment.

14.4 Self-stabilization

We've seen that L^4_{e2e} is a legal set for $E2E$. Here we show that $\prod_{u \in U} Fail(VBDelay_u \| V^{Geo}_u \| V^{HL}_u \| V^{E2E}_u) \prod_{p \in P} Fail(VBDelay_p \| C^{HL}_p \| C^{E2E}_p)$ self-stabilizes to L^4_{e2e} relative to $R(RW \| VW \| Vbcast)$ (Theorem 14.16), meaning that if certain program portions of the implementation are started in an arbitrary state and run with $R(RW \| VW \| Vbcast)$, the resulting execution eventually gets into a state in L^4_{e2e}. Using Theorem 14.16, we then conclude that after an execution of $E2E$ has stabilized, the execution fragment from the point of stabilization on satisfies the properties described in Section 14.3.4.

The proof of the main stabilization result for the chapter, Theorem 14.16, breaks stabilization down into two large phases, corresponding to stabilization of the lower level HLS service (which includes the stabilization of the $GeoCast$ service), followed by stabilization of the $E2E$ service assuming that HLS has stabilized. We have seen that HLS stabilizes to the set of legal states L^5_{hls} in Section 13.4. What we need to show for Theorem 14.16 is that, starting from a set of states where HLS is already stabilized, $E2E$ stabilizes to L^4_{e2e} (Lemma 14.15). We do this in four stages, one for each of the legal sets described in Section 14.3. The first stage starts from a state where HLS is already stabilized and ends up in the first legal set. The second stage starts in the first legal set and ends up in the second, etc.

The first lemma describes the first stage of $E2E$ stabilization, to legal set L^1_{e2e}. It says that within t^1_{e2e} time of HLS stabilizing, where $t^1_{e2e} > \epsilon_{sample}$, the system ends up in a state in L^1_{e2e}.

Lemma 14.11 *Let t^1_{e2e} be any t such that $t > \epsilon_{sample}$. $frags_{E2E}^{\{x|x\lceil X_{HLS} \in L^5_{hls}\}}$ stabilizes in time t^1_{e2e} to $frags_{E2E}^{L^1_{e2e}}$.*

Proof sketch: To see this result, just consider the first time after each node has received a time or GPSupdate input, which takes at most ϵ_{sample} time to happen. ■

The next lemma describes the second stage of $E2E$ stabilization. It shows that starting from a state in L^1_{e2e}, $E2E$ ends up in a state in L^2_{e2e} within t^2_{e2e} time, where t^2_{e2e} is any time greater than $e + d$.

Lemma 14.12 *Let t^2_{e2e} be any t such that $t > e + d$. $frags_{E2E}^{L^1_{e2e}}$ stabilizes in time t^2_{e2e} to $frags_{E2E}^{L^2_{e2e}}$.*

Proof: By Lemma 3.21, we just need to show that for any length-t^2_{e2e} prefix α of an element of $frags_{E2E}^{L^1_{e2e}}$, $\alpha.lstate$ is in L^2_{e2e}. We examine each property of L^2_{e2e}.

By Lemma 14.11, since the first state of α is in L^1_{e2e}, we know that property 1 of L^2_{e2e} holds in each state of α.

For property 2, we note that each new such sdata message added to one of a client's to_send queues and then propagated to $Vbcast$, the property will hold and continue to hold thereafter. Hence, the only thing we need to worry about messages already in a to_send queue or in $vbcastq$ in $\alpha.fstate$. However, after d time elapses from the start of α, the property will be trivially true.

For property 3, we consider each part. Property 3(a) will hold after at most e time, the time it takes for any such errant messages in $\alpha.fstate$ to be propagated out to $Vbcast$. For property 3(b), we note that a new rdata message is only added to to_send_u if there previously was a corresponding pair $\langle m, p\rangle$ in the VSA's $bcastq$, which by property 3(d) of L^1_{e2e} implies that any newly added rdata message satisfies this property 3(b). This means that we only need to worry about rdata messages already in to_send_u at the start of α.

Once in to_send_u, it is at most e time before a message is removed from to_send_u. Hence, after e time has passed, the property will be trivially true.

For property 4, since each new rdata message added to $vbcastq$ first is in to_send_u, we know that any such messages added after property 3(b) holds will satisfy property 4. After d time elapses from when property 3(b) holds, the property will be trivially true. ■

For the third stage of $E2E$ stabilization, the next lemma shows that starting from a state in L^2_{e2e}, $E2E$ ends up in a state in L^3_{e2e} within t^3_{e2e} time, where t^3_{e2e} is any time greater than $2D(e + d)$.

Lemma 14.13 *Let t^3_{e2e} be any t such that $t > 2(e + d)D$. (Recall D is the hop count diameter of the network.) $frags^{L^2_{e2e}}_{E2E}$ stabilizes in time t^3_{e2e} to $frags^{L^3_{e2e}}_{E2E}$.*

Proof: By Lemma 3.21, we just need to show that for any length-t^3_{e2e} prefix α of an element of $frags^{L^2_{e2e}}_{E2E}$, $\alpha.lstate$ is in L^3_{e2e}. We examine each property of L^3_{e2e}.

By Lemma 13.14, since the first state of α is in L^2_{e2e}, we know that property 1 of L^3_{e2e} holds in each state of α.

For property 2, notice that for each new entry added to $tosend$ the property will hold, since the new entry will be the result of the receipt of an sdata message that satisfies the properties from $Vbcast$. Hence, the only $tosend$ entries we need to worry about are the $tosend$ entries already there in $\alpha.fstate$. However, after $2D(e + d)$ time elapses from the start of α, the property will be trivially true. For the $ledger$ entries, we note that each new entry in the $ledger$ after the bogus $tosend$ entries are cleared satisfy the property. ■

The next lemma, for the fourth stage of $E2E$ stabilization, shows that starting from a state in L^3_{e2e}, $E2E$ ends up in a state in L^4_{e2e} within t^4_{e2e} time, where t^4_{e2e} is any time greater than $d + e + (e + d)D$.

Lemma 14.14 *Let t^4_{e2e} be any t such that $t > d + e + (e + d)D$. $frags^{L^3_{e2e}}_{E2E}$ stabilizes in time t^4_{e2e} to $frags^{L^4_{e2e}}_{E2E}$.*

Proof: By Lemma 3.21, we just need to show that for any length-t^4_{e2e} prefix α of an element of $frags^{L^3_{e2e}}_{E2E}$, $\alpha.lstate$ is in L^4_{e2e}. We examine each property of L^4_{e2e}.

By Lemma 14.13, since the first state of α is in L^3_{e2e}, we know that property 1 of L^4_{e2e} holds in each state of α.

For property 2, notice that for each new tuple added to $vbcastq$ for a geocast of a fdata message, the property will be true since the message will come from the VSA's $ledger$, which we know by property 2 of L^3_{e2e} will satisfy the property we need here. Hence, the only fdata geocast messages in $vbcastq$ that we need to worry about are those that are present in the first state of α. However, after $d + e + (e + d)D$ time, the property will trivially be true. ■

We now have all of the pieces of reasoning for the four stages of the second phase of $E2E$ stabilization. (Recall that the second phase of $E2E$ stabilization occurs after HLS has stabilized, corresponding to HLS state being in the set L^5_{hls}.) We then combine this reasoning from Lemmas 14.11-14.14 to show that the second phase of stabilization of $E2E$ takes t'_{e2e} time, $t'_{e2e} > \epsilon_{sample} + (3D + 2)(e + d)$, to stabilize:

Lemma 14.15 *Let t'_{e2e} be any t such that $t > \epsilon_{sample} + (3D + 2)(e + d)$. Then $frags^{\{x|x\lceil X_{HLS} \in L^5_{hls}\}}_{E2E}$ stabilizes in time t'_{e2e} to $frags^{L^4_{e2e}}_{E2E}$.*

Proof: The result follows from the application of Lemma 3.7 on the four lemmas (Lemmas 14.11-14.14) above.

Let t' be $(t'_{e2e} - (\epsilon_{sample} + (3D + 2)(e + d)))/4$. Then let t^1_{e2e} be $t' + \epsilon_{sample}$, t^2_{e2e} be $t' + e + d$, t^3_{e2e} be $t' + 2(e + d)D$, and t^4_{e2e} be $t' + d + e + (e + d)D$. (These terms are chosen so as to satisfy the constraints that $t^1_{e2e} > \epsilon_{sample}$, $t^2_{e2e} > e + d$, etc.)

Let B_0 be $frags^{\{x|x\lceil X_{HLS} \in L^5_{hls}\}}_{E2E}$, B_1 be $frags^{L^1_{e2e}}_{E2E}$, B_2 be $frags^{L^2_{e2e}}_{E2E}$, B_3 be $frags^{L^3_{e2e}}_{E2E}$, and B_4 be $frags^{L^4_{e2e}}_{E2E}$ in Lemma 3.7. Let t_1 be t^1_{e2e}, t_2 be t^2_{e2e}, t_3 be t^3_{e2e}, and t_4 be t^4_{e2e} in Lemma 3.7. Then by Lemma 3.7 and Lemmas 14.11-14.14, we have that $frags^{\{x|x\lceil X_{HLS} \in L^5_{hls}\}}_{E2E}$ stabilizes in time $t^1_{e2e} + t^2_{e2e} + t^3_{e2e} + t^4_{e2e}$ to $frags^{L^4_{e2e}}_{E2E}$.

Since $t'_{e2e} = t^1_{e2e} + t^2_{e2e} + t^3_{e2e} + t^4_{e2e}$, we conclude that $frags^{\{x|x\lceil X_{HLS} \in L^5_{hls}\}}_{E2E}$ stabilizes in time t'_{e2e} to $frags^{L^4_{e2e}}_{E2E}$. ■

Using this and our prior result on HLS stabilization (Theorem 13.19) we can now finally show the main stabilization result of this chapter. The proof of the result breaks

down the self-stabilization of $E2E$ into two phases, the first being where HLS stabilizes, and the second being where the remaining pieces of $E2E$ stabilize.

Theorem 14.16 $\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}\|V_u^{HL}\|V_u^{E2E})\prod_{p\in P} Fail(VBDelay_p\|C_p^{HL}\|C_p^{E2E})$ *self-stabilizes in* t_{e2e} *time,* $t_{e2e} > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$*, to* L^4_{e2e} *relative to* $R(RW\|VW\|Vbcast)$*.*

Proof: For brevity, we will use $execs_{U-E2E}$ to refer to $execs_{U(\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}\|V_u^{HL}\|V_u^{E2E})\prod_{p\in P} Fail(VBDelay_p\|C_p^{HL}\|C_p^{E2E}))\|R(RW\|VW\|Vbcast)}$. We must show that $execs_{U-E2E}$ stabilizes in time t_{e2e} to $frags^{L^4_{e2e}}_{\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}\|V_u^{HL}\|V_u^{E2E})\prod_{p\in P} Fail(VBDelay_p\|C_p^{HL}\|C_p^{E2E})\|R(RW\|VW\|Vbcast)}$. By Corollary 3.11, $frags^{L^4_{e2e}}_{\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}\|V_u^{HL}\|V_u^{E2E})\prod_{p\in P} Fail(VBDelay_p\|C_p^{HL}\|C_p^{E2E})\|R(RW\|VW\|Vbcast)}$ is the same as $frags^{L^4_{e2e}}_{E2E}$. This means that we must show that $execs_{U-E2E}$ stabilizes in time t_{e2e} to $frags^{L^4_{e2e}}_{E2E}$. The result follows from the application of transitivity of stabilization (Lemma 3.6) on the two phases of $E2E$ stabilization.

For the first phase, we note that by Theorem 13.19, $execs_{U-E2E}$ stabilizes in time t_{hls} to $frags^{\{x|x\lceil X_{HLS}\in L^5_{hls}\}}_{E2E}$.

For the second phase, let t'_{e2e} be $t_{e2e} - t_{hls}$. Since $t_{e2e} > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$, this implies that $t'_{e2e} > \epsilon_{sample} + 2e + 2d + 3(e+d)D$. By Lemma 14.15, we have that $frags^{\{x|x\lceil X_{HLS}\in L^5_{e2e}\}}_{E2E}$ stabilizes in time t'_{e2e} to $frags^{L^4_{e2e}}_{E2E}$.

Taking B to be $execs_{U-E2E}$, C to be $frags^{\{x|x\lceil X_{HLS}\in L^5_{hls}\}}_{E2E}$, and D to be $frags^{L^4_{e2e}}_{E2E}$ in Lemma 3.6, we have that $execs_{U-E2E}$ stabilizes in time $t_{hls} + t'_{e2e}$ to $frags^{L^4_{e2e}}_{E2E}$.

Since $t_{e2e} = t_{hls} + t'_{e2e}$, we conclude that $\prod_{u\in U} Fail(VBDelay_u\|V_u^{Geo}\|V_u^{HL}\|V_u^{E2E})\prod_{p\in P} Fail(VBDelay_p\|C_p^{HL}\|C_p^{E2E})$ self-stabilizes in t_{e2e} time, $t_{e2e} > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$, to L^4_{e2e} relative to $R(RW\|VW\|Vbcast)$. ■

This immediately implies the following result about the associated VSA layer algorithm:

Lemma 14.17 *Let* alg_{e2e} *be a* $VAlg$ *such that for each* $p \in P$*,* $alg_{e2e}(p) = C_p^{HL}\|C_p^{E2E}$ *and for each* $u \in U$*,* $alg_{e2e}(u) =$

$\mathsf{ActHide}(\{\mathsf{geocast}(m,v)_u, \mathsf{georcv}(m)_v, \mathsf{HLQuery}(p)_u, \mathsf{HLreply}(p,v)_u | m \in Msg, u, v \in U, p \in P\}, V_u^{Geo} \| V_u^{HL} \| V_u^{E2E})$.
Let t_{e2e} *be any* t *such that* $t > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$.
Then $VLNodes[alg_{e2e}]$ *self-stabilizes in time* t_{e2e} *to* L^4_{e2e} *relative to* $R(RW \| VW \| Vbcast)$.

With Lemma 14.10, this allows us to conclude that after an execution of $E2E$ has stabilized, the execution fragment from that point on satisfies the properties in Section 14.3.4:

Lemma 14.18 *Let* t_{e2e} *be any* t *such that* $t > t_{hls} + \epsilon_{sample} + 2e + 2d + 3(e+d)D$.
Then $execs_{U(VLNodes[alg_{e2e}]) \| R(RW \| VW \| Vbcast)}$ *stabilizes in time* t_{e2e} *to a set* $\mathcal{A}$ *of execution fragments such that for each* $\alpha \in \mathcal{A}$*, there exists a subset* Π *of the* ercv *events in* α *such that:*

1. *There exists a function mapping each* ercv *event in* Π *to an* esend *event such that the three properties (Integrity, Bounded-time delivery, and Reliable receivable delivery) hold.*

2. *For every* $\mathsf{ercv}(m)_q$ *event* π *not in* Π *where* π *occurs at some time* t*, it must be the case that* $t - \alpha.fstate(now) \leq 3D(e+d) + e + 2d$.

In other words, if we start each client and VSA running the end-to-end routing program in an arbitrary state and run them with $RW \| VW \| Vbcast$ started in a reachable state, then the execution eventually reaches a point from which the properties of the end-to-end routing service described in Section 14.3.4 are satisfied. These properties basically say that Integrity, Bounded-time delivery, and Reliable receivable delivery hold for most of the ercv and esend events in the fragment, modulo several straggler ercv events that occur early in the execution fragment.

14.5 Extensions

Here we briefly describe some possible extensions to our $E2E$ algorithm:

Routing optimizations: Once the location of a client is known, communication with the client can be continued directly, and movements during the conversation may be piggybacked on the information transferred in order to update the destination according to the move (as suggested [38]). We also note that we can use an embedded tree location scheme such as the one in [38], implemented by virtual automata, where intermediate tree nodes are also mapped to regions.

Sleeping client messaging service: Mobile clients might be able to shut down to conserve power. We could guarantee that a sleeping client eventually receives messages intended for it by having local VSAs save the messages. The VSAs then, at predefined times, broadcast the messages. Sleeping clients wake up for these broadcasts, receive their messages, and can go to sleep again afterwards.

Chapter 15

Motion Coordination

In this chapter, we describe how to use a variant of the VSA layer to help a set of mobile robots arrange themselves on any specified curve on the plane in the presence of dynamic changes both in the underlying ad hoc network and the set of participating robots. This application serves as an example of a coordination problem, where VSAs can communicate with client nodes to change the motion trajectories of those clients. The VSAs coordinate among themselves to distribute the client nodes relatively uniformly among the VSAs' regions. Each VSA directs its local client nodes to align themselves on the local portion of the target curve, and each client node then moves towards the points indicated. The resulting motion coordination protocol is self-stabilizing, in that each robot can begin the execution in any arbitrary state and at any arbitrary location in the plane. In the context of this application, self-stabilization is especially desirable since it ensures that the robots can adapt to changes in the desired target formation.

