PSYNC: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms

Cezara Drăgoi
INRIA, ENS, CNRS, France
cezara.dragoi@inria.fr

Thomas A. Henzinger
IST Austria, Austria
tah@ist.ac.at

Damien Zufferey
MIT CSAIL, USA
zufferey@csail.mit.edu

Abstract

Fault-tolerant distributed algorithms play an important role in many critical/high-availability applications. These algorithms are notoriously difficult to implement correctly, due to asynchronous communication and the occurrence of faults, such as the network dropping messages or computers crashing.

We introduce PSYNC, a domain specific language based on the *Heard-Of* model, which views asynchronous faulty systems as synchronous ones with an adversarial environment that simulates asynchrony and faults by dropping messages. We define a runtime system for PSYNC that efficiently executes on asynchronous networks. We formalize the relation between the runtime system and PSYNC in terms of observational refinement. The high-level lockstep abstraction introduced by PSYNC simplifies the design and implementation of fault-tolerant distributed algorithms and enables automated formal verification.

We have implemented an embedding of PSYNC in the SCALA programming language with a runtime system for asynchronous networks. We show the applicability of PSYNC by implementing several important fault-tolerant distributed algorithms and we compare the implementation of consensus algorithms in PSYNC against implementations in other languages in terms of code size, runtime efficiency, and verification.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Verification

Keywords Fault-tolerant distributed algorithms, Round model, Partial synchrony, Automated verification, Consensus

1. Introduction

The need for highly available data storage systems and for higher processing power has led to the development of distributed systems. A distributed system is a set of independent nodes in a network, that communicate and synchronize via message passing, giving the illusion of acting as a single system. The difficulty in designing these systems comes from the network unreliability and the host failures: messages can be dropped and nodes can crash. The processes have only a limited view over the entire system and they must coordinate to achieve global goals.

A general mechanism for implementing a fault-tolerant service is replication: the application is copied on different replicas which are kept consistent. Clients send requests to a replica running the service, the service communicates with the other nodes to maintain a consistent state of the global system, and replies to the clients. Consistency is maintained by solving *consensus problems*. Each process has an initial value, and all processes have to agree on a unique decision value among the initial ones. Therefore, all replicas return the same value when queried for the attribute of an object. Consensus algorithms have received a lot of attention in academia and industry, because they are at the core of most high-availability systems, but are difficult to design and implement. Because consensus is not solvable in asynchronous networks in the presence of faults [38], a large number of algorithms have been developed [19, 34, 48, 51, 57, 63], each of them solving consensus under different assumptions on the type of faults, and the degree of synchrony of the system. Moreover, many other problems in distributed systems can be reduced to consensus, e.g., atomic broadcast. Noteworthy examples of applications from industry that use consensus algorithms include the Chubby lock service [16], which uses the Paxos [51] consensus algorithm, and Apache Zookeeper which has a dedicated algorithm [48] for primary-backup replication.

In this paper we unify the modeling, programming, and verification of fault-tolerant distributed algorithms with a domain specific language, called PSYNC, that

- has a high-level, round-based, control structure that helps the programmer focus on the algorithmic questions rather than spending time fiddling with low-level network and timer code;
- compiles into working, efficient systems that preserve the important user-defined properties of high-level algorithms;
- is amenable to automated verification.

Despite the importance of fault-tolerant algorithms, no widely accepted programming model has emerged for these systems. The algorithms are often published with English descriptions, or, in the best case, pseudo-code. Moreover, fault-tolerant algorithms are rarely implemented as published, but modified to fit the constraints and requirements of the system in which they are incorporated [20]. General purpose programming languages lack the primitives to implement these algorithms in a simple way. A developer is forced to choose between low-level libraries, e.g., POSIX, to write network and timer code, or high-level abstractions, like supervision trees, that come with their own limitations. The first approach leads to programs with a very complex control structure that hides the algorithm's principles. The latter may not work because the abstraction's assumptions do not fit the system's requirements.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-3549-2/16/01...
http://dx.doi.org/10.1145/2837614.2837650

The complexity of fault-tolerant implementations makes them prone to error so they are prime candidates for automated verification. Their complexity is twofold: (1) the algorithms these implementations are based on, have an intricate data-flow and (2) the implementations have complex concurrent control structures, e.g., an unbounded number of replicas communicate via event-handlers, and complex data structures, e.g., unbounded buffers. Although fault-tolerant algorithms are at the core of critical applications, there are no automated verification techniques that can deal with the complexity of an implementation of an algorithm like Paxos.

The standard programming paradigm for implementing fault-tolerant distributed algorithms requires reasoning about asynchrony and faults separately. Programming languages provide only asynchronous communication primitives. A fundamental result on distributed algorithms [38] shows that it is impossible to reach consensus in asynchronous systems where at least one process might crash. Therefore, the algorithms that solve consensus make network assumptions finer than asynchrony, i.e., they need to reason about time explicitly. In order to reconcile the modeling of various network assumptions, the distributed algorithms community introduced computational models that uniformly model uniformly asynchrony and faults using an adversarial environment that drops messages [23, 40]. We take a programming language perspective on this matter and propose a domain specific language, PSYNC, that offers the programmer the illusion of synchrony, uses the adversarial environment to reason about faults and asynchrony, and efficiently executes on asynchronous networks.

High-level computational model. PSYNC is based on the *Heard-Of* model [23], which structures algorithms in communication-closed rounds [35]. A PSYNC program is defined by a sequence of rounds, and has a lockstep semantics where all the processes execute the same round. Each round consists of two consecutive operations: (1) a *send* method for sending messages and (2) an *update* method that updates the local state according to the messages received during the round. A round is communication-closed if all the messages are either delivered in the round they are sent or dropped. Communication-closed rounds provide a clear scope for the messages and the associated computations.

In the HO-model, a distributed system is a set of processes together with an adversarial environment, where the environment determines the set of messages received in a round. Each process has a *Heard-Of set*, denoted HO, which is a variable over sets of process identities exclusively under the control of the environment. HO-sets abstract the asynchronous and faulty behaviors of the network. In a given round, process p receives a message from process q if q sends a message to p and $q \in \mathsf{HO}(p)$. The network's degree of synchrony and the type of faults correspond to assumptions on how the environment assigns the HO-sets.

The round structure together with the HO-sets define an abstract notion of time. Using communication-closed rounds, PSYNC introduces a high-level control structure that allows the programmer to focus on the data computation performed by each process to ensure progress towards solving the considered problem, as opposed to spending time on the programming language constructs, e.g., incrementing message counters, and setting timers.

Runtime. We define a runtime that executes PSYNC programs on top of asynchronous faulty networks. The runtime defines an asynchronous semantics for PSYNC programs.

The main challenge in deriving asynchronous code from a PSYNC program is defining a procedure that decides when to go to the next round while allowing sufficiently many messages to be delivered. Our approached is based on timeouts: during an execution a process, after sending the messages for a round, accumulates messages for a duration specified by a timeout before calling the *update* method and moving to the next round. The timeout is a parameter that influences performance and can be fine-tuned to take advantage of the network and the algorithm specificity.

We have implemented PSYNC as an embedding in Scala. PSYNC programs link against a runtime system that manages the interface with the network and the resources used by the PSYNC program. The code runs on top of an event-driven framework for asynchronous network applications, uses UDP to transmit data, in a context where processes may permanently crash.

We show that for any PSYNC program $\mathcal{P}$ the asynchronous executions of $\mathcal{P}$ generated by the runtime are *indistinguishable* from the lockstep executions of $\mathcal{P}$. Indistinguishability is a relation between executions which roughly says that no process can locally distinguish between them. Intuitively, this relation is important because it is not possible to solve a problem that requires different answers for executions which are indistinguishable from the local perspective of processes.

Using indistinguishability we show that the asynchronous system defined by the runtime of a program $\mathcal{P}$ observationally refines the lockstep system defined by $\mathcal{P}$, that is every behavior of a client that uses the runtime can be reproduced if the client uses $\mathcal{P}$ instead, provided that the client operations that are not calls to $\mathcal{P}$ are commutative. This result makes it possible for developers to program using the lockstep semantics while the actual semantics is asynchronous.

Verification. Due to the complexity distributed systems have reached, we believe it is no longer realistic nor efficient to assume that high level specifications can be proved when development and verification are two disconnected steps in the software production process. PSYNC provides a simple round structure whose lockstep semantics leads to a smaller number of interleavings and simpler inductive characterizations of the set of reachable states. We have identified a large class of specifications such that if a PSYNC program satisfies the specification under the lockstep semantics, then its asynchronous semantics defined by the runtime satisfies the specification as well.