15.1 Background

In this chapter, we study the problem of coordinating the behavior of a set of autonomous mobile robots (physical nodes) in the presence of changes in the underlying communication network as well as changes in the set of participating robots. Consider, for example, a system of firefighting robots deployed throughout forests and other arid wilderness areas. Significant levels of coordination are required in order to combat the fire: to prevent the

fire from spreading, it has to be surrounded; to put out the fire, firefighters need to create "firebreaks" and spray water; they need to direct the actions of (potentially autonomous) helicopters carrying water. All this has to be achieved with the set of participating agents changing and with unreliable (possibly wireless) communication between agents. Similar scenarios arise in a variety of contexts, including search and rescue, emergency disaster response, remote surveillance, and military engagement, among many others. In fact, autonomous coordination has long been a central problem in mobile robotics.

We focus on a generic coordination problem that, we believe, captures many of the complexities associated with coordination in real-world scenarios. We assume that the mobile robots are deployed in a large two-dimensional plane, and that they can coordinate their actions by local communication using wireless radios. The robots must arrange themselves to form a particular pattern, specifically, a continuous curve drawn in the plane. The robots must spread themselves uniformly along this curve. In the firefighting example described above, this curve might form the perimeter of the fire.

The problem of motion coordination has been studied in a variety of contexts, focusing on several different goals: flocking [55]; rendezvous [5, 63, 69]; aggregation [43]; deployment and regional coverage [21]. Control theory literature contains several algorithms for achieving spatial patterns [10, 19, 41, 77]. These algorithms assume that the agents process information and communicate synchronously, and hence, they are analyzed based on differential or difference equations models of the system. Convergence of this class of algorithms over unreliable and delay-prone communication channels have been studied recently in [15].

Geometric pattern formation with vision-based models for mobile robots have been investigated in [22, 40, 42, 80, 81, 83]. In these weak models, the robots are oblivious, identical, anonymous, and often without memory of past actions. For the memoryless models, the algorithms for pattern formation are often automatically self-stabilizing. In [22, 23], for instance, a self-stabilizing algorithm for forming a circle has been presented. These weak models have been used for characterizing the class of patterns that can be formed and for studying the computational complexity of formation algorithms, under different assumptions about the level of common knowledge amongst agents, such as, knowledge of

distance, direction, and coordinates [80, 83].

These types of coordination problems can be quite challenging due to the dynamic and unpredictable environment that is inherent to wireless ad hoc networks. Robots may be continuously joining and leaving the system, and they may fail. In addition, wireless communication is notoriously unreliable due to collisions, contention, and various wireless interference.

Here we show how the VSA Layer can implement a reliable and robust protocol for coordinating mobile robots. The protocol relies on the VSAs to organize the mobile robots in a consistent fashion. Each VSA must decide based on its own local information which robots to keep in its own region, and which to assign to neighboring regions; for each robot that remains, the VSA determines where on the curve the robot should reside. Unlike in the prior three applications (Geocast, location management, and end-to-end communication), the client motion in the motion coordination protocol is controllable by the client, allowing the client to change its motion trajectory based on instructions from a VSA.

We have previously presented a protocol for coordinating mobile devices using virtual infrastructure in [66]. The paper described how to implement a simple asynchronous virtual infrastructure, and proposed a protocol for motion coordination. This earlier protocol relies on a weaker (i.e., untimed) virtual layer (see [30, 75]), while the current protocol relies on a stronger (i.e., timed) virtual layer. As a result, our new coordination protocol is somewhat simpler and more elegant than the previous version. Virtual infrastructure has also been considered in [11] for collision prevention of airplanes.

In order that the robot coordination be truly robust, our new coordination protocol is also *self-stabilizing*, meaning that each robot can begin in an arbitrary state, in an arbitrary location in the network, and yet the distribution of the robots will still converge to the specified curve. When combined with our stabilizing emulation of the VSA Layer, we end up with entirely self-stabilizing solution for the problem of autonomous robot coordination.

Recall that self-stabilization provides many advantages. Given the unreliable nature of wireless networks, it is possible that occasionally (due to aberrant interference) a significant fraction of messages may be lost, disrupting the protocol; a self-stabilizing algorithm can readily recover from this. Moreover, a self-stabilizing algorithm can cope with more

dynamic coordination problems. In real-life scenarios, the required formation of the mobile nodes may change. In the firefighting example above, as the fire advances or retreats, the formation of firefighting robots must adapt. A self-stabilizing algorithm can adapt to these changes, continually re-arranging the robots along the newly chosen curve.

Another technical contribution of this chapter is the exemplification of a proof technique for showing self-stabilization of systems implemented using virtual infrastructure. The proof technique has three parts. First, using invariant assertions and standard control theory results we show that from any initial state, the application protocol, in this case, the motion coordination algorithm converges to an *acceptable state* (Section 15.3). Next, we describe a set of *legal states* of the algorithm (Section 15.4.1). Using a simulation relation we show that the set of legal states behaves just like the set of reachable states of the complete system—the VSA layer running the coordination algorithm (Section 15.4.2). Then we show that the algorithm always stabilizes to a legal state even when it starts from some arbitrary state after failures (Section 15.4.3). From any legal state the algorithm then eventually behaves as if it has reached an acceptable state provided there are no further failures. It has already been shown in Section 11.3.4 that our implementation of the VSA layer itself is self-stabilizing and produces traces that satisfy certain properties with respect to the failure pattern of VSAs. Combining the stabilization of the implementation of the VSA layer and the application protocol, we are able to conclude self-stabilization of the emulation of the system (Theorem 15.22).

15.2 Motion Coordination using Virtual Nodes

We assume a variant of the VSA layer described in Chapter 7. The only difference between the original VSA layer and the variant used in this chapter is in the control of the motion of client nodes, described in Section 15.2.3.

To describe the motion coordination problem, we fix $\Gamma : A \to R$ to be a simple, differentiable curve on R that is parameterized by arc length. The domain set A of parameter values is an interval in the real line. We also fix a particular network tiling given by the collection of regions $\{R_u\}_{u \in U}$ such that each point in Γ is also in some region R_u. Let

$A_u \triangleq \{p \in A : region(\Gamma(p)) = u\}$ be the domain of Γ in region u. We assume that A_u is convex for every region u; it may be empty for some u. The local part of the curve Γ in region u is the restriction $\Gamma_u : A_u \to R_u$. We write $|A_u|$ for the length of the curve Γ_u. We define the *quantization* of a real number x with quantization constant $\sigma > 0$ as $q_\sigma(x) = \lceil \frac{x}{\sigma} \rceil \sigma$. We fix σ, and write q_u as an abbreviation for $q_\sigma(|A_u|)$, q_{min} for the minimum nonzero q_u, and q_{max} for the maximum q_u.

15.2.1 Problem Statement

Our goal is to design an algorithm for mobile robots such that, once the failures and recoveries cease, within finite time all the robots are located on Γ and as time progresses they eventually become equally spaced on Γ. Formally, if no fail and restart actions occur after time t_0, then:

1. there exists a constant T, such that for each $u \in U$, within time $t_0 + T$ the set of robots located in R_u becomes fixed and its cardinality is roughly proportional to q_u; moreover, if $q_u \neq 0$ then the robots in R_u are located on[1] Γ_u, and

2. in the limit, as time goes to infinity, all robots in R_u are uniformly spaced[2] on Γ_u.

15.2.2 Overview of Solution using the VSA Layer

The VSA Layer is used as a means to coordinate the movement of client nodes, i.e., robots. A VSA controls the motion of the clients in its region by setting and broadcasting target waypoints for the clients: VSA VN_u, $u \in U$, periodically receives information from clients in its region, exchanges information with its neighbors, and sends out a message containing a calculated target point for each client node "assigned" to region u. VN_u performs two tasks when setting the target points: (1) it re-assigns some of the clients that are assigned to

[1]For a given point $x \in R$, if there exists $p \in A$ such that $\Gamma(p) = x$, then we say that the point x is on the curve Γ; abusing the notation, we write this as $x \in \Gamma$.

[2]A sequence $x_1, \ldots, x_n$ of points in R is said to be *uniformly spaced* on a curve Γ if there exists a sequence of parameter values $p_1 < p_2 \ldots < p_n$, such that for each i, $1 \leq i \leq n$, $\Gamma(p_i) = x_i$, and for each i, $1 < i < n$, $p_i - p_{i-1} = p_{i+1} - p_i$.

itself to neighboring VSAs, and (2) it sends a target position on Γ to each client that is assigned to itself. The objective of (1) is to prevent neighboring VSAs from getting depleted of robots and to achieve a distribution of robots over the regions that is proportional to the length of Γ in each region. The objective of (2) is to space the nodes uniformly on Γ within each region. The client algorithm, in turn, receives its current position information from a modified version of RW called RW' and computes a velocity vector for reaching its latest received target point from a VSA.

Each virtual node VN_u uses only information about the portions of the target curve Γ in region u and neighboring regions. For the sake of simplicity, we assume that all client nodes know the complete curve Γ. We could as well have modeled the client nodes in u as receiving external information about the nature of the curve in region u and neighboring regions only.

15.2.3 RW': modified RW

In our solution, we have VSAs direct CNs to new locations. In order to have CNs comply, we need to modify our virtual layer model. In particular, we need to modify RW slightly to allow a mobile node to communicate to the real world automaton what its desired velocity is, rather than allowing RW to nondeterministically choose the node's velocity itself.

We call our modified real world automaton RW'. It is very similar to RW, except for the addition of the velocity action for each mobile node. As before, RW' models system time and mobile node locations. It is an external source of reliable time and location knowledge for physical nodes. The *RW'* TIOA in Figure 15-1 maintains location/ time information and updates mobile nodes with that information.

The new velocity input allows a mobile node to communicate a new desired velocity to RW'. In particular, a velocity$(v)_p$ input prompts RW' to change process p's velocity to v.

As you can see, in addition to the new velocity action, RW' is also different from RW in that it has one additional state variable, vel. In addition, the development of the loc variable for each process p is now as before, unless $vel(p)$ is not $\perp$, in which case $loc(p)$ changes as specified by $vel(p)$:

```
  Signature:
2   Output GPSupdate(l, t)_p, l ∈ R, p ∈ P, t ∈ ℝ^{≥0}
    Input velocity(v)_p, v ∈ ℝ^2, p ∈ P
4
  State:
6   analog now: ℝ^{≥0}, initially 0
    updates(p): 2^{R×ℝ^{≥0}}, for each p ∈ P, initially ∅
8   analog loc(p): R, for each p ∈ P, initially arbitrary
    vel(p):ℝ^2 ∪ {⊥}:|vel(p)| ≤ v_max, for each p ∈ P, initially ⊥
10
  Trajectories:
12  evolve
      d(now) = 1
14    ∀ p ∈ P:
        if vel(p) ≠ ⊥ then d(loc(p)) =
  vel(p) else |d(loc(p))| ≤ v_max
16  stop when
      ∃p ∈ P: ∀⟨l, t⟩ ∈ updates(p): now ≥ t+ ε_sample

  Transitions:
    Output GPSupdate(l, t)_p                           20
    Precondition:
      ∀⟨u, t'⟩ ∈ updates(p): t ≠ t'                    22
      l = loc(p) ∧ t = now
    Effect:                                            24
      updates(p) ← updates(p) ∪ {⟨l, t⟩}
                                                       26
    Input velocity(v)_p
    Effect:                                            28
      vel(p) ← v
```

Figure 15-1: $RW'[v_{max}, \epsilon_{sample}]$.

- $loc : P \to R$ maps each physical node id to a point in R indicating the node's current location. Initially this is arbitrary. We assume that the change in loc for each $p \in P$ is equal to $vel(p)$, unless $vel(p) = \perp$, in which case $loc(p)$ changes at a rate no greater than v_{max}.

- $vel : P \to \mathbb{R}^2 \cup \{\perp\}$ is the velocity of each mobile node. It is initially $\perp$, and is updated via velocity inputs.

The set of reachable states for RW' is the same as for RW, except that vel can be arbitrary.

15.2.4 *CN*: Client Node Algorithm

The algorithm for the client node $CN(\delta)_p$, $p \in P$ (see Figure 15-2) follows a round structure, where rounds begin at times that are multiples of δ. At the beginning of each round, a CN stops moving and sends a cn-update message to its local VSA (that is, the VSA in whose region the CN currently resides). The cn-update message tells the local VSA the CN's *id* and its current location in R. The local VSA then sends a response to the client, i.e., a target-update message. Each such message describes the new target location x_p^* for CN_p, and possibly an assignment to a different region. CN_p computes its velocity vector v_p, based on its current position x_p and its target position x_p^*, as $v_p = (x_p - x_p^*)/||x_p - x_p^*||$

and communicates $v_{max}v_p$ to RW'. As a result then RW' moves the position of CN_p (with maximum velocity) towards x^*_p.

```
   Signature:
2    Input GPSupdate(l, t)_p, l ∈ R, t ∈ ℝ^{≥0}
     Input vrcv(m)_p, m ∈ {target-update} ×(P → R)
4    Output vcast(⟨cn-update, p, l⟩)_p, l ∈ R
     Output velocity(v)_p, v ∈ R^2
6
   State:
8    analog clock: ℝ^{≥0} ∪ {⊥}, initially ⊥
     analog x ∈ R ∪ {⊥}, location, initially ⊥
10   x* ∈ R ∪ {⊥}, target point, initially ⊥
     v ∈ {⊥, 0} ∪ {v : R^2 | |v| = 1}, initially ⊥
12
   Trajectories:
14   evolve
     if clock ≠ ⊥
16      then d(clock) = 1 else d(clock) = 0
     if v ≠ ⊥
18      then d(x) = v · v_max else d(x) = 0
     stop when [x ≠ ⊥ ∧ x* ≠ ⊥
20         ∧ clock mod δ = 0]
        ∨ [x ≠ ⊥ ∧ x* ≠ ⊥ ∧ v||x* − x|| ≠ x* − x]
22      ∨ [(x = x* ∨ x = ⊥ ∨ x* = ⊥) ∧ v ≠ 0]

   Transitions:                                                24
     Input GPSupdate(l, t)_p
     Effect                                                    26
     if ⟨x, clock⟩ ≠ ⟨l, t⟩ ∨
        ||x*-l|| ≥ v_max(δ⌈t/δ⌉-t-d_r) ∨                        28
        x* = ⊥ ∨ t mod δ ∉ (e+2d+2ε, δ-d_r)
     then x, x* ← l; clock ← t                                 30
     v ← ⊥
                                                               32
     Input vrcv(⟨target-update, target⟩)_p
     Effect                                                    34
     if ||target(p)-x|| < v_max(δ⌈clock/δ⌉-clock-d_r)
              ∧ clock mod δ > e + 2d + 2ε                      36
     then x* ← target(p)
                                                               38
     Output vcast(⟨cn-update, p, x⟩)_p
     Precondition                                              40
        x= x ≠ ⊥ ∧ clock mod δ = 0 ∧ x* ≠ ⊥
     Effect                                                    42
        x* ← ⊥
                                                               44
     Output velocity(v)_p
     Precondition                                              46
        v = v_max · (x* − x)/||x* − x||
        ∨ (v = 0 ∧ [x = x* ∨ x* = ⊥ ∨ x = ⊥])                   48
     Effect
        v ← v / v_max                                          50
```

Figure 15-2: Client node $CN(\delta)_p$ automaton.

15.2.5 *VN*: Virtual Stationary Node Algorithm

The algorithm for virtual node $VN(\delta, k, \rho_1, \rho_2)_u$, $u \in U$, appears in Figure 15-3, where $k \in \mathsf{Z}^+$ and $\rho_1, \rho_2 \in (0, 1)$ are parameters of the TIOA. VN_u collects cn-update messages sent at the beginning of the round from CN's located in region R_u, and aggregates the location and round information in a table, M. When $d + \epsilon$ time passes from the beginning of the round, VN_u computes from M the number of client nodes assigned to it that it has heard from in the round, and sends this information in a vn-update message to all of its neighbors.