We have implemented a state-based verification engine for PSYNC, that checks safety and liveness properties. The engine assumes that the program is annotated with inductive invariants and ranking functions, and proves the validity of those annotations and the fact that they imply the specification. In general annotating a program with inductive invariants is a hard task even for an expert. The advantage of PSYNC lies in the simplicity of the required inductive invariants. Compared with asynchronous programming models, the round structure allows looking at the system's invariants at the boundary between rounds where the communication channels are empty.

Contributions. We introduce PSYNC a domain specific language for fault-tolerant algorithms.

- PSYNC makes it possible to write, execute, and verify high-level implementations of fault-tolerant algorithms.
- PSYNC has a simple lockstep semantics used to write programs which abstracts an asynchronous semantics defining the executions of PSYNC on general asynchronous networks.
- We prove that for any program $\mathcal{P}$, the runtime of $\mathcal{P}$ observationally refines $\mathcal{P}$, assuming clients with commutative operations.
- We prove that, for an important class of specifications including consensus, if a PSYNC program satisfies the specification, then its runtime system satisfies it as well.
- We have implemented and verified several fault-tolerant algorithms using PSYNC. We evaluate the *LastVoting* [23] that corresponds to Paxos [51] in the HO-model, in a distributed key-

value store, and show that the PSYNC implementation performs comparably to high-performance consensus implementations.

Sec. 2 introduces PSYNC using an example. Sec. 3 defines indistinguishability and observational refinement. Sec. 4 defines the domain specific language PSYNC, while Sec. 5 describes its runtime system. The verification techniques are presented in Sec. 6. Finally, Sec. 7 presents the experimental evaluation of PSYNC.

2. Overview

In this section we present the main features of PSYNC using the *LastVoting* [23] algorithm, shown in Fig. 1. This algorithm is an adaptation of the Paxos algorithm to the HO-model and solves consensus for integer values.

LastVoting communicates with clients using an interface defined by input operations, denoted `init`, and output operations, denoted `out`. A client sends a request to the *LastVoting* program using the input event `init`(v), which triggers a new instance the program, and the program replies to the client by generating an output event `out`(v'), where v, v' are integers.

A process that receives a client request `init`(v), starts executing *LastVoting* by calling its initialization function `init` with the same parameters. We assume that all processes constantly receive client requests so all process start executing *LastVoting*. Each process receives its initial value from the client via `init` events and stores it in the variable `x`. After the initialization phase processs typically have different `x` values. The goal of the algorithm is to make the processs agree on one of these values. The execution of *LastVoting* terminates when all processs agree. The client receives the agreed value from the process it communicates with, via an output even `out`(v'), where v' is the agreed value.

LastVoting roughly works by first establishing a majority of processes that agree on the same value. A designated process, called the coordinator, collects proposals for the value of `x` from the other processs and picks one of them (execution of the round `Collect`, Line 2). In the next round, `Candidate` (Line 9), the coordinator tries to impose the chosen value to a majority of processs. The round `Quorum` (Line 17) checks that this majority has been correctly established. If a quorum is formed then a decision is made, and all processs will accept it as the `decision` value, during the `Accept` round (Line 24). The execution of these rounds repeats, starting again with the `Collect` round, possibly with a different coordinator. The crux for the correctness of the algorithm is that strict majorities have a non-empty intersection. Since a decision requires the agreement of a majority of processes, the decision is unique.

Program structure. A PSYNC program is composed of an interface, a set of local variables, an initialization function, and a sequence of rounds. Each round defines a `send` and an `update` method. All processes execute in lockstep the same round. A part from the declared local variables processes use build-in read-only variables: `r` represents the round number, `n` the number of processes in the system, and `id` is the process unique identifier.

The `init` method takes as argument an integer. The `send` function returns the messages sent by the process executing it during the current round. Messages are of the form (recipient, payload), where the recipient is identified by its process identity. A set of messages is represented by a map indexed by process identities, associating the message payload to its recipient. The payload type might differ across rounds. For example, in `Collect` processes send pairs of integers while in `Quorum` they send one integer. The statement `broadcast(T)` returns a map from `ProcessId` to `T` where `T` is the type of payload. This map contains one message for every process in the system (all messages have the same payload).