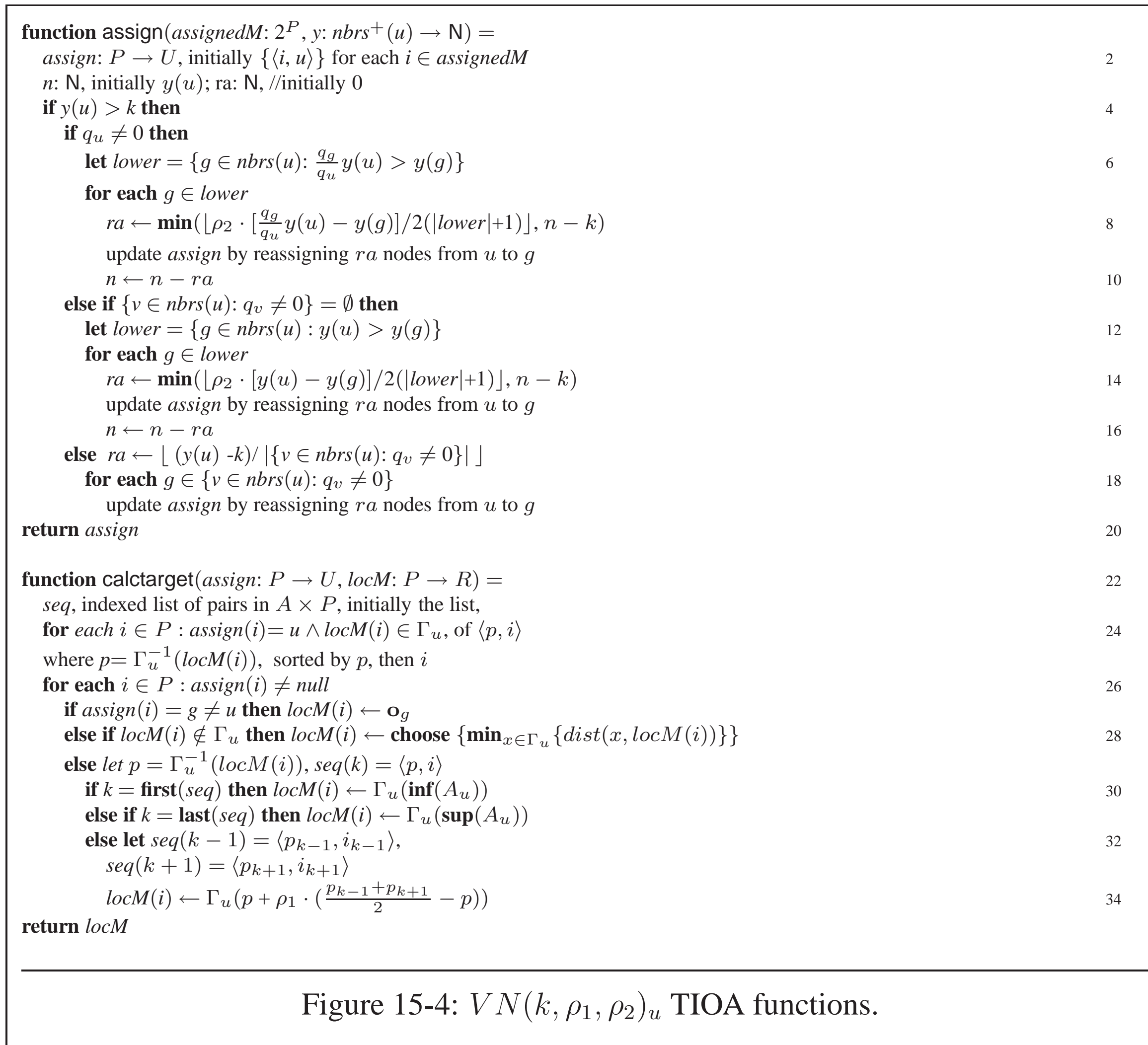
When VN_u receives a vn-update message from a neighboring VSA, it stores the CN population information in a table, V. When $e + d + \epsilon$ time from the sending of its own vn-update passes, VN_u uses the information in its tables M and V about the number of CNs in its and its neighbors' regions to calculate how many CNs assigned to itself should

```
1  Signature:
     Input time(t)_u, t ∈ ℝ^{≥0}
3    Input vrcv(m)_u,
       m ∈ ({cn-update} ×P ×R) ∪ ({vn-update} ×U ×N)
5    Output vcast(m)_u,
       m ∈ ({vn-update} ×{u} ×N) ∪ ({target-update} ×
   (P → R))
7
   State:
9    analog clock: ℝ^{≥0} ∪ {⊥}, initially ⊥.
     M:P→R, initially ∅.
11   V : U → N, initially ∅.

13 Trajectories:
     evolve
15   if clock ≠ t
       then d(clock) = 1 else d(clock) = 0
17   stop when Any precondition is satisfied.

19 Transitions:
     Input time(t)_u
21   Effect
     if clock ≠ t ∨ t mod δ ∉ (0, e + 2d + 2ϵ]
23     then M, V ← ∅; clock ← t

   Input vrcv(⟨cn-update, id, loc⟩)_u
   Effect                                              26
   if u = region(loc) ∧ clock mod δ ∈ (0, d]
     then M(id) ← loc; V ← ∅                           28

   Output vcast(⟨vn-update, u, n⟩)_u                   30
   Precondition
     clock mod δ = d+ϵ                                 32
     n= |M|≠ 0∧ V≠ {⟨u, n⟩}
   Effect                                              34
     V ← {⟨u, n⟩}
                                                       36
   Input vrcv(⟨vn-update, id, n⟩)_u
   Effect                                              38
     if id ∈ nbrs(u) then V(id) ← n
                                                       40
   Output vcast(⟨target-update, target⟩)_u
   Precondition                                        42
     clock mod δ = e + 2d + 2ϵ ∧ M ≠ ∅
     target = calctarget(assign(id(M), V), M)          44
   Effect
     M, V ← ∅                                          46
```

Figure 15-3: $VN(\delta, k, \rho_1, \rho_2)_u$ TIOA, with parameters: safety k, and damping ρ_1, ρ_2.

be reassigned and to which neighbor. This is done through the assign function, and these assignments are then used to calculate new target points for local CNs through the calctarget function (see Figure 15-4). The choice of point assignments draws on the intuition for solutions to what members of the control community call the *consensus problem* [77], where several agents try to converge at a point, usually their average. One standard way for solving continuous consensus is for the agents to interact pair-wise and replace their current values with their average. Our assignment algorithm is similar, but more complicated due to a policy of maintaining a minimum number of agents in an alive region (to help prevent alive VSAs from failing), the fact that each region has multiple neighboring regions with which to coordinate, and the effects of quantization.

If the number of CNs assigned to VN_u exceeds the minimum *safe number* k, then assign may reassign some CNs to neighbors, based on the number of CNs at those neighbors. Let In_u denote the set of neighboring VSAs of VN_u that are on the curve Γ and $y_u(g)$, denote the number $num(V_u(g))$ of CNs assigned to VN_g, where g is either u or a neighbor of u. If $q_u \neq 0$, meaning VN_u is on the curve then we let $lower_u$ denote the subset of $nbrs(u)$ that are on the curve and have fewer assigned CNs than VN_u has after



```
function assign(assignedM: 2^P, y: nbrs+(u) → N) =
  assign: P → U, initially {⟨i, u⟩} for each i ∈ assignedM                      2
  n: N, initially y(u); ra: N, //initially 0
  if y(u) > k then                                                                4
    if q_u ≠ 0 then
      let lower = {g ∈ nbrs(u): (q_g/q_u) y(u) > y(g)}                           6
      for each g ∈ lower
        ra ← min(⌊ρ2 · [(q_g/q_u) y(u) − y(g)]/2(|lower|+1)⌋, n − k)             8
        update assign by reassigning ra nodes from u to g
        n ← n − ra                                                               10
    else if {v ∈ nbrs(u): q_v ≠ 0} = ∅ then
      let lower = {g ∈ nbrs(u) : y(u) > y(g)}                                   12
      for each g ∈ lower
        ra ← min(⌊ρ2 · [y(u) − y(g)]/2(|lower|+1)⌋, n − k)                      14
        update assign by reassigning ra nodes from u to g
        n ← n − ra                                                               16
    else  ra ← ⌊ (y(u) -k)/ |{v ∈ nbrs(u): q_v ≠ 0}| ⌋
      for each g ∈ {v ∈ nbrs(u): q_v ≠ 0}                                       18
        update assign by reassigning ra nodes from u to g
return assign                                                                    20

function calctarget(assign: P → U, locM: P → R) =                               22
  seq, indexed list of pairs in A × P, initially the list,
  for each i ∈ P : assign(i)= u ∧ locM(i) ∈ Γ_u, of ⟨p, i⟩                       24
  where p= Γ_u^{-1}(locM(i)),  sorted by p, then i
  for each i ∈ P : assign(i) ≠ null                                             26
    if assign(i) = g ≠ u then locM(i) ← o_g
    else if locM(i) ∉ Γ_u then locM(i) ← choose {min_{x∈Γ_u} {dist(x, locM(i))}}  28
    else let p = Γ_u^{-1}(locM(i)), seq(k) = ⟨p, i⟩
      if k = first(seq) then locM(i) ← Γ_u(inf(A_u))                            30
      else if k = last(seq) then locM(i) ← Γ_u(sup(A_u))
      else let seq(k − 1) = ⟨p_{k−1}, i_{k−1}⟩,                                  32
        seq(k + 1) = ⟨p_{k+1}, i_{k+1}⟩
        locM(i) ← Γ_u(p + ρ1 · ((p_{k−1}+p_{k+1})/2 − p))                       34
return locM
```

Figure 15-4: $VN(k, \rho_1, \rho_2)_u$ TIOA functions.

normalizing with $\frac{q_g}{q_u}$. For each $g \in lower_u$, VN_u reassigns the smaller of the following two quantities of CNs to VN_g: (1) $ra = \rho_2 \cdot [\frac{q_g}{q_u} y_u(u) - y_u(g)]/2(|lower_u| + 1)$, where $\rho_2 < 1$ is a *damping factor*, and (2) the remaining number of CNs over k still assigned to VN_u.

If $q_u = 0$, meaning VN_u is not on the curve, and VN_u has no neighbors on the curve (lines 11–15), then we let $lower_u$ denote the subset of $nbrs(u)$ with fewer assigned CNs than VN_u. For each $g \in lower_u$, VN_u reassigns the smaller of the following two quantities of CNs: (1) $ra = \rho_2 \cdot [y_u(u) - y_u(g)]/2(|lower_u| + 1)$ and (2) the remaining number of CNs over k still assigned to VN_u. VN_u is on a *boundary* if $q_u = 0$, but there is a $g \in nbrs(u)$ with $q_g \neq 0$. In this case, $y_u(u) - k$ of VN_u's CNs are assigned equally to neighbors in In_u (lines 17–19).

The calctarget function assigns to every CN_p assigned to VN_u a target point $locM_u(p)$

in region R_g, where $g = u$ or it is one of u's neighbors. The target point $locM_u(p)$ is computed as follows: If CN_p is assigned to VN_g, $g \neq u$, then its target is set to the center $\mathbf{o}_g$ of region g (line 27); if CN_p is assigned to VN_u but is not located on the curve Γ_u then its target is set to the nearest point on the curve, nondeterministically choosing one if there are several (line 28); if CN_p is either the first or last client node on Γ_u then its target is set to the corresponding endpoint of Γ_u (lines 30–31); if CN_p is on the curve but is not the first or last client node then its target is moved to the mid-point of the locations of the preceding and succeeding CNs on the curve (line 34). For the last two computations a sequence seq of nodes on the curve sorted by curve location is used (line 25).

Lastly, VN_u broadcasts new waypoints for the round via a target-update message to its CNs.

15.2.6 *MC*: Complete System

Define *MC* to be the element of $VAlgs$, the set of VSA layer algorithms (Definition 7.3), where for each $p \in P$, $MC(p) = CN_p$, and for each $u \in U$, $MC(u) = VN_u$.

The complete system is then $VLayer'[MC]$, which is exactly the same as $VLayer[MC]$, the VSA layer instantiated with MC (Definition 7.4), except that RW is replaced with RW':

- RW',
- VW,
- $VBcast$,
- $Fail(VBDelay_p \| CN_p)$, one for each $p \in P$, and
- $Fail(VBDelay_u \| VN_u)$, one for each $u \in U$.

Recall that $Fail(\mathcal{A})$ denotes the fail-transformed version of TIOA $\mathcal{A}$ (see Chapter 5).

Round length

Given the maximum Euclidean distance, r, between points in neighboring regions, it can take up to $\frac{r}{v_{max}}$ time for a client to reach its target. Also, after the client arrives in the

region it was assigned to, it could find the local VSA has failed. Let d_r be the time it takes a VSA to start up, once a new node enters the region and assuming no nodes in the region fail or leave until after the startup (notice that such a constant may not exist; however, under the assumption that executions of the virtual layer are in the execution fragment set S described in Definition 11.12, such a constant does exist and is equal to $d + t_{slice}$). To ensure a round is long enough for a client node to send the cn-update, allow VNs to exchange information, allow clients to receive a target-update message and arrive at new assigned target locations, and be sure virtual nodes are alive in their region before a new round begins, we require that δ, the round length parameter, satisfies $\delta > 2e + 3d + 2\epsilon + r/v_{max} + d_r$.

15.3 Correctness of the Algorithm

In this section, we show that *starting from an initial state* and assuming that executions of the virtual layer satisfy the properties of set S in Definition 11.12 (where S describes execution fragments of the virtual layer that satisfy certain properties with respect to when a fail or restart of a VSA is allowed to occur and when a VSA restart is guaranteed to occur), the system described in 15.2.2 satisfies the requirements specified in Section 15.2.1. The proofs of the results in this section parallel those presented in [66], albeit the semantics of the Virtual Layer used here is different (the virtual nodes used in [66] were untimed and hence dependent on the timing of client node messages to complete their tasks). The proofs still look similar since the reasoning both here and in [66] uses the same round-based structure. In the following section we show self-stabilization.

We define round t as the interval of time $[\delta(t-1), \delta \cdot t)$. That is, round t begins at time $\delta(t-1)$ and is completed by time $\delta \cdot t$. We say $CN_p, p \in P$, is *active* in round t if node p is not failed throughout round t. A VN_u, $u \in U$, is *active* in round t, $t > 0$ if it is alive from the beginning of round t until its $VBDelay$ performs a vcast′ of a target-update message. By definition none of the VN s is active in the first round. We also define the following notation:

- $In(t) \subseteq U$ is the subset of VN ids that are active in round t and $q_u \neq 0$;

- $Out(t) \subseteq U$ is the subset of VNs that are active in round t and $q_u = 0$;

- $C(t) \subseteq P$ is the subset of active CNs at round t;

- $C_{in}(t) \subseteq P$ is the set of active CNs located in regions with id in $In(t)$ at the beginning of round t;

- $C_{out}(t) \subseteq P$ is subset of active CNs located in regions with id in $Out(t)$ at the beginning of round t.

For every pair of regions u, w and for every round t, we define $y(w,t)_u$ to be the value of $V(w)_u$ (i.e., the number of clients u believes are available in region w) immediately prior to VN_u performing a vcast_u in round t, i.e., at time $e+2d+2\epsilon$ after the beginning of round t. If there are no new client failures or recoveries in round t, then for every pair of regions $u, w \in nbrs^+(v)$, we can conclude that $y(v,t)_u = y(v,t)_w$, which we denote simply as $y(v,t)$. We define $\rho_3 \triangleq \frac{q_{max}^2}{(1-\rho_2)\sigma}$. The rate ρ_3 effects the rate of convergence, and will be used in the analysis. Notice that $\rho_3 > 1$. Notice that for any $v, w \in nbrs(u) \cup \{u\}$, in the absence of failures and recoveries of CNs in round t, $y_{v,t} = y_{w,t}$; we write this simply as $y_h(t)$.

15.3.1 Approximately Proportional Distribution

For the rest of this section we fix a particular round number t_0 and assume that, for all $p \in P$, no fail_p or $\mathsf{restart}_p$ events occur at or after round t_0. We also assume that all executions of *VLayer'[MC]* satisfy the properties of S in Definition 11.12. The first lemma states some basic facts about the assign function.

Lemma 15.1 *In every round $t \geq t_0$:*

1. *If $y(u,t) \geq k$ for some $u \in U$, then $y(u,t+1) \geq k$;*

2. *$In(t) \subseteq In(t+1)$;*

3. *$Out(t) \subseteq Out(t+1)$.*

Proof: We fix round $t \geq t_0$.

1. From line 4 of the assign function (Figure 15-4) it is clear that VN_u, $u \in U$, reassigns some of its CN s in round t only if $y(u,t) > k$. And if a CN is not reassigned and does not fail, it remains active in the same region.

2. For any VN_u, $u \in In(t)$, if $y(u,t) < k$ then VN_u does not reassign CN s, and $y(u,t+1) = y(u,t)$. Otherwise, from line 8 of Figure 15-4 it follows that $y(u,t+1) \geq k$. In both cases $u \in In(t+1)$. (Since all processes that move do so after receiving a target-update message from their region, an alive VSA won't fail in a round until after its vcast′ of a target-update has occurred. Also, by our assumption on the size of δ, it is obvious that by the start of the next round the VSA will again be alive since no processes die or leave in the first d portion of a round.)

3. For any VN_u, $u \in Out(t)$, if $y(u,t) < k$ then VN_u does not reassign CN s, and $y(u,t+1) = y(u,t)$. Otherwise, from line 14 and line 17 of Figure 15-4 it follows that $y(u,t+1) \geq k$. In both cases $u \in Out(t+1)$. (This follows the reasoning of the prior item.)

■

We now identify a round $t_1 \geq t_0$ after which the set of regions $In(t)$ and $Out(t)$ remain fixed.

Lemma 15.2 *There exists a round $t_1 \geq t_0$ such that for every round $t \in [t_1, t_1 + (1 + \rho_3)m^2n^2]$:*

1. $In(t) = In(t_1)$*;*

2. $Out(t) = Out(t_1)$*;*

3. $C_{in}(t) \subseteq C_{in}(t+1)$*; and*

4. $C_{out}(t+1) \subseteq C_{out}(t)$.

Proof: By Lemma 15.1, Part 2, we know that the set $In(t) \subseteq U$ is non-decreasing as t increases. From Part 3, we know that set $Out(t) \subseteq U$ is non-decreasing as t increase. Since U is finite, we conclude from this that there is some round t_1 after which no new regions $u \in U$ are added to either $In(t)$ or $Out(t)$. Thus we have satisfied Parts 1 and 2. Notice that this occurs no later than round $t_0 + 2m^2 \cdot (1 + \rho_3)m^2n^2$.

For Part 3, consider a client CN_p, $p \in C_{in}(t)$, that is currently assigned in round t to VN_u, $u \in In(t)$. From lines 5–9 of Figure 15-4 we see that CN_p is assigned to some VN_w, $w \in nbrs^+(u)$ where $q_w \neq 0$. If VN_w is inactive in round $t+1$, then client CN_p remains in VN_w until it becomes active, resulting in VN_w being added to $In(t)$, thus contradicting the fact that for every round $t' \geq t_1$, $In(t') = In(t_1)$. We conclude that VN_w is active in round t, and hence round $t+1$, from which the claim follows.

For Part 4, notice that since there are no failures and recoveries of CN s, $C(t) = C(t+1)$. By definition, $C_{in}(t) \cup C_{out}(t) = C(t)$, $C_{in}(t) \cap C_{out}(t) = \emptyset$, and $C_{in}(t+1) \cup C_{out}(t+1) = C(t+1)$, $C_{in}(t+1) \cap C_{out}(t+1) = \emptyset$. The result follows from Part (3). ■

Fix t_1 for the rest of this section such that it satisfies Lemma 15.2. The next lemma states that eventually, regions bordering on the curve stop assigning clients to regions that are on the curve. That is, assume that u is a region where $q_u = 0$, but that u has a neighbor v where $q_v \neq 0$; then, eventually, from some round onwards, u never again assigns clients to v.

Lemma 15.3 *There exists some round $t_2 \in [t_1, t_1 + (1+\rho_3)m^2n^2]$ such that for every round $t \in [t_2, t_2 + (1+\rho_3)m^2n]$: if $u \in Out(t)$ and $v \in In(t)$ and if u and v are neighboring regions, then u does not assign any clients to v in round t.*

Proof: Notice that if u assigns a client to v, then C_{out} decreases by one. During the interval $[t_1, t_1 + (1+\rho_3)m^2n^2]$, we know that C_{out} is non-increasing by Lemma 15.2. Thus, eventually, there is some round t_2 after which either $C_{out} = \emptyset$ or after which no further clients are assigned from a region $Out(\cdot)$ to a region $In(\cdot)$. Since there are at most n clients, we can conclude that this occurs at latest by round $t_1 + n \cdot [(1+\rho_3)m^2n]$. ■

Fix t_2 for the rest of this section such that it satisfies Lemma 15.3. Lemma 15.2 implies that in every round $t \geq t_1$, $In(t) = In(t_1)$ and $Out(t) = Out(t_1)$; we denote these simply

as In and Out. The next lemma states a key property of the assign function after round t_1. For a round $t \geq t_1$, consider some VN_u, $u \in Out(t)$, and assume that VN_w is the neighbor of VN_u assigned the most clients in round t. Then we can conclude that VN_u is assigned no more clients in round $t+1$ than VN_w is assigned in round t. A similar claim holds for regions in $In(t)$, but in this case with respect to the *density* of clients with respect to the quantized length of the curve. The proof of this lemma is based on careful analysis of the behavior of the assign function.

Lemma 15.4 *In every round $t \in [t_2, t_2 + (1+\rho_3)m^2 n]$, for $u, v \in U$ and $u \in nbrs(v)$:*

1. *If $u, v \in Out(t)$ and $y(v,t) = \max_{w \in nbrs(u) \cap Out(t)} y(w,t)$ and $y(u,t) < y(v,t)$, then $y(u,t+1) < y(v,t)$.*

2. *If $u, v \in In(t)$ and $y(v,t)/q_v = \max_{w \in nbrs(u) \cap In(t)} [y(w,t)/q_w]$ and $y(u,t)/q_u < y(v,t)/q_v$, then:*
$$\frac{y(u,t+1)}{q_u} \leq \frac{y(v,t)}{q_v} - (1-\rho_2)\frac{\sigma}{q_{max}^2} .$$

Proof: For Part 1, fix u, v and t, as in the statement of the lemma. Consider some region w that is a neighbor of u and that assigns clients to u in round $t+1$. Since $q_u = 0$, notice that w assigns clients to u only if the conditions of lines 11–16 in Figure 15-4 are met. This implies that $w \in Out(t)$, and hence $y(w,t) \leq y(v,t)$, by assumption. We can also conclude that $lower_w \geq 1$, as w assigns clients to u only if $u \in lower_w$. Finally, from line 14 of Figure 15-4, we observe that the number of clients that are assigned to u by w in round t is at most:

$$\frac{\rho_2 \left[y(w,t) - y(u,t)\right]}{2(|lower_w(t)|+1)} \leq \frac{\rho_2 \left[y(v,t) - y(u,t)\right]}{4}$$

Since u has at most four neighbors, we conclude that it is assigned at most

$\rho_2 \left[y(v,t) - y(u,t)\right]$ clients. Since $\rho_2 < 1$ and $y(u,t) < y(v,t)$, this implies that:

$$\begin{aligned} y(u,t+1) &\leq y(u,t) + \rho_2 \left[y(v,t) - y(u,t)\right] \\ &\leq \rho_2 \cdot y(v,t) + (1-\rho_2) y(u,t) \\ &< \rho_2 \cdot y(v,t) + (1-\rho_2) y(v,t) \\ &< y(v,t) \,. \end{aligned}$$

For Part 2, as in Part 1, fix u, v and t as in the lemma statement. Recall we have assumed that $y(u,t)/q_u < y(v,t)/q_v$. We begin by showing that, due to the manner in which the curve is quantized, $y(u,t)/q_u \leq y(v,t)/q_v - \sigma/q_{max}^2$. Since q_u is defined as $\lceil P_u/\sigma \rceil \sigma$, and since q_v is defined as $\lceil P_v/\sigma \rceil \sigma$, we notice that, by assumption:

$$y(u,t) \left\lceil \frac{P_v}{\sigma} \right\rceil \sigma < y(v,t) \left\lceil \frac{P_u}{\sigma} \right\rceil$$

We divide both sides by σ, and since both sides are integral, we exchange the '$<$' with a '$\leq$':

$$y(u,t) \left\lceil \frac{P_v}{\sigma} \right\rceil \leq y(v,t) \left\lceil \frac{P_u}{\sigma} \right\rceil - 1$$

From this we conclude:

$$\frac{y(u,t)}{\left\lceil \frac{P_u}{\sigma} \right\rceil} \leq \frac{y(v,t)}{\left\lceil \frac{P_v}{\sigma} \right\rceil} - \frac{\sigma^2}{q_u q_v}$$

Dividing everything by σ, and bounding q_u and q_v by q_{max}, we achieve the desired calculation.

Now, consider some region w that is a neighbor of u and that assigns clients to u in round $t+1$. First, notice that $w \notin Out(t)$, since by Lemma 15.3, no clients are assigned from an Out region to an In region after round t_2 (prior to $t_2 + (1+\rho_3)m^2 n$). Thus, w assigns clients to u only if the conditions of lines 5–10 in Figure 15-4 are met. This implies that $w \in In(t)$, and hence $y(w,t)/q_w \leq y(v,t)/q_v$, by assumption. We can also conclude that $lower_w \geq 1$, as w assigns clients to u only if $u \in lower_w$. Finally, from line 8 of Figure 15-4, we observe that the number of clients that are assigned to u by w in round t is

at most:

$$\frac{\rho_2\left[\left(\frac{q_u}{q_w}\right)y(w,t)-y(u,t)\right]}{2(|lower_w(t)|+1)} \leq \frac{\rho_2\left[\left(\frac{q_u}{q_v}\right)y(v,t)-y(u,t)\right]}{4}$$

Since u has at most four neighbors, we conclude that it is assigned at most $\rho_2\left[(q_u/q_v)y(v,t)-y(u,t)\right]$ clients. This implies that:

$$\begin{aligned} y(u,t+1) &\leq y(u,t)+\rho_2\left[\left(\frac{q_u}{q_v}\right)y(v,t)-y(u,t)\right] \\ &\leq \rho_2\left(\frac{q_u}{q_v}\right)\cdot y(v,t)+(1-\rho_2)\,y(u,t) \end{aligned}$$

Thus, dividing everything by q_u, and recalling that $y(u,t)/q_u \leq y(v,t)/q_v - \sigma/q_{max}^2$:

$$\begin{aligned} \frac{y(u,t+1)}{q_u} &\leq \rho_2\left(\frac{y(v,t)}{q_v}\right)+(1-\rho_2)\cdot\left(\frac{y(u,t)}{q_u}\right) \\ &\leq \rho_2\left(\frac{y(v,t)}{q_v}\right)+(1-\rho_2)\cdot\left(\frac{y(v,t)}{q_v}-\frac{\sigma}{q_{max}^2}\right) \\ &\leq \frac{y(v,t)}{q_v}-(1-\rho_2)\frac{\sigma}{q_{max}^2} \end{aligned}$$

■

The next lemma states that there exists a round T_{out} such that in every round $t \geq T_{out}$, the set of CNs assigned to region $u \in Out(t)$ does not change.

Lemma 15.5 *There exists a round $T_{out} \in [t_2, t_2+m^2n$ such that in any round $t \geq T_{out}$, the set of CNs assigned to VN_u, $u \in Out(t)$, is unchanged.*

Proof: First, we show that there exists some round T_{out} such that the aggregate number of CN s assigned to VN_u remains the same in both T_{out} and $T_{out}+1$ for all $u \in Out(t_2)$. We then show that the actual assignment of individual clients remains the same in T_{out} and $T_{out}+1$.

We consider a vector $E(t)$ that represents the distribution of clients among regions in $Out(t)$. That is, the first element in $E(t)$ represents the largest number of clients in

any region; the second element in $E(t)$ represents the second largest number of clients in any region; and so forth. We then argue that, compared lexicographically, $E(t+1) \leq E(t)$. Since the elements in $E(t)$ are integers, we conclude from this that eventually the distribution of clients becomes stables and ceases to change.

We proceed to define $E(t)$ as follows for $t \geq t_2$. Let $N_{out} = |Out|$. Let $\Pi(t)$ be a permutation of Out that orders the regions by the number of assigned clients, i.e., if u precedes v in $\Pi(t)$, then $y(u,t) \leq y(v,t)$. When we say that some region u has index k, we mean that $\Pi(t)_k = u$. Define $E(t)$ as follows:

$$E(t) = \langle y(\Pi(t)_{N_{out}}, t), y(\Pi(t)_{N_{out}-1}, t), \ldots, y(\Pi(t)_1, t) \rangle \ .$$

We use the notation $E(t)_\ell$ to refer to the ℓth component of $E(t)$ counting from the right, i.e., it refers to $\Pi(t)_\ell$. Any two vectors $E(t)$ and $E(t+1)$ can be compared lexicographically, examining each of the elements in turn from left to right, i.e., largest to smallest.

We now consider some round $t \in [t_2, t_2 + m^2 n]$, and show that $E(t) \geq E(t+1)$. Consider the case where $E(t) \neq E(t+1)$, and let u be the region with maximum index that assigns clients to another region. Let k be the index of region u.

First, we argue that for every region v with index $\leq k$, we can conclude that $y(v, t{+}1) < y(u,t)$. Consider some particular region v. Notice that v has no neighbors in Out that are assigned more than $y(u,t)$ clients in round t; otherwise, such a neighbor would assign clients to v, contradicting our choice of u. Thus, by Lemma 15.4, Part 1, we can conclude that $y(v, t{+}1) < y(u,t)$ (as long as $t \in [t_2, t_2{+}2m^2 n]$, which we will see to be sufficient).

Since this implies that there are at least k regions assigned fewer than $y(u,t) = E(t)_k$ clients in round $t+1$, we can conclude that $E(t+1)_k < E(t)_k$. In order to show that $E(t+1) < E(t)$, it remains to show that for every $k' > k$, $E(t)_{k'} = E(t+1)_{k'}$.

Consider some region v with index $> k$. By our choice of u, it is clear that v is not assigned any clients by a region with index $> k$. It is also easy to see that v is not assigned any clients by a region w with index $\leq k$, since $y(v,t) \geq y(u,t) \geq y(w,t)$; as per line 12, region w does not assign any clients to a region with $\geq y(w,t)$ clients. Thus no new clients are assigned to region v. Moreover, by choice of u, region v assigns none of its clients

elsewhere. Finally, since $t \geq t_0$, none of the clients fail. Thus, $y(v,t) = y(v,t+1)$.

Since the preceding logic holds for all $N_{out} - k + 1$ regions with index $> k$, and all have more than $y(u,t) > y(u,t+1)$ clients, we conclude that for every $k' > k$, $E(t)_{k'} = E(t+1)_{k'}$, implying that $E(t) > E(t+1)$, as desired.

Since $E(\cdot)$ is non-increasing, and since it is bounded from below by the zero vector, we conclude that eventually there is a round T_{out} such that for all $t \geq T_{out}$, $E(t) = E(t+1)$.

Now suppose the set of clients assigned to region u changes in some round $t \geq T_{out}$. The only way the set of clients assigned to region u could change, without changing $y(u,t)$ and the set C_{out}, is if there existed a cyclic sequence of VN s with ids in Out in which each VN gives up $c > 0$ CN s to its successor VN in the sequence, and receives c CN s from its predecessor. However, such a cycle of VN s cannot exist because the $lower$ set imposes a strict partial ordering on the VN s.

Finally, we observe that if $E(t) = E(t+1)$ for any t, then the assignment of clients does not change from that point onwards: since all the clients remained in the same regions in $E(t)$ and $E(t+1)$, we can conclude that the $assign$ function produced the same assignment in $E(t+1)$ as in $E(t)$. Since the vector $E(\cdot)$ has at most m^2 elements, each with at most n values, we can conclude that T_{out} is at most $m^2 n$ rounds after t_2. ■

For the rest of the section we fix T_{out} to be the first round after t_0, at which the property stated by Lemma 15.5 holds. Lemma 15.5, together with Lemmas 15.1, 15.2, and 15.3, imply that in every round $t \geq T_{out}$, $C_{In}(t) = C_{In}(t_1)$ and $C_{Out}(t) = C_{Out}(t_1)$; we denote these simply as C_{In} and C_{Out}. The next lemma states a property similar to that of Lemma 15.5 for VN_u, $u \in In$, and the argument is similar to the proof of Lemma 15.5, and uses Part (2) of Lemma 15.4.

Lemma 15.6 *There exists a round $T_{stab} \in [T_{out}, T_{out} + \rho_3 m^2 n]$ such that in every round $t \geq T_{stab}$, the set of CNs assigned to VN_u, $u \in In$, is unchanged.*

Proof: We proceed to define $E(t)$ as follows for $t \geq T_{out}$. Let $N_{in} = |In|$. Let $\Pi(t)$ be a permutation of In that orders the regions by the density of assigned clients, i.e., if u precedes v in $\Pi(t)$, then $y(u,t)/q_u \leq y(v,t)/q_v$. When we say that some region u has

index k, we mean that $\Pi(t)_k = u$. Define $E(t)$ as follows:

$$E(t) = \left\langle \frac{y(\Pi(t)_{N_{in}}, t)}{q_{\Pi(t)_{N_{in}}}}, \frac{y(\Pi(t)_{N_{in}-1}, t)}{q_{\Pi(t)_{N_{in}-1}}}, \ldots, \frac{y(\Pi(t)_1, t)}{q_{\Pi(t)_1}} \right\rangle .$$

We use the notation $E(t)_\ell$ to refer to the ℓth component of $E(t)$ counting from the right, i.e., it refers to $\Pi(t)_\ell$. Any two vectors $E(t)$ and $E(t+1)$ can be compared lexicographically, examining each of the elements in turn from left to right, i.e., largest to smallest.

We now consider some round $t \geq T_{out}$, and show that $E(t) \geq E(t+1)$. Consider the case where $E(t) \neq E(t+1)$, and let u be the region with maximum index that assigns clients to another region. Let k be the index of region u.

First, we argue that for every region v with index $\leq k$, we can conclude that $y(v, t+1)/q_v \leq y(u,t)/q_u - \zeta$ for some constant ζ. Consider some particular region v. Notice that v has no neighbors in In that have density greater than $y(u,t)/q_u$ in round t; otherwise, such a neighbor would assign clients to v, contradicting our choice of u. Thus, by Lemma 15.4, Part 2, we can conclude that $y(v,t+1)/q_v \leq y(u,t)/q_u - \zeta$ where $\zeta = (1-\rho_2)\frac{\sigma}{q_{max}^2}$ (as long as $t \in [t_2, t_2 + (1+\rho_3)m^2 n]$, which we will see to be sufficient).

Since this implies that there are at least k regions assigned fewer than $y(u,t) = E(t)_k$ clients in round $t+1$, we can conclude that $E(t+1)_k \leq E(t)_k - \zeta$. In order to show that $E(t+1) < E(t)$, it remains to show that for every $k' > k$, $E(t)_{k'} = E(t+1)_{k'}$.