The `update` takes as argument the set of messages received during the current round and modifies the local state of the process. The input of `update` is a map from sender (`ProcessId`) to payload (`T`). The `update` method can use external functions to perform sequential computations and modify the local state. For example, `mbox.valWithMaxTs` used in `Collect` scans the received messages and returns a value v such that (v, t) belongs to `mbox` and for any other (v', t') in `mbox`, $t' \leq t$.

Execution. A run of a PSYNC program starts with a call to the initialization function `init` (on each process). In *LastVoting* the `init` function initializes the value of `x` known locally by each process. The run continues with processes repeatedly executing, in lock step, the array of rounds defined by `phase` in the program.

In an ideal system (where no message is lost or delayed) an execution of one phase of the *LastVoting* program would result in the trace shown in Fig. 2. The processes proceed in lockstep, messages are delivered in time, and agreement is reached after four rounds. In reality an execution is more likely to look like the one shown in Fig. 3a, due to different delivery times for messages, different processors speeds, and crashes.

To reason about asynchrony and faults, the PSYNC semantics is based on the HO-model. Each process has a variable interpreted over sets of processes, called the HO-set. The messages received by a process p in round r, are the messages that were sent to p by the processes in its HO-set. At the beginning of each round, HO-sets are non-deterministically modified by the environment.

The HO-model models uniformly asynchronous behaviors and faults while providing the illusion of a lockstep semantics. Therefore, it is possible to reflect the faults in Fig. 3a using a lockstep semantics by setting the HO-sets to the appropriate values. Fig. 3b shows a lockstep execution where each process receives the same messages as in Fig. 3a. If a message is dropped by the network, e.g., the message p_2 sends to p_1 in `Collect`, then p_2 is not included in the HO of p_1. If a message is delayed far too long, then the sender is not included in the HO-set of the receiver. For example, in `Quorum`, the coordinator decides on a value if a majority of processes agrees with its proposal. Therefore, the coordinator moves to the next round without waiting for acknowledgements from all processes. In Fig. 3a the coordinator p_1 starts executing round `Accept` despite not receiving the acknowledgement sent by p_3 in round `Quorum`. However, this acknowledgement is eventually delivered when the coordinator is in the fourth round, but the message comes too late to influence the local computation. Therefore, it is as if the message was lost. In the lockstep execution, Fig. 3b, p_3 is not included in the HO-set of p_1 in the `Quorum` round. Finally, crashes do not directly impact the view of correct processes. Crashed processes are modeled using correct processes which are not included in the HO-set of any other process. It is as if all messages they send are dropped after the instant of the actual crash.

Specification and verification. We have developed a state-based verification engine for PSYNC programs. The specification of *LastVoting* includes properties like agreement, which says that all processes decided on the same value:

$$\Box(\forall p, p'.\ p.\texttt{decided} \wedge p'.\texttt{decided} \Rightarrow p.\texttt{decision} = p'.\texttt{decision}),$$

where p, p' are processes and $p.\texttt{decided}$, $p.\texttt{decision}$ is the value of the local variable `decided`, resp. `decision`, of process p.