Consider some region v with index $> k$. By our choice of u, it is clear that v is not assigned any clients by a region with index $> k$. It is also easy to see that v is not assigned any clients by a region w with index $\leq k$, since $y(v,t)/q_v \geq y(u,t)/q_u \geq y(w,t)/q_w$; as per line 6, region w does not assign any clients to a region with a density $\geq y(w,t)/q_w$. Thus no new clients are assigned to region v. Moreover, by choice of u, region v assigns none of its clients elsewhere. Finally, since $t \geq t_0$, none of the clients fail. Thus, $y(v,t)/q_v = y(v,t+1)/q_v$.

Since the preceding logic holds for all $N_{in} - k + 1$ regions with index $> k$, and all have more than $y(u,t)/q_u$ clients, we conclude that for every $k' > k$, $E(t)_{k'} = E(t+1)_{k'}$, implying that $E(t) > E(t+1)$, as desired.

Since $E(\cdot)$ is non-increasing, and since it decreases by at least a constant ζ in every

round in which it decreases, and since it is bounded from below by the zero vector, we conclude that eventually there is a round T_{stab} such that for all $t \geq T_{stab}$, $E(t) = E(t+1)$.

Now suppose the set of clients assigned to region u changes in some round $t \geq T_{stab}$. The only way the set of clients assigned to region u could change, without changing $y(u,t)/q_u$ and the set C_{in}, is if there existed a cyclic sequence of VN s with ids in In in which each VN gives up $c > 0$ CN s to its successor VN in the sequence, and receives c CN s from its predecessor. However, such a cycle of VN s cannot exist because the $lower$ set imposes a strict partial ordering on the VN s.

Finally, we observe that if $E(t) = E(t+1)$ for any t, then the assignment of clients does not change from that point onwards: since all the clients remained in the same regions in $E(t)$ and $E(t+1)$, we can conclude that the $assign$ function produced the same assignment in $E(t+1)$ as in $E(t)$. Since the vector $E(\cdot)$ has at most m^2 elements, each with at most $n\frac{q_{max}^2}{(1-\rho)\sigma}$ values, we can conclude that T_{stab} is at most $\rho_3 m^2 n$ rounds after T_{out}, and hence at most $(1+\rho_3)m^2 n$ rounds after t_2, as needed. ■

The following bounds the total number of clients located in regions with ids in Out to be $O(m^3)$.

Lemma 15.7 *In every round $t \geq T_{out}$, $|C_{out}(t)| = O(m^3)$.*

Proof: From Lemma 15.5, the set of CN s assigned to each VN_u, $u \in Out(t)$, is unchanged in every round $t \geq T_{out}$. This implies that in any round $t \geq T_{out}$, the number of CN s assigned by VN_u to any of its neighbors is 0. Therefore, from line 17 of Figure 15-4, for any boundary VN_v, $(y(v,t)-k)/|In_v| < 1$. Recall that In_v is the (constant) set of neighbors of v with quantized curve length $\neq 0$. Since $|In_v| \leq 4$, $y(v,t) < 4+k$.

From line 14 of Figure 15-4, for any non-boundary VN_v, $v \in Out(t)$, if v is 1-hop away from a boundary region u, then $\frac{\rho_2(y(v,t)-y(u,t))}{2(|lower_v(t)|+1)} < 1$. Since $|lower_v(t)| \leq 4$, $y(v,t) \leq \frac{10}{\rho_2} + 4 + k$. Inducting on the number of hops, the maximum number of clients assigned to a VN_v, $v \in Out(t)$, at ℓ hops from the boundary is at most $\frac{10\ell}{\rho_2} + k + 4$. Since for any ℓ, $1 \leq \ell \leq 2m-1$, there can be at most m VN s at ℓ-hop distance from the boundary, summing gives $|C_{out}| \leq (k+4)(2m-1)m + \frac{10m^2(2m-1)}{\rho_2} = O(m^3)$. ■

For the rest of the section we fix T_{stab} to be the first round after T_{out}, at which the property stated by Lemma 15.6 holds. Lemma 15.8 states that the number of clients assigned to each VN_u, $u \in In$, in the stable assignment after T_{stab} is proportional to q_u within a constant additive term. The proof follows by induction on the number of hops from between any pair of VNs.

Lemma 15.8 *In every round $t \geq T_{stab}$, for $u, v \in In(t)$:*

$$\left| \frac{y(u,t)}{q_u} - \frac{y(v,t)}{q_v} \right| \leq \left\lceil \frac{10(2m-1)}{q_{min}\rho_2} \right\rceil .$$

Proof: Consider a pair of VN s for neighboring regions u and v, $u, v \in In$. Assume without loss of generality that $y(u,t) \geq y(v,t)$. From line 8 of Figure 15-4, it follows that $\rho_2(\frac{q_v}{q_u} y(u,t) - y(v,t)) \leq 2(|lower_u(t)| + 1)$. Since $|lower_u(t)| \leq 4$, $|\frac{y(u,t)}{q_u} - \frac{y(v,t)}{q_v}| \leq \frac{10}{q_v \rho_2}$ $\leq \frac{10}{q_{min}\rho_2}$. By induction on the number of hops from 1 to $2m-1$ between any two VN s, the result follows. ■

15.3.2 Uniform Spacing

From line 28 of Figure 15-4, it follows that by the beginning of round $T_{stab}+2$, all CNs in C_{in} are located on the curve Γ. Thus, the algorithm satisfies our first goal. The next lemma states that the locations of the CNs in each region u, $u \in In$, are uniformly spaced on Γ_u in the limit, and it is proved by analyzing the behavior of calctarget as a discrete time dynamical system.

Lemma 15.9 *Consider a sequence of rounds $t_1 = T_{stab}, \ldots, t_n$. As $n \to \infty$, the locations of CNs in u, $u \in In$, are uniformly spaced on Γ_u.*

Proof: From Lemma 15.6 we know that the set of CN s assigned to each VN_u, $u \in In$, remains unchanged. Then, at the beginning of round t_2, every CN assigned to VN_u is located in region u and is on the curve Γ_u. Assume without loss of generality that VN_u is assigned at least two CN s. Then, at the beginning of round t_3, one CN is positioned at

each endpoint of Γ_u, namely at $\Gamma_u(inf(P_u))$ and $\Gamma_u(sup(P_u))$. From lines 30–31 of Figure 15-4, we see that the target points for these *endpoint* CN s are not changed in successive rounds.

Let $seq_u(t_2) = \langle p_0, i_{(0)} \rangle, \ldots, \langle p_{n+1}, i_{(n+1)} \rangle$, where $y_u = n+2$, $p_0 = inf(P_u)$, and $p_{n+1} = sup(P_u)$. From line 34 of Figure 15-4, for any i, $1 < i < n$, the ith element in seq_u at round t_k, $k > 2$, is given by:

$$p_i(t_{k+1}) = p_i(t_k) + \rho_1 \left(\frac{p_{i-1}(t_k) + p_{i+1}(t_k)}{2} - p_i(t_k) \right).$$

For the endpoints, $p_i(t_{k+1}) = p_i(t_k)$. Let the ith uniformly spaced point on the curve Γ_u between the two endpoints be x_i. The parameter value $\bar{p}_i$ corresponding to x_i is given by $\bar{p}_i = p_0 + \frac{i}{n+1}(p_{n+1} - p_0)$. In what follows, we show that as $n \rightarrow \infty$, the p_i converge to $\bar{p}_i$ for every i, $0 < i < n+1$, that is, the location of the non-endpoint CN s are uniformly spaced on Γ_u. The rest of this proof is exactly the same as the proof of Theorem 3 in [46] in which the authors prove convergence of points on a straight line with uniform spacing.

Observe that $\bar{p}_i = \frac{1}{2}(\bar{p}_{i-1} + \bar{p}_{i+1}) = (1-\rho_1)\bar{p}_i + \frac{\rho_1}{2}(\bar{p}_{i-1} + \bar{p}_{i+1})$. Define error at step k, $k > 2$, as $e_i(k) = p_i(t_k) - \bar{p}_i$. Therefore, for each i, $2 \leq i \leq n-1$, $e_i(k+1) = p_i(t_{k+1}) - \bar{p}_i = (1-\rho_1)e_i(k) + \frac{\rho_1}{2}(e_{i-1}(k) + e_{i+1}(k))$, $e_1(k+1) = (1-\rho_1)e_1(k) + \frac{\rho_1}{2}e_2(k)$, and $e_n(k+1) = (1-\rho_1)e_n(k) + \frac{\rho_1}{2}e_{n-1}(k)$. The matrix for this can be written as: $e(k+1) = Te(k)$, where T is an $n \times n$ matrix:

$$\begin{bmatrix} 1-\rho_1 & \rho_1/2 & 0 & 0 & \ldots & 0 \\ \rho_1/2 & 1-\rho_1 & \rho_1/2 & 0 & \ldots & 0 \\ . & . & . & . & . & . \\ 0 & \ldots & 0 & \rho_1/2 & 1-\rho_1 & \rho_1/2 \\ 0 & \ldots & 0 & 0 & 1-\rho_1 & \rho_1/2 \end{bmatrix}.$$

Using symmetry of T, $\rho_1 \leq 1$, and some standard theorems from control theory, it follows that the largest eigenvalue of T is less than 1. This implies $lim_{k \rightarrow \infty} T^k = 0$, which implies $lim_{k \rightarrow \infty} e(k) = 0$. ∎

We conclude by summarizing the results in this section, Section 15.3:

Theorem 15.10 *If there are no* fail *or* restart *actions for robots at or after some round* t_0 *and the execution fragments of* VLayer'[MC] *satisfy the properties of set* S *from Definition 11.12, then:*

1. *Within a finite number of rounds after* t_0*, the set of* CN*s assigned to each* VN_u*,* $u \in U$*, becomes fixed, and the size of the set is proportional to the quantized length* q_u*, within a constant additive term* $\frac{10(2m-1)}{q_{min}\rho_2}$*.*

2. *All client nodes in a region* $u \in U$ *for which* $q_u \neq 0$ *are located on* Γ_u *and uniformly spaced on* Γ_u *in the limit.*

15.4 Self-stabilization of the Algorithm

In this section we show that the VSA-based motion coordination scheme is self-stabilizing. Specifically, we show that when the VSA and client components in the VSA layer start out in some *arbitrary state* owing to failures and restarts, they eventually produce traces that look like reachable traces of the motion coordination algorithm. Thus, the traces of $VLayer'[MC]$ running with some reachable state of $Vbcast\|RW'\|VW$, eventually, becomes indistinguishable from a reachable trace of $VLayer'[MC]$. Note that the virtual layer algorithm alg is instantiated here with the motion coordination algorithm MC of Section 15.2.

To show correctness, we use the strategy described in Section 9.3, where we describe a legal set L_{MC} of *VLayer'[MC]*, and show that it is a legal set (Section 15.4.1), and then legal states of the specification (here they are the reachable states). We then define a simulation relation $\mathcal{R}_{MC}$ between states of *VLayer'[MC]* (see Definition 15.13), and show the relation is a simulation relation (Lemma 15.14). We then show that for each state in L_{MC}, there exists a state in the invariant set $reachable_{VLayer'[MC]}$ such that $\mathcal{R}_{MC}$ holds between the states (Lemma 15.15). (This is to conclude that the system started in the set of legal states implements the system started in a reachable state.) We then show that *VLNodes[MC]* is self-stabilizing to L_{MC} relative to $R(RW'\|\|VW\|Vbcast)$ (Theorem 15.20). We conclude that the set of traces of the implementation stabilizes to the set of

reachable traces of executions of *VLayer'[MC]*.

We then go a step further, and connect the result to an emulation of the VSA layer. In Chapter 11 we showed how to implement a self-stabilizing VSA Layer. In particular, that implementation guarantees that for each algorithm $alg \in VAlgs$, the implementation stabilizes in some t_{stab} time to execution fragments whose traces are the same as those of execution fragments of the virtual layer that also happen to be in the set S described in Definition 11.12. Thus, if the coordination algorithm MC is such that $VLNodes[MC]$ self-stabilizes in some time t to L_{MC} relative to $R(RW' \| VW \| Vbcast)$, then we can conclude that physical node traces of the emulation algorithm on MC stabilize in time $t_{stab} + t$ to client traces of executions of the VSA layer started in legal set L_{MC} and that satisfy the properties of S (Theorem 15.22).

15.4.1 Legal Sets

First we describe two legal sets for $VLayer'[MC]$, L^1_{MC} and L_{MC} where L_{MC} is a subset of L^1_{MC}. Recall from Lemma 3.13 that a legal set of states for a TIOA is one where each closed execution fragment starting in a state in the set ends in a state in the set. We break the definition of the legal set up into two sets in order to simplify the proof reasoning and more easily prove stabilization later.

Legal set L^1_{MC}

The first legal set L^1_{MC} describes a set of states that result after the first GPSupdate occurs at each client node and the first timer occurs at each virtual node.

Definition 15.11 *A state* $\mathbf{x}$ *of* $VLayer'[MC]$ *is in* L^1_{MC} *iff the following hold:*

1. $\mathbf{x}\lceil X_{Vbcast \| RW' \| VW} \in reachable_{Vbcast \| RW' \| VW}$.

2. $\forall u \in U : \neg failed_u : clock_u \in \{RW'.now, \bot\} \wedge (M_u \neq \emptyset \Rightarrow clock_u \mod \delta \in (0, e + 2d + 2\epsilon])$.

3. $\forall p \in P : \neg failed_p \Rightarrow \mathbf{v}_p \in \{RW'.vel(p)/v_{max}, \bot\}$.

4. $\forall p \in P : \neg failed_p \wedge x_p \neq \bot$:

   (a) $x_p = RW'.loc(p) \wedge clock_p = RW'.now$.

   (b) $x_p^* \in \{x_p, \bot\} \vee ||x_p^* - x_p|| < v_{max}(\delta\lceil clock_p/\delta\rceil - clock_p - d_r)$.

   (c) $Vbcast.reg(p) = region(x_p) \vee clock \mod \delta \in (e + 2d + 2\epsilon, \delta - d_r + \epsilon_{sample})$.

Part (1) requires that $\mathbf{x}$ restricted to the state of $Vbcast\|RW'\|VW$ be a reachable state of $Vbcast\|RW'\|VW$. Part (2) states that nonfailed VSAs have $clock$s that are either equal to real-time or $\bot$, and have nonempty M only after the beginning of a round and up to $e+2d+2\epsilon$ time into a round. Part (3) states that nonfailed clients have velocity vectors that are equal either to $\bot$ or equal to the client's velocity vector in RW', scaled down by v_{max} (this scaling to a unit velocity vector is done for convenience; the domain of the client's local velocity variable is simply a direction, not a magnitude, which constrains the possible values of the variable and hence marginally simplifies stabilization reasoning). Finally, Part (4) states that nonfailed clients with non-$\bot$ positions have: (4a) positions equal to their actual location and local $clock$s equal to the real-time, (4b) targets that are one of $\bot$, the location, or a point reachable from the current location within d_r before the end of the round, and (4c) $Vbcast$ last region updates that match the current region or the time is within a certain time window in a round. It is routine to check that L^1_{MC} is indeed a legal set for *VLayer'[MC]*.

Legal set L_{MC}

Now we describe the main legal set L_{MC} for our algorithm. First we describe a set of *reset* states, states corresponding to states of $VLayer'[MC]$ at the start of a round. It turns out that it is relatively simple to show that an execution fragment of *VLayer'[MC]* reaches a reset state. We define L_{MC} to be the set of states reachable from these reset states. Due to our use of reset states, it is simple to show that our algorithm stabilizes to L_{MC}.

Definition 15.12 *A state* $\mathbf{x}$ *of* $VLayer'[MC]$ *is in* $Reset_{MC}$ *iff:*

1. $\mathbf{x} \in L^1_{MC}$.

2. $\forall p \in P : \neg failed_p \Rightarrow$
   $[to_send^-_p = to_send^+_p = \lambda \wedge (x_p = \bot \vee \ (x^*_p \neq \bot \wedge v_p = 0))]$.

3. $\forall u \in U : \neg failed_u \Rightarrow to_send_u = \lambda$.

4. $\forall \langle m, u, t, P' \rangle \in vbcastq : P' = \emptyset$.

5. $RW'.now \mod \delta = 0 \wedge \forall p \in P : \forall \langle l, t \rangle \in RW'.updates(p) : t < RW'.now$.

L_{MC} is the set of reachable states of $Start(VLayer'[MC], Reset_{MC})$.

$Reset_{MC}$ consists of states in which (1) the state is in L^1_{MC}, (2) each nonfailed client has an empty queue in its $VBDelay$ and either has a position variable equal to $\bot$ or has both a non-$\bot$ target and 0 velocity, (3) each nonfailed VSA has an empty queue in its $VBDelay$, (4) all messages in $Vbcast$ have either been delivered or dropped at each process, and (5) the time is the starting time for a round and no GPSupdates have yet occurred at this time. Once again, it is routine to check that that L_{MC} is a legal set for *VLayer'[MC]*.

15.4.2 Relationship between L_{MC} and reachable states

Now we define a simulation relation $\mathcal{R_{MC}}$ on the states of $VLayer'[MC]$, and then prove that for each state $\mathbf{x} \in L_{MC}$, there exists a state $\mathbf{y} \in reachable_{VLayer'[MC]}$ such that $\mathbf{x}$ and $\mathbf{y}$ are related by $\mathcal{R_{MC}}$. This implies that the trace of any execution fragment starting with $\mathbf{x}$ is the trace of an execution fragment starting with $\mathbf{y}$, which is a reachable trace of $VLayer'[MC]$. We define the candidate relation $\mathcal{R_{MC}}$ and prove that it is indeed a simulation relation.

Definition 15.13 *$\mathcal{R}_{MC}$ is a relation between states of $VLayer'[MC]$ such for any states $\mathbf{x}$ and $\mathbf{y}$ of $VLayer'[MC]$, $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$ iff the following conditions are satisfied:*

1. $\mathbf{x}(RW'.now) = \mathbf{y}(RW'.now) \wedge \mathbf{x}(RW'.loc) = \mathbf{y}(RW'.loc)$.

2. *For all* $p \in P$, $\mathbf{y}(vel(p)) \in \{\mathbf{x}(vel(p)), \bot\} \wedge$
   $\{t \in \mathbb{R}^{\geq 0} \mid \exists l \in R : \langle l, t \rangle \in \mathbf{x}(RW'.updates(p))\}$
   $= \{t \in \mathbb{R}^{\geq 0} \mid \exists l \in R : \langle l, t \rangle \in \mathbf{y}(RW'.updates(p))\}$.

3. $\mathbf{x}(VW) = \mathbf{y}(VW) \wedge \mathbf{x}(Vbcast.now) = \mathbf{y}(Vbcast.now)$.

4. $\mathbf{x}(Vbcast.reg) = \mathbf{y}(Vbcast.reg) \wedge$
   $\{\langle m, u, t, P' \rangle \in \mathbf{x}(Vbcast.vbcastq) \mid P' \neq \emptyset\}$
   $= \{\langle m, u, t, P' \rangle \in \mathbf{y}(Vbcast.vbcastq) \mid P' \neq \emptyset\}$.

5. *For all* $i \in P \cup U$, $\mathbf{x}(failed_i) = \mathbf{y}(failed_i)$.

6. *For all* $u \in U : \neg\mathbf{x}(failed_u)$*:*

   (a) $\mathbf{x}(clock_u) = \mathbf{y}(clock_u) \wedge \mathbf{x}(M_u) = \mathbf{y}(M_u)$
   $\wedge [\mathbf{x}(M_u) \neq \emptyset \Rightarrow \forall v \in nbrs^+(u) : \mathbf{x}(V_u(v)) = \mathbf{y}(V_u(v))]$.

   (b) $|\mathbf{x}(to_send_u)| = |\mathbf{y}(to_send_u)| \wedge \forall i \in [1, |\mathbf{x}(to_send_u)|] : \forall \langle m, t \rangle = \mathbf{x}(to_send_u[i]) : \mathbf{y}(to_send_u[i]) = \langle m, t + \mathbf{y}(rtimer_u) - \mathbf{x}(rtimer_u) \rangle$.

7. *For all* $p \in P : \neg\mathbf{x}(failed_p)$*:*

   (a) $\mathbf{x}(CN_p) = \mathbf{y}(CN_p) \vee [\mathbf{x}(\mathbf{x}_p) = \mathbf{y}(\mathbf{x}_p) = \bot \wedge \mathbf{x}(v_p) = \mathbf{y}(v_p)]$.

   (b) $\mathbf{x}(VBDelay_p) = \mathbf{y}(VBDelay_p)$.

   (c) $\mathbf{x}(to_send_p^-) \neq \lambda \Rightarrow \mathbf{x}(Vbcast.oldreg(p)) = \mathbf{y}(Vbcast.oldreg(p))$.

We describe the various conditions two related states $\mathbf{x}$ and $\mathbf{y}$ must satisfy. Part (1) requires that they share the same real-time and locations for CNs. Part (2) requires that for each client, the velocity at RW' is equal or the velocity in $\mathbf{y}$ is $\bot$, and GPSupdate records in the two states are for the same times. Part (3) requires that VW's state and $Vbcast.now$ are the same in $\mathbf{x}$ and $\mathbf{y}$. Part (4) requires that the unprocessed message tuples are the same and that the last recorded regions in $Vbcast$ for clients are the same in both states. Part (5) says that failure status of each CN and VN is the same in both states. Part (6a) requires that for a nonfailed VSA, local time and the set M are equal in $\mathbf{x}$ and $\mathbf{y}$, and further, if M is nonempty then V is equal for local regions in both states. Part (6b) says that the to_send queues for a nonfailed VSA are the same, except with the timestamps for messages in $\mathbf{y}$ adjusted up by the difference between $rtimer_u$ in state $\mathbf{y}$ and $\mathbf{x}$. Part (7a) requires that the algorithm state of a nonfailed CN is either the same, or both states share the same local v

and have locations equal to $\perp$. Part (7b) says that the $VBDelay$ state is the same for each nonfailed CN in $\mathbf{x}$ and $\mathbf{y}$. Finally, Part (7b) requires that if the $to_send_p^-$ buffer is nonempty in state $\mathbf{x}$ for a nonfailed client, then $Vbcast.oldreg(p)$ is the same in both states.

The proof of the following lemma is also routine and it breaks down into a large case analysis. Say that $\mathbf{x}$ and $\mathbf{y}$ are states in $Q_{VLayer'[MC]}$ such that $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$. For any action or closed trajectory σ of $VLayer'[MC]$, suppose $\mathbf{x}'$ is the state reached from $\mathbf{x}$, then, we have to show there exists a closed execution fragment β of $VLayer'[MC]$ with $\beta.fstate = \mathbf{y}$, $trace(\beta) = trace(\sigma)$, and $\mathbf{x}'\mathcal{R}_{MC}\beta.lstate$.

Lemma 15.14 *$\mathcal{R}_{\mathcal{MC}}$ is a simulation relation for $VLayer'[MC]$.*

Proof: It suffices to show that for every state $x \in VLayer'[MC]$, the following three conditions hold:

1. If $x \in \Theta_{VLayer'[MC]}$ then there exists a state $y \in \Theta_{VLayer'[MC]}$ such that $x\mathcal{R}_{MC}y$. It is obvious that taking $y = x$ satisfies this condition.

2. Say that x and y are states in $Q_{VLayer[MC]}$ such that $x\mathcal{R}_{MC}y$. Then for any action $a \in A_{VLayer'[MC]}$, if $VLayer'[MC]$ performs action a and the state changes from x to x', we must show there exists a closed execution fragment β of $VLayer'[MC]$ with $\beta.fstate = y, trace(\beta) = trace(\wp(x)a\wp(x'))$, and $x'\mathcal{R}_{MC}\beta.lstate$. For this proof we must consider each action. For each action, we can show the closed execution fragment β is simply $\wp(y)a\wp(y')$. This obviously satisfies the trace requirement. It is also easy to verify that $x'\mathcal{R}_{MC}y'$. This is because the relation $\mathcal{R}_{MC}$ holds between states that are effectively the same (any differences in state variables occur in circumstances where the differences are irrelevant). We do not perform the case analysis here since it is trivial.

3. Say that $\{x, y\} \subseteq Q_{VLayer'[MC]}$ and $x\mathcal{R}_{MC}y$. Let α be an execution fragment of $VLayer[MC]$ consisting of one closed trajectory, with $\alpha.fstate = x$. We must show that there is a closed execution fragment β of $VLayer'[MC]$ with $\beta.fstate = y, trace(\beta) = trace(\alpha)$, and $\alpha.lstate\mathcal{R}_{MC}\beta.lstate$. This is trivial in that we just take β to be the actionless fragment where client locations, clocks, and timers develop in a

similar manner to their counterparts in α. The only interesting thing to check is that if for some $p \in P$, $x(vel(p)) \neq \bot$ and $y(vel(p)) = \bot$, then any change in location for p in α is permissible in β. This holds because any change in location that is bounded by speed v_{max} is permissible when $vel(p)$ is set to $\bot$.

■

To show that each state in L_{MC} is related to a reachable state of $VLayer'[MC]$, it is enough to show that each state in $Reset_{MC}$ is related to a reachable state of $VLayer'[MC]$. The proof proceeds by providing a construction of an execution of $VLayer'[MC]$ for each state in $Reset_{MC}$.

Lemma 15.15 *For each state* $\mathbf{x} \in Reset_{MC}$*, there exists a state* $\mathbf{y} \in reachable_{VLayer'[MC]}$ *such that* $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$.

Proof: Let $\mathbf{x}$ be a state in $Reset_{MC}$. We construct an execution α based on state $\mathbf{x}$ such that $\mathbf{x}\mathcal{R}_{MC}\alpha.lstate$. The construction of α is in three phases. Each phase is constructed by modifying the execution constructed in the prior phase to produce a new valid execution of $VLayer'[MC]$. After Phase 1, the final state of the constructed execution shares client locations and real-time values with state $\mathbf{x}$. Phase 2 adds client restarts and velocity actions for nonfailed clients in state $\mathbf{x}$, making the final state of clients consistent with state $\mathbf{x}$. Phase 3 adds VSA restart actions to make the final state of VSAs consistent with state $\mathbf{x}$.

1. Let α_1 be an execution of $VLayer'[MC]$ where each client and VSA starts out failed, no restart or fail events occur, and $\alpha_1.ltime = \mathbf{x}(RW.now)$. For each failed $p \in P$, there exists some history of movement that never violates a maximum speed of v_{max}, is consistent with stored updates for p, and that lead to the current location of p. We move each failed p in just such a way and add a GPSupdate$(\langle l, t\rangle)_p$ at time t for each $\langle l, t\rangle \in \mathbf{x}(RW'.updates(p))$.

   For each nonfailed $p \in P$ and each state in α_1, we set $RW'.loc(p) = \mathbf{x}(RW'.loc(p))$ (meaning the client does not move). For each nonfailed $p \in$

P, add a GPSupdate$(\mathbf{x}(RW'.loc(p)), t)_p$ action for each t such that $\exists\langle l, t\rangle \in \mathbf{x}(RW'.updates(p))$.

For each $u \in U$, if $\mathbf{x}(last(u)) \neq \bot$ then add a time$(t)_u$ output at time t in α_1 for each t in the set $\{t^* \mid t^* = \mathbf{x}(last(u)) \vee (t^* < \mathbf{x}(last(u)) \wedge t^* \mod \epsilon_{sample} = 0)\}$.

Validity: It is obvious that the resulting execution is a valid execution of $VLayer'[MC]$.

Relation between $\mathbf{x}$ *and* $\alpha_1.lstate$*:* They satisfy (1)-(4) of Definition 15.13.

2. In order to construct α_2, we modify α_1 in the following way for each $p \in P$ such that $\neg\mathbf{x}(failed_p)$: If $\mathbf{x}(x_p) \neq \bot$, we add a restart$_p$ event immediately before and a velocity$(0)_p$ immediately after the last GPSupdate$_p$ event in α_1. If $\mathbf{x}(x_p) = \bot$ and $\mathbf{x}(v_p) = 0$, then we add a restart$_p$ and velocity$(0)_p$ event immediately after the last GPSupdate$_p$ event in α_1. If $\mathbf{x}(x_p) = \bot$ and $\mathbf{x}(v_p) = \bot$, then we add a restart$_p$ event at time $\mathbf{x}(RW'.now)$ in α_1.

   Validity Since restart actions are inputs they are always enabled, and a velocity$_p$ action is always enabled at client CN_p. Also, there can be no trajectory violations since any alive clients receive their first GPSupdate within ϵ_{sample} time of $\mathbf{x}(RW'.now)$ in α_2, meaning that since δ is larger than ϵ_{sample} and $\mathbf{x}(RW'.now)$ is a round boundary, there is no time before $\mathbf{x}(RW'.now)$ in α_2 where a cn-update should have been sent. It is obvious that this is a valid execution of *VLayer'[MC]*.

   Relation betweem $\mathbf{x}$ *and* $\alpha_2.lstate$ They satisfy (1)-(4) and (7) of Definition 15.13.

3. To construct α, we modify α_2 in the following way for each $u \in U$ such that $\neg\mathbf{x}(failed_u)$: If $\mathbf{x}(clock_u) = \bot$, we add a restart$_u$ event after any time$_u$ actions. If $\mathbf{x}(clock_u) \neq \bot$, we add a restart$_u$ event immediately before the last time$_u$ action.

   Validity A restart action is always enabled. Also, there can be no trajectory violations since no outputs at a VSA are enabled until its local M is nonempty. Since M is empty, we can conclude that this is a valid execution of *VLayer'[MC]*.

   Relation between $\mathbf{x}$ *and* $\alpha.lstate$ $\mathbf{x}\mathcal{R}_{MC}\alpha.lstate$.

We conclude that α is an execution of $VLayer'[MC]$ such that if we take $\mathbf{y} = \alpha.lstate$, then $\mathbf{y} \in reachable_{VLayer[MC]}$ and $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$. ■

It directly follows that for every state in L_{MC} there is a reachable state of $VLayer'[MC]$ that is related to it. (This result can be seen by noting that each state in L_{MC} is reachable from a state in $Reset_{MC}$, which the prior lemma implies is related to some state in $reachable_{VLayer'[MC]}$.)

Lemma 15.16 *For each state* $\mathbf{x} \in L_{MC}$*, there exists a state* $\mathbf{y} \in reachable_{VLayer'[MC]}$ *such that* $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$.

From Lemmas 15.16 and 15.14 it follows that the set of trace fragments of $VLayer'[MC]$ corresponding to execution fragments starting from L_{MC} is contained in the set of traces of $R(VLayer'[MC])$.

As a corollary to this result, we have the following simple observation, based on the matching execution constructed in the proof of the simulation relation above. It says that for any execution fragment α of $VLayer'[MC]$ in $S[VLNodes[MC]]$ and starting in a state x in L_{MC}, and given a state y related to x, there is an execution fragment starting with y that has the same trace as α and is also in $S[VLNodes[MC]]$. (This is very useful in Theorem 15.22, where we show that our emulation of a VSA layer can run the MC algorithm and eventually produce reachable traces of execution fragments satisfying certain failure patterns of VSAs.)

Corollary 15.17 *Let* α *be an execution fragment of* $VLayer'[MC]$ *where* $\alpha.fstate \in L_{MC}$ *and* α *is in* $S[VLNodes[MC]]$*. Let* y *be a state in* $reachable_{VLayer'[MC]}$ *such that* $\alpha.fstate\mathcal{R}_{MC}y$*. Then there exists an execution fragment* α' *of* $VLayer'[MC]$ *where:*

1. $\alpha'.fstate = y$.

2. $trace(\alpha) = trace(\alpha')$.

3. *If* α *is a closed execution fragment, then* $\alpha.lstate\mathcal{R}_{MC}\alpha'.lstate$.

4. $\alpha' \in S[VLNodes[MC]]$.

The first three properties of the corollary follow from the fact that $\mathcal{R}_{MC}$ is a simulation relation. The fourth follows from the proof that $\mathcal{R}_{MC}$ is a simulation relation; the constructed execution in the proof shows exactly the same mobile node movements and process failures and restarts. Hence, if α satisfies the properties of Definition 11.12, then α' must as well.

15.4.3 Stabilization to L_{MC}

We've seen that L_{MC} (Section 15.4.1) is a legal set for $VLayer'[MC]$, and that each state in L_{MC} is related to some reachable state of the system (Lemma 15.16). Now we can show that our algorithm stabilizes to the legal set (Theorem 15.20). We do this in two phases, corresponding to each legal set.

After we show that *VLNodes[MC]* self-stabilizes to L_{MC} relative to $R(RW'\|VW\|Vbcast)$, we use the fact that $\mathcal{R}_{MC}$ (see Definition 15.13) is a simulation relation that relates states in L_{MC} with reachable states of $VLayer'[MC]$ to conclude that a stabilizing VSA emulation algorithm emulating MC will eventually produce reachable traces of the system (Theorem 15.22).

First, we state the following the stabilization result. To see this, consider the moment after each client has received a GPSupdate and each virtual node has received a time, which takes at most ϵ_{sample} time.

Lemma 15.18 *$VLNodes[MC]$ is self-stabilizing to L^1_{MC} in time t for any $t > \epsilon_{sample}$ relative to the automaton $R(Vbcast\|RW'\|VW)$.*

Next we show that starting from a state in L^1_{MC}, we eventually arrive at a state in $Reset_{MC}$, and hence, a state in L_{MC}.

Lemma 15.19 *Executions of $VLayer'[MC]$ started in states in L^1_{MC} stabilize in time $\delta + d + e$ to executions started in states in L_{MC}.*

Proof: It suffices to show that for any length-$\delta+d+e$ prefix α of an execution fragment of $VLayer'[MC]$ starting from L^1_{MC}, $\alpha.lstate \in L_{MC}$. By the definition of L_{MC}, it suffices to show that there is at least one state in $Reset_{MC}$ that occurs in α.

Let t_0 be equal to $\alpha.fstate(RW'.now)$, the time of the first state in α. We consider all the "bad" messages that are about to be delivered after $\alpha.fstate$. (1) There may be messages in $Vbcast.vbcastq$ that can take up to d time to be dropped or delivered at each process. (2) There may be messages in to_send^- or to_send^+ queues at clients that can submitted to $Vbcast$ and take up to d time to be dropped or delivered at each process. And (3), there may be messages in to_send queues at VSAs that can take up to e time to be submitted to $Vbcast$ and an additional d time to be dropped or delivered at each process. We know that all "bad" messages will be processed (dropped or delivered at each process) by some state $\mathbf{x}$ in α such that $x(RW'.now) = t_1 = t_0 + d + e$.