The verification engine is based on deductive verification. This assumes that the program is annotated with inductive invariants and the engine proves the validity of the annotations and the fact that they imply the specification. The lockstep semantics is essential in order to have simple inductive invariants. For instance, the crux of the invariant that shows agreement is the existence of a majority of

```
interface
    init(v: Int); out(v: Int)

variable
    x: Int; ts: Int; vote: Int
    ready: Boolean; commit: Boolean
    decided: Boolean; decision: Int

//auxiliary function: rotating coordinator
def coord(phi: Int): ProcessID =
    new ProcessID((phi/phase.length) % n)

//initialization
def init(v: Int) =
    x := v
    ts := -1
    ready := false
    commit := false
    decided := false
```

```
val phase = Array[Round]( //the rounds
  Round /* Collect */ {
    def send(): Map[ProcessID, (Int,Int)] =
        return MapOf(coord(r) → (x, ts))
    def update(mbox: Map[ProcessID, (Int,Int)]) =
        if (id = coord(r) ∧ mbox.size > n/2)
            vote := mbox.valWithMaxTS
            commit := true },
  Round /* Candidate */ {
    def send(): Map[ProcessID, Int] =
        if (id = coord(r) ∧ commit) return
              broadcast(vote)
        else return ∅
    def update(mbox: Map[ProcessID, Int]) =
        if (mbox contains coord(r))
            x := mbox(coord(r))
            ts := r/4 },
  Round /* Quorum */ {
    def send(): Map[ProcessID, Int] =
        if ( ts = r/4 ) return MapOf(coord(r) → x)
        else return ∅
    def update(mbox: Map[ProcessID, Int]) =
        if (id = coord(r) ∧ mbox.size > n/2)
            ready := true },
  Round /* Accept */ {
    def send(): Map[ProcessID, Int] =
        if (id = coord(r) ∧ ready) return broadcast(vote)
        else return ∅
    def update(mbox: Map[ProcessID, Int]) =
        if (mbox contains coord(r) ∧ ¬decided)
            decision := mbox(coord(r))
            out(decision)
            decided := true
        ready := false
        commit := false })
```

A simplified version of *LastVoting* in PSYNC. The program has one phase defined by four rounds, The phase is executed in a loop. r contains the round number. The function `coord(r)` returns the identity of the coordinator of round r. Its identity changes between phases. In one phase, the coordinator collects proposals from the other replicas, picks one of them (`Collect`), and tries to impose it to the other replicas (`Candidate`). If a majority of processes agree with the coordinator's proposal (`Quorum`) then eventually all processes will accept this value as their decision (`Accept`).

Figure 1: The *LastVoting* consensus algorithm in PSYNC

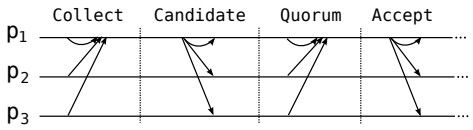
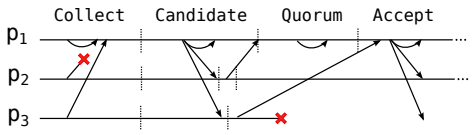
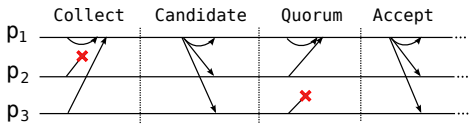


Figure 2: Execution of the *LastVoting* in a synchronous environment without faults. Process $\mathtt{p}_1$ is the coordinator. Dotted lines represent the boundaries between rounds.



(a) An asynchronous, faulty execution of the *LastVoting*



(b) Corresponding indistinguishable lockstep execution

Figure 3: Correspondence between the semantics and execution

processes that agree on a value v when at least one process decides:

$$\begin{array}{lll} & \forall p. & p.\mathtt{decided} = \mathit{false} \\ \vee & \exists v, t, A. & A = \{p \mid p.\mathtt{ts} \geq t\} \wedge |A| > n/2 \\ & & \wedge\, \forall p.\, p \in A \Rightarrow p.\mathtt{x} = v. \end{array}$$

Beyond safety properties, we are also interested in proving liveness properties, such as, $\Diamond(\forall p.\ p.\mathtt{decided})$. Because consensus is not solvable in asynchronous networks with faults, the network must satisfy liveness assumptions to ensure progress of the algorithm. The liveness assumptions impose constraints, typically lower bounds on the cardinality of the HO-sets. For example, proving that *LastVoting* eventually makes a decision requires a sequence of four rounds starting with `Collect` during which the environment picks values for the HO-sets such that:

$$|\mathsf{HO}(\mathtt{coord}(r))| > n/2 \wedge \forall q.\ \mathtt{coord}(r) \in \mathsf{HO}(q). \qquad (1)$$

For fault-tolerant distributed algorithms, the specification, the liveness assumptions, and the inductive invariants, require reasoning about set comprehensions, cardinality constraints, and process quantification. To express these properties and check their validity we use a fragment of first-order logic, called $\mathbb{CL}$, and its semi-decision procedure [33].

3. Indistinguishability and Observational Refinement

In this section we define the main theoretical concepts used in the paper: indistinguishability, observational refinement, and the relationship between them. The semantics of PSYNC programs is defined in terms of labeled transition systems. Most of the definitions given here are used in Section 5 and Section 6.

3.1 Definitions

A *transition system* is a tuple $TS = (P, V, A, s_0, T)$, where P is a set of processes, V is a finite set of variables, $V = \bigcup_{p \in P} V_p$ with V_p the set of local variables of process $p \in P$, A is a (possibly infinite) set of labels, $A = \bigcup_{p \in P} A_p$ with A_p the set of transition labels of process $p \in P$, $\Sigma = [P \to V \to \mathcal{D}] \uplus \{*\}$ is the state space of TS, where $*$ is a special state different from the other ones and $\mathcal{D}$ is the data domain where variables are evaluated, and $s_0 \in \Sigma$ is the initial state of the system. Let $\mathcal{A}^{tr}$ be the set of subsets of A such that each subset in $\mathcal{A}^{tr}$ contains at most one label from each A_p, with $p \in P$. The transition relations of TS is $T \subseteq \Sigma \times \mathcal{A}^{tr} \times \Sigma$.

A state $s \in \Sigma$ is a valuation of the processes variables. Given a process $p \in P$, $s(p)$ is the local state of p, which is a valuation of p's local variables, i.e., $s(p) \in [V_p \to \mathcal{D}]$. We use a special value $\perp$ to represent the state of crashed processes. When comparing local states, $\perp$ is treated as a wildcard state that matches any state.

An *execution* of a system TS is an infinite sequence $s_0 A_0 s_1 A_1 \ldots$ such that for all $i \geq 0$, $s_i \in \Sigma$, $A_i \in \mathcal{A}^{tr}$ ($A_i \neq \emptyset$), and $(s_i, A_i, s_{i+i}) \in T$. We denote by $[\![TS]\!]$ the set of executions of the system TS. A *run* is the sequence of states in an execution and a *trace* is the sequence of labels in an execution. We denote by $\mathsf{Runs}(TS)$, $\mathsf{Traces}(TS)$, the set of runs, respectively the set of traces, of a transition system TS.

The *projection* of an execution π on a process p, denoted $\pi\downarrow_p$, is obtained from π by keeping only the state of the variables and transition labels local to p, i.e., $\pi\downarrow_p = s'_0 A'_1 s'_1 \ldots$ where $s'_i = s_i(p)$ and $A'_i = A_i \cap A_p$.

Let π be a sequence alternating symbols from Σ and A. The *stuttering* closure of an execution π is the set of sequences obtained from π by replacing every state $s \in \Sigma$ with an arbitrary sequence of the form $s(\emptyset s)^*$. We write $\pi_1 \equiv \pi_2$ iff the execution π_1 is equivalent to the execution π_2 up to stuttering, i.e., there is an execution in the stuttering closure of both π_1 and π_2.

3.2 Indistinguishability

We define indistinguishability, an equivalence relation between executions of transition systems.

Definition 1 (Indistinguishability)**.** *Given two executions π and π' of a transition system TS, a process p cannot distinguish locally between π and π', denoted $\pi \simeq_p \pi'$, iff the projection of both executions on p agree up to finite stuttering, i.e., $\pi\downarrow_p \equiv \pi'\downarrow_p$.*

Two executions π and π' are indistinguishable*, denoted $\pi \simeq \pi'$, iff no process can distinguish between them, i.e., $\forall p \in P.\ \pi \simeq_p \pi'$.*

To relate executions of different systems, we first consider their projections on the sets W and L of common variables and respectively, labels, and relate the executions of these projected systems using the standard indistinguishability relation $\simeq$. The obtained relation is denoted $\simeq_{W,L}$.

Definition 2 (Indistinguishable systems)**.** *A system TS_1 is indistinguishable from a system TS_2 denoted $TS_1 \trianglerighteq TS_2$ iff they are defined over the same set of processes and for any execution $\pi \in [\![TS_1]\!]$ there exists an execution $\pi' \in [\![TS_2]\!]$ such that $\pi \simeq_{W,L} \pi'$ where $W = V_1 \cap V_2$ and $L = A_1 \cap A_2$.*

Notice that indistinguishability is agnostic to the order between transitions of different processes. For example, the two executions in Fig 3 are indistinguishable, although the `Quorum` round is executed during non-overlapping periods of time by process p_1 and p_2 in Fig