Consider the state $\mathbf{x}^*$ at the start of the first round after state $\mathbf{x}$. Since $\mathbf{x}^*(RW'.now) = \delta(\lfloor t_1/\delta \rfloor + 1)$, we have that $\mathbf{x}^*(RW'.now) - t_0 = \mathbf{x}^*(RW'.now) - t_1 + e + d \leq \delta + e + d$. The only thing remaining to show is that $\mathbf{x}^*$ is in $Reset_{MC}$. It's obvious that $\mathbf{x}^*$ satisfies (1) and (5) of Definition 15.12. Code inspection tells us that for any state in L^1_{MC}, and hence, for any state in α, any new vcast transmissions of messages will fall into one of three categories:

1. Transmission of cn-update by a client at a time t such that $t \mod \delta = 0$. Such a message is delivered by time $t + d$.

2. Transmission of vn-update by a virtual node at a time t such that $t \mod \delta = d + \epsilon$. Such a message is delivered by time $t + d + e$.

3. Transmission of target-update by a virtual node at a time t such that $t \mod \delta = 2d + e + 2\epsilon$. Such a message is delivered by time $t + d + e$.

In each of these cases, any vcast transmission is processed before the start of the next round. Thus, $\mathbf{x}^*$ satisfies properties (2), (3), and (4) of Definition 15.12. To check (2), we just need to verify that for all nonfailed clients if x_p is not $\perp$ then x^*_p is not $\perp$ and v_p is 0. It suffices to show that at least one GPSupdate occurs at each client between state $\mathbf{x}$ and state $\mathbf{x}^*$. (Such an update at a nonfailed client would update x^*_p to be x_p for clients with $x^*_p = \perp$ or x^*_p too far away from x_p to arrive at x^*_p before $\mathbf{x}^*$. Any subsequent receipts of target-update messages will only result in an update to x^*_p if the client will be able to

arrive at x_p^* before $\mathbf{x}^*$. This implies that v_p can only be $\perp$ or 0, and since no GPSupdates could have occurred at the same time as $\mathbf{x}^*$, stopping conditions ensure that $v_p \neq \perp$.)

To see that at least one GPSupdate occurs at each client between state $\mathbf{x}'$ and state $\mathbf{x}^*$, we need that $\mathbf{x}^*(RW'.now) - \mathbf{x}'(RW'.now) > \epsilon_{sample}$. Since $\mathbf{x}^*(RW'.now) - \mathbf{x}'(RW'.now) = \delta - (\mathbf{x}'(RW'.now) \mod \delta) \geq \delta - e - 2d - 2\epsilon$, $\delta > e + 2d + 2\epsilon + d_r$, and $d_r > \epsilon_{sample}$ it follows that $\delta > e + 2d + 2\epsilon + \epsilon_{sample}$. ■

Combining our stabilization results we conclude that $VLNodes[MC]$ started in an arbitrary state and run with $R(Vbcast \| RW' \| VW)$ stabilizes to L_{MC} in time t_{mcstab}, where t_{mcstab} is any t such that $t > \delta + d + e + \epsilon_{sample}$. From transitivity of stabilization and 15.19, the next result follows.

Theorem 15.20 *Let t_{mcstab} be any t such that $t > \delta + d + e + \epsilon_{sample}$. $VLNodes[MC]$ is self-stabilizing to L_{MC} in time t_{mcstab} relative to $R(Vbcast \| RW' \| VW)$.*

Thus, despite starting from an arbitrary configuration of the VSA and client components in the VSA layer, within t_{mcstab} time, the system reaches a state in L_{MC}.

We can take this a step further to reason about the behavior of the system from the physical level implementation of the virtual layer:

Lemma 15.21 *Consider the S-constrained t_{stab}-stabilizing VSA emulation algorithm defined in Lemma 11.22. Then $traces_{\mathsf{ActHide}(H_{PL}, U(PLNodes[amap[MC]]) \| R(RW' \| Pbcast))}$ stabilizes in time $t_{stab} + t_{mcstab}$ to $\{trace(\alpha) \mid \alpha \in execs_{\mathsf{ActHide}(H_{VL}, Start(VLayer'[MC], L_{MC}))} \cap S(VLNodes[MC])\}$.*

The result is just an application of Corollary 8.4 to the emulation algorithm $amap$ of Lemma 11.22 and Theorem 15.20.

We then combine this result with Corollary 15.17 and Lemma 15.15 to arrive at the following result, which says that our stabilizing emulation algorithm from Section 11 running the MC algorithm produces traces that stabilize in time $t_{stab} + t_{mcstab}$ to traces of reachable execution fragments of the MC algorithm that also happen to satisfy the VSA failure patterns described in Definition 11.12:

Theorem 15.22 *Consider the S-constrained t_{stab}-stabilizing VSA emulation algorithm defined in Lemma 11.22. Then $traces_{\mathsf{ActHide}(H_{PL}, U(PLNodes[amap[MC]]) \| R(RW' \| Pbcast))}$ stabilizes in time $t_{stab} + t_{mcstab}$ to $\{trace(\alpha) \mid \alpha \in execs_{\mathsf{ActHide}(H_{VL}, R(VLayer'[MC]))} \cap S(VLNodes[MC])\}$.*

Thus, putting together this result and Theorem 15.10, we can make the following statement about the locations of physical nodes that run our VSA emulation of the MC algorithm starting in some arbitrary state:

Theorem 15.23 *Let α be any execution of the S-constrained t_{stab}-stabilizing VSA emulation algorithm defined in Lemma 11.22, running MC and starting from an arbitrary configuration of the physical nodes. Assume that there is some time t after which there are no failures or restarts of the physical nodes.*
Then: (1) within a finite amount of time after t, the set of physical nodes assigned to each region becomes fixed and the size of the set is proportional to the quantized length q_u, within a constant additive term $\frac{10(2m-1)}{q_{min}\rho_2}$, and (2) and the physical nodes in regions u for which $q_u \neq 0$ are located on Γ_u and uniformly spaced in the limit.

15.5 Conclusion

We have described how we can use the Virtual Stationary Automaton infrastructure to design protocols that are resilient to failure of participating agents. In particular, we presented a protocol by which the participating robots can be uniformly spaced on an arbitrary curve. The VSA layer implementation and the coordination protocol are both self-stabilizing. Thus, each robot can begin in an arbitrary state, in an arbitrary location in the network, and the distribution of the robots will still converge to the specified curve. The proposed coordination protocol uses only local information, and hence, should adapt well to flocking or tracking problems where the target formation is dynamically changing.

Chapter 16

Conclusions

In this thesis we have introduced the idea of the *Virtual Stationary Automata* layer for simplifying implementations of applications for mobile wireless networks, a theory for self-stabilization in timed systems, and a theory for stabilizing emulations. We have provided a stabilizing emulation of the VSA layer and shown it to be a stabilizing emulation. We have demonstrated the use of the VSA layer to provide implementations of several services for mobile networks.

In this chapter, we begin by reviewing the main contributions of this thesis (Section 16.1). We then discuss some conclusions about our approach (Section 16.2) and some open questions and ongoing research (Section 16.3).

16.1 Contributions

The first main contribution of this thesis is the introduction of formal semantics for stabilization and crash/ restart failures in the TIOA model (Chapters 3 and 5). Self-stabilization [26, 27] is the ability to recover from an arbitrarily corrupt state. We define stabilization in the TIOA systems using hybrid sequences, and develop several techniques to use this theory throughout the thesis. Our definition of stabilization makes provisions for discussing external sources of stability and allows us to tackle stabilization of implementations of long-lived services with invocation / response or send / receive behavior, where it might not be possible to find a "reset" state. Our crash/ restart failure modeling is done with a

general transformation that takes a TIOA program and produces a new program that can suffer from crash failures and restarts.

The second main contribution of this thesis is the presentation of a formal semantics for emulation of a system (Chapter 4) and the application of this definition to an emulation of a virtual layer by a physical node layer (Chapter 8). This provides proof obligations required to conclude that one system successfully emulates another system. We describe an emulation as a kind of implementation relationship between two sets of timed machines, where an emulation of a program produces behavior that looks like that of the program being emulated. We also present a formal semantics for a stabilizing emulation of a system, where an emulation of a program can start in an arbitrary state but eventually behave as though it is the program started in an arbitrary state. We observe that if a stabilizing emulation of a stabilizing program is used, then the resulting system will eventually behave like the program started from some desirable state.

The third main contribution of this thesis is the introduction of the timed Virtual Stationary Automata programming layer (Chapter 7), which can help application developers write simpler algorithms for mobile networks. This is a *virtual* fixed infrastructure, consisting of *timing-aware* and *location-aware* VSAs at fixed locations which mobile nodes can interact with. Each VSA represents a predetermined geographic area and has broadcast capabilities similar to those of the mobile nodes, though perhaps suffering from an additional additive broadcast delay, allowing nearby VSAs and mobile nodes to communicate with one another.

Our fourth main contribution is a protocol for emulating the VSA layer using mobile nodes with access to a GPS oracle and a proof that the protocol is a stabilizing VSA layer emulation (Part II). We use a leader-based replicated state machine approach to implement each region's VSA with mobile nodes located in that region. The proof that this protocol is a stabilizing emulation of the VSA layer exercises the stabilizing emulation definitions, as well as the stabilization theory. A phase-based approach to proving stabilization is used to show that the protocol is stabilizing.

Our fifth main contribution is to use the VSA layer to provide stabilizing implementations of two main services: end-to-end routing (Chapter 14) and motion coordination

(Chapter 15). The end-to-end routing service is implemented in three stabilizing layers: geocast (Chapter 12), location management (Chapter 13), and the top-level implementation of the end-to-end routing service. The stabilization of the top-level end-to-end routing service is dependent on the stabilization of the location management service, which is in turn dependent on the stabilization of the geocast service; we develop proof techniques to show these stabilization results. The motion coordination algorithm is especially interesting in that it demonstrates the use of the VSA layer to actively direct movement of client nodes. Using a stabilizing emulation of the VSA layer such as the one from Part II, we can take a stabilizing VSA layer implementation of an application (such as the end-to-end routing application or the motion coordination application), run the stabilizing emulation algorithm on that VSA layer implementation, and conclude that the resulting system produces behaviors that eventually look like those of the application.

To summarize, this thesis develops theories of stabilization and crash/ restart failures for timed systems and a theory for emulation and stabilizing emulation; it introduces the idea of a VSA programming layer; it presents a stabilizing emulation of the VSA layer; and it presents stabilizing VSA layer implementations of an end-to-end routing service and a motion coordination service.

16.2 Evaluation

Here we discuss several issues related to the VSA layer and its implementation in this thesis.

The theories of stabilization and crash/ restart failures in Chapters 3 and 5 provide simple formal foundations for reasoning about failure-prone timed systems. There is still work to be done to further develop the stabilization theory to include other concepts, such as *snap stabilization* (instantaneous fault containment) [13], from the general stabilization literature.

Because this is a theoretical thesis where we demonstrate new theories of stabilization and emulation, we concentrate on only a virtual layer with very strong semantics, making it easy to use the layer to program applications. The communication between clients and

VSAs in neighboring regions is reliable and the clocks in the system do not drift. This is very useful in circumstances where safety-critical applications require timely and reliable coordination and communication, and where the devices ultimately emulating the layer have hardware that behaves well enough to have the implementation be successful.

However, such strong semantics are not necessary for many applications. For example, in the case of a shoe sale application where a VSA for a region relies on messages from mobile shoppers to compile a "hot list" of stores to visit, it might not be critical for each new sale message sent to the VSA be received or that each shopper in the region is guaranteed to get each notification from the VSA of a store they could shop at. Such a service really only needs to be best effort.

In addition, the hardware of the underlying mobile devices might be able to support implementing the VSA layer described. Without reliable communication on the part of mobile nodes within some distance of each other, we can't provide VSAs that have reliable communication. Also, if mobile nodes have clocks that drift, we can't provide VSAs with perfect clocks. In addition, if the RW service is inexact, we would need to take this into account in our algorithm.

Another perhaps-too-strong feature of the VSA layer is that there is a VSA at each region of a network, and that each VSA must be able to communicate with each neighboring VSA. In the real world, where wireless broadcast becomes less reliable as more congestion occurs, it is possible that having VSAs be so close to one another can result in many lost messages, leading to VSA failures. Also, it might be that not every region of a deployment space needs a VSA. If coordination only needs to be done locally and only at areas remote from one another, the VSA layer model described here might be overkill.

Even taking the strong semantics of the VSA layer as given, the implementation of that layer in this thesis is not optimized for any performance metric, such as the maximum delay of a VSA broadcast, message overhead of the emulation, stabilization time, VSA restart time, or the local computation complexity.

The implementations of the VSA layer applications in Chapters 12-15 were also not optimized for message complexity, time complexity, or fault-tolerance. The idea of using virtual nodes to help accomplish routing does seem to simplify the task of providing such

an application; however, the geocast application does not, for example, try to do anything in the way of routing around failed VSAs. The fault-tolerance and message complexity of the location management service could be improved by, for example, using ideas from [8] to limit information propagation through the occasional use of forwarding pointers.

We believe that the motion coordination application of Chapter 15 presents a very interesting paradigm for coordination. The implementation of the service introduces a framework for interaction between mobile nodes and virtual controllers that can be useful for other coordination applications. One example is air traffic control; in [11] VSA controllers for sectors of airspace were responsible for issuing flight vectors to aircraft while maintaining certain safety conditions. Another example is in [12], where a VSA is used to implement a virtual traffic light.

16.3 Open questions and avenues for research

Considering the fact that strong semantics for the VSA layer is not always necessary and that it is not always able to be provided, it would be interesting to consider what a weaker semantics for the VSA layer would look like. For example, what should the semantics be if probabilistic message loss is possible at the physical layer? What if the message delay at the physical layer comes from some distribution, rather than being nicely bounded by d_{phys}? What should the model for a VSA look like if the physical nodes only have access to clocks that suffer some bounded drift?

How do we handle message collision at the physical layer? There is recent work [47] that implies that collision might be something that can be worked around most of the time, implying that a stabilizing emulation of a VSA layer might very well not need much modification to work in this environment. There are also TDMA timeslot-based approaches that could help us prevent collisions to begin with; timeslots could be apportioned amongst regions such that neighboring regions are on different timeslots, minimizing the chances of collision. There is also work on handling collisions that is specifically geared towards other virtual node layers [44].

How do we handle the case where RW is only approximate or is a service that might

take some time to stabilize? In the second case, where it is a service that might take time to stabilize, the only impact on this work would be to extend the stabilization time of each of the algorithms by the amount of time it takes for RW to stabilize. In the first case, where RW is only approximate, if we have a bound on how inexact the location information from RW can be, we might be able to accomodate it with the algorithm presented here; we simply require that the broadcast range for nodes that "think" they are in some region is such that they can reach all nodes that "think" they are in that region or a neighboring one. However, there is a tradeoff that becomes apparent in this approach: since broadcast range is bounded, the additional fuzziness results in the shrinking of region sizes. In the real-world, this can result in increased message loss, due to additional congestion.

For each physical model, what are the best/ most efficient algorithms for implementing the VSA layer under various metrics for performance?

Since power consumption is also a common concern for mobile nodes in the real-world, it would also be interesting to consider implementations of VSAs that are power and trajectory sensitive, in that physical nodes with ample power resources that are likely to remain in a region for a longer period are more likely to take on the burden of virtual machine emulation.

Another thing to pursue is the question of how to split up the virtual machine emulation to lessen the burden of emulation. For example, if a database is being replicated, it might be possible for emulators to be responsible for something less than the full database. Such an approach can also help alleviate some privacy concerns, as no one emulator might have access to all potentially sensitive information in a region. What would be the semantics of a virtual layer implemented in this way?

An implementation of a version of the VSA layer with much simpler semantics was examined in [12]; it would be interesting to examine multiple implementation algorithms for different semantics of the VSA layer so as to both: (1) experiment with just how easy/ hard it is to implement efficient versions of some of these layers in the real world, and (2) study the difficulty of implementing different applications on these layers with different semantics. For the second point, it would also be useful to compare the complexity of algorithms implemented with various VSA layers to the complexity of algorithms for the

same services but that do not use a VSA layer; how much overhead is being introduced by use of the layer and how does it seem to trade off with the ease of implementing correct algorithms?

The VSA model makes the assumption of a globally known static carve-up of the deployment space into non-overlapping regions. We could consider an extension to the VSA model that allows regions to be overlapping or the region map to be dynamic. The model and emulation implementation can be relatively easily extended to allow overlapping regions; the only real change that should be needed is for emulators to run multiple copies of the programs described in part I of the thesis, one for each region the emulator is in. On the question of the static nature of the region map, while this makes the model predictable and easy to work with, it is possible that over time we might want to modify the regions of the network by splitting regions, merging them, or some combination of the two. This leads to the question of how such changes get communicated to emulators, and what circumstances should cause the change to occur. The RW automaton could perhaps be modified so that it reports a region map as well as a location. However, this would also introduce additional stabilization difficulties (both in emulation and in using the virtual layer), since the assumed global region map would no longer be something we could consider hard-wired, meaning it is soft state that would be susceptible to corruption failure or could be started in an arbitrary state.

Also of interest would be developing more applications for the VSA layer. The motion coordination algorithms seem particularly interesting; I mentioned the air traffic control [11] and traffic light [12] work, but there are a number of extensions and additional applications whose implementations could benefit from use of the VSA layer. For example, the virtual traffic light application is just one possible piece of a larger potential group of intelligent-highway applications, in which cars will carry on-board computers with wireless communication capabilities. Distributed algorithms running on these systems will need to conduct a variety of activies, including collecting data (e.g., about traffic patterns), alerting cars about road hazards (e.g., accidents or arriving emergency vehicles), and providing advice and control. For example, the distributed protocol may suggest less-congested alternative routes, or may even emulate the functions of virtual traffic lights at intersections

having no real traffic lights.

Other applications of interest could include things like virtual storage. Because VSAs are failure prone, the state of a VSA can be lost. A virtual storage application could provide a means by which to back-up the data at a VSA. This would not be an additional feature of the VSA layer, but instead an application implemented on top of the VSA layer. The Geoquorums work [34] describes such an application for a different virtual layer model.

On the theory side, as I mentioned, many concepts in stabilization could be formalized using the definition of stabilization for TIOA defined in this thesis. There are also other results that may be useful; one theory in particular that would be useful to provide is a theory of stabilizing composition [27] for TIOA that accommodates the $Fail$-transform described in this thesis. Roughly, we would like to have a result saying that for comparable TIOAs A and B and a TIOA C that is compatible with both, if the traces of $Fail(A)$ stabilize to the traces of $Fail(U(B))$, then the traces of $Fail(A \| C)$ stabilize to the traces of $Fail(U(B \| C))$. We would also like to have a generalization that allows us to consider multiple machines composed together within the $Fail$-transform ($A_1 \| A_2 \cdots A_n$, rather than A), or a generalization that allows us to consider the traces of $Fail(A) \| D$ stabilizing to traces of $Fail(U(B)) \| E$ and conclude that the traces of $Fail(A \| C) \| D$ stabilizes to the traces of $Fail(U(B \| C)) \| E$, etc. The proof of Lemma 11.23 would have been much simpler if such results existed.

Bibliography

[1] Abraham, I., Dolev, D., and Malkhi, D., "LLS: a locality aware location service for mobile ad hoc networks", *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing*, 2004.

[2] *ACM Transactions on Sensor Networks*.

[3] *Ad Hoc Networks Journal*, Elsevier.

[4] Akylidz, I.F., Su, W., Sankarasubramanian, Y., and Cayirci, E., "Wireless sensor networks: a survey", *Computer Networks* (Elsevier), 38(4), pp. 393–422, 2002.

[5] Ando, H., Oasa, Y., Suzuki, I., and Yamashita, M., "Distributed memoryless point convergence algorithm for mobile robots with limited visibility", *IEEE Transactions on Robotics and Automation*, 15(5):818–828, 1999.

[6] Arora, A., Demirbas, M., Lynch, N., and Nolte, T., "A Hierarchy-based Fault-local Stabilizing Algorithm for Tracking in Sensor Networks", *8th International Conference on Principles of Distributed Systems (OPODIS)*, 2004.

[7] Awerbuch, B. and Peleg, D., "Sparse partitions (extended abstract)", *IEEE Symposium on Foundations of Computer Science*, 1990.

[8] Awerbuch, B. and Peleg, D., "Online tracking of mobile users", *Journal of the Association for Computing Machinery*, 42, 1995.

[9] Beal, J., "Persistent nodes for reliable memory in geographically local networks", Tech Report AIM-2003-11, MIT, 2003.

[10] Blondel, V., Hendrickx, J., Olshevsky, A., and Tsitsiklis, J. 2005. Convergence in multiagent coordination consensus and flocking. In *Proceedings of the Joint forty-fourth IEEE Conference on Decision and Control and European Control Conference*. 2996–3000.

[11] Brown, M. D. 2007. Air traffic control using virtual stationary automata. M.S. thesis, Massachusetts Institute of Technology.

[12] Brown, M., Gilbert, S., Lynch, N., Newport, C., Nolte, T., and Spindel, M. The Virtual Node Layer: A Programming Abstraction for Wireless Sensor Networks. In *International Workshop on Wireless Sensor Network Architecture*, April 2007.

[13] Bui, A., Datta, A., Petit, F., and Villain, V. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, June 1999.

[14] Camp, T. and Liu, Y., "An adaptive mesh-based protocol for geocast routing", *Journal of Parallel and Distributed Computing: Special Issue on Mobile Ad-hoc Networking and Computing*, pp. 196–213, 2002.

[15] Chandy, K. M., Mitra, S., and Pilotto, C. 2008. Convergence verification: From shared memory to partially synchronous systems. In *In proceedings of Formal Modeling and Analysis of Timed Systems (FORMATS'08)*. LNCS, vol. 5215. Springer Verlag, 217–231.

[16] Chockler, G., Demirbas, M., Gilbert, S., Newport, C., and Nolte, T., "Consensus and Collision Detectors in Wireless Ad Hoc Networks", *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

[17] Chockler, G., Demirbas, M., Gilbert, S., Newport, C., and Nolte, T., "Consensus and Collision Detectors in Wireless Ad Hoc Networks", *Distributed Computing*, June, 2008.

[18] Chockler, G., Gilbert, S., and Lynch, N. 2008. Virtual infrastructure for collision-prone wireless networks. In *Proceedings of PODC*. To appear.

[19] Clavaski, S., Chaves, M., Day, R., Nag, P., Williams, A., and Zhang, W. 2003. Vehicle networks: achieving regular formation. In *Proceedings of the American control Conference*.

[20] Cooper, M., comment, `http://www.arraycomm.com/news/pr_detail.htm?id=104`, 1973.

[21] Cortes, J., Martinez, S., Karatas, T., and Bullo, F., "Coverage control for mobile sensing networks", *IEEE Transactions on Robotics and Automation*, 20(2):243–255, 2004.

[22] Défago, X. and Konagaya, A. 2002. Circle formation for oblivious anonymous mobile robots with no common sense of orientation. In *Proc. 2nd Int'l Workshop on Principles of Mobile Computing (POMC'02)*. ACM, Toulouse, France, 97–104.

[23] Défago, X. and Souissi, S. 2008. Non-uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. *Theor. Comput. Sci. 396,* 1-3, 97–112.

[24] Demers, A., Gehrke, J., Rajaraman, R., Trigoni, N., and Yao, Y., "Energy-Efficient Data Management for Sensor-Networks: A Work-In-Progress Report", *2nd IEEE Upstate New York Workshop on Sensor Networks*, `comlab.ecs.syr/edu/workshop`, 2003.

[25] Demirbas, M., Arora, A., and Gouda, M., "A pursuer-evader game for sensor networks", *Symposium on Self-Stabilizing Systems (SSS)*, 2003.

[26] Dijkstra, E.W., "Self stabilizing systems in spite of distributed control", *Communications of the ACM*, 1974.

[27] Dolev, S., *Self-Stabilization*, MIT Press, 2000.

[28] Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., and Nolte, T., "Brief announcement: Virtual stationary automata for mobile networks", *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

[29] Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., and Nolte, T., "Timed virtual stationary automata for mobile networks", *TR MIT-LCS-TR-979a*, 2005.

[30] Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., and Nolte, T. 2005a. Virtual stationary automata for mobile networks. In *Proceedings of OPODIS*.

[31] Dolev, S., Gilbert, S., Lynch, N., Schiller, E., Shvartsman, A., and Welch, J., "Virtual Mobile Nodes for Mobile Ad Hoc Networks", *International Conference on Principles of Distributed Computing (DISC)*, 2004.

[32] Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., Welch, J., "GeoQuorums: Implementing Atomic Memory in Ad Hoc Networks", *17th International Conference on Principles of Distributed Computing (DISC)*, Springer-Verlag LNCS:2848, pp. 306-320, 2003.

[33] Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., and Welch, J. 2003. Geoquorums: Implementing atomic memory in ad hoc networks. In *Distributed algorithms*, F. E. Fich, Ed. Lecture Notes in Computer Science, vol. 2848/2003. 306–320.

[34] Dolev, S., Gilbert, S., Lynch, N. A., Shvartsman, A. A., and Welch, J. 2005. Geoquorums: Implementing atomic memory in mobile ad hoc networks. *Distributed Computing*.

[35] Dolev, S., Herman, T., and Lahiani, L., "Polygonal Broadcast, Secret Maturity and the Firing Sensors", *Third International Conference on Fun with Algorithms (FUN)*, pp. 41-52, May 2004. Also to appear in *Ad Hoc Networks Journal*, Elseiver.

[36] Dolev, S., Israeli, A., and Moran, S., "Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity", *Proceeding of the ACM Symposium on the Principles of Distributed Computing (PODC 90)*, pp. 103-117. Also in *Distributed Computing* 7(1): 3-16 (1993).

[37] Dolev, S., Lahiani, L., Lynch, N., and Nolte, T., "Self-stabilizing Mobile Node Location Management and Message Routing", *7th Self-stabilizing Systems (SSS), 2005*.

[38] Dolev, S., Pradhan, D.K., and Welch, J.L., “Modified Tree Structure for Location Management in Mobile Environments”, *Computer Communications*, Special issue on mobile computing, Vol. 19, No. 4, pp. 335-345, April 1996. Also INFOCOM 1995, Vol. 2, pp. 530-537, 1995.

[39] Dolev, S. and Welch, J.L.,“Crash Resilient Communication in Dynamic Networks”, *IEEE Transactions on Computers*, Vol. 46, No. 1, pp.14-26, January 1997.

[40] EFRIMA, A. AND PELEG, D. 2007. Distributed models and algorithms for mobile robot systems. In *SOFSEM (1)*. Lecture Notes in Computer Science, vol. 4362. Springer, Harrachov, Czech Republic, 70–87.

[41] FAX, J. AND MURRAY, R. 2004. Information flow and cooperative control of vehicle formations. *IEEE Transactions on Automatic Control 49*, 1465–1476.

[42] FLOCCHINI, P., PRENCIPE, G., SANTORO, N., AND WIDMAYER, P. 2001. Pattern formation by autonomous robots without chirality. In *SIROCCO*. 147–162.

[43] Gazi, V., and Passino, K.M., “Stability analysis of swarms”, *IEEE Transactions on Automatic Control*, 48(4):692–697, 2003.

[44] Gilbert, S., ”Virtual Infrastructure for Wireless Ad Hoc Networks”, Thesis, MIT, 2007.

[45] Gilbert, S., Lynch, N., Mitra, S., and Nolte, T. ”Self-Stabilizing Mobile Robot Formations with Virtual Nodes”, *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, To appear: November 2008.

[46] Goldenberg, D.K., Lin, J., and Morse, A.S., “Towards mobility as a network control primitive”, *MobiHoc ’04: Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing*, pages 163–174. ACM Press, 2004.

[47] Gollakota, S. and Katabi, D., ”ZigZag Decoding: Combating Hidden Terminals in Wireless Networks”, *ACM SIGCOMM*, 2008.

[48] Haas, Z.J. and Liang, B., "Ad Hoc Mobility Management With Uniform Quorum Systems", *IEEE/ACM Trans. on Networking*, Vol. 7, No. 2, p. 228-240, April 1999.

[49] Herlihy, M.P. and Tirthapura, S., "Self-stabilizing distributed queueing", *Proceedings of 15th International Symposium on Distributed Computing*, pages 209–219, October 2001.

[50] HERMAN, T. 1996. Self-stabilization bibliography: Access guide. *Theoretical Computer Science*.

[51] Hubaux, J.P., Le Boudec, J.Y., Giordano, S., and Hamdi, M., "The Terminodes Project: Towards Mobile Ad-Hoc WAN", *Proceedings of MOMUC*, 1999.

[52] *IEEE Pervasive Computing: Mobile and Ubiquitous Systems*.

[53] *IEEE Transactions on Mobile Computing*.

[54] Imielinski, T. and Badrinath, B.R., "Mobile wireless computing: challenges in data management", *Communications of the ACM*, Vol. 37, Issue 10, pp. 18-28, October 1994.

[55] Jadbabaie, A., Lin, J., and Morse, A.S., "Coordination of groups of mobile autonomous agents using nearest neighbor rules", *IEEE Transactions on Automatic Control*, 48(6):988–1001, 2003.

[56] Johnson, D., Maltz, D., and Broch, J., "DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks", chapter 5, pp.139–172, Addison-Wesley, 2001.

[57] Karp, B. and Kung, H. T., "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks", *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, pp. 243-254, SCM Press, 2000.

[58] Kaynar, D., Lynch, N., Segala, R., and Vaandrager, F., *The Theory of Timed I/O Automata*. Morgan Claypool, 2006.

[59] Kuhn, F., Wattenhofer, R., Zhang, Y., and Zollinger, A., "Geometric Ad-Hoc Routing: Of Theory and Practice", *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.

[60] Kuhn, F., Wattenhofer, R., and Zollinger, A., "Asymptotically Optimal Geometric Mobile Ad-Hoc routing", *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (Dial-M)*, pp. 24-33, ACM Press, 2002.

[61] Lamport, L., "Time, clocks, and the ordering of events in a distributed system", *Communications of the ACM*, 1978.

[62] Li, J., Jannotti, J., De Couto, D.S.J., Karger, D.R., and Morris, R., "A Scalable Location Service for Geographic Ad Hoc Routing", *Proceedings of Mobicom*, 2000.

[63] Lin, J., Morse, A.S., and Anderson, B., "Multi-agent rendezvous problem", *42nd IEEE Conference on Decision and Control*, 2003.

[64] Lok, C., "Instant Networks: Just Add Software", *Technology Review*, June, 2005.

[65] Lynch, N., *Distributed Algorithms*, Morgan Kaufman, 1996.

[66] Lynch, N., Mitra, S., and Nolte, T., "Motion coordination using virtual nodes", *IEEE Conference on Decision and Control*, 2005.

[67] Lynch, N., Segala, R., and Vaandrager, F., "Hybrid I/O automata", *Information and Computation*, 185(1):105–157, August 2003.

[68] Malkhi, D., Reiter, M., and Wright, R., "Probabilistic Quorum Systems", *Proceeding of the 16th Annual ACM Symposium on the Principles of Distributed Computing (PODC 97)*, pp. 267-273, Santa Barbara, CA, August 1997.

[69] Martinez, S., Cortes, J., and Bullo, F., "On robust rendezvous for mobile autonomous agents", *IFAC World Congress*, Prague, Czech Republic, 2005.

[70] Merritt, M., Modugno, F., and Tuttle, M., "Time constrained automata", *2nd International Conference on Concurrency Theory (CONCUR)*, 1991.

[71] Mittal, V., Demirbas, M., and Arora, A., "LOCI: Local clustering in large scale wireless networks", *TR OSU-CISRC-2/03-TR07*, 2003.

[72] Nath, B. and Niculescu, D., "Routing on a curve", *ACM SIGCOMM Computer Communication Review*, 2003.

[73] Navas, J.C. and Imielinski, T., "Geocast- geographic addressing and routing", *Proceedings of the 3rd MobiCom*, 1997.

[74] Neogi, N., "Designing Trustworthy Networked Systems: A Case Study of the National Airspace System", International System Safety Conference, Ottawa, Canada, August 3-11, 2003.

[75] NOLTE, T. AND LYNCH, N. A. 2007a. Self-stabilization and virtual node layer emulations. In *Proceedings of SSS*. 394–408.

[76] NOLTE, T. AND LYNCH, N. A. 2007b. A virtual node-based tracking algorithm for mobile networks. In *ICDCS*.

[77] OLFATI-SABER, R., FAX, J., AND MURRAY, R. 2007. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE 95,* 1 (January), 215–233.

[78] Park, V. and Corson, M., A highly adaptive distributed routing algorithm for mobile wireless networks. *IEEE Infocom*, April 1997.

[79] Perkins, C. and Royer, E., Ad hoc on-demand distance vector routing. *2nd IEEE Workshop on Mobile Computing Systems and Applications*, February 1999.

[80] PRENCIPE, G. 2000. Achievable patterns by an even number of autonomous mobile robots. Tech. Rep. TR-00-11. 17.

[81] PRENCIPE, G. 2001. Corda: Distributed coordination of a set of autonomous mobile robots. In *ERSADS*. 185–190.

[82] Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., and Shenker, S., "GHT: A Geographic Hash Table for Data-Centric Storage", *First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[83] Suzuki, I. and Yamashita, M., "Distributed autonomous mobile robots: Formation of geometric patterns", *SIAM Journal of computing*, 28(4):1347–1363, 1999.

[84] Talbot, D., "Airborne Networks", *Technology Review*, May, 2005.

[85] Talbot, D., "The Ascent of the Robotic Attack Jet", *Technology Review*, March, 2005.

[86] TinyOS Community Forum, `http://www.tinyos.net`.

[87] Vasek, T., "World Changing Ideas: Germany", *Technology Review*, April, 2005.

[88] Weisman, R., "MIT seeks computing revolution", *Boston Globe*, 2005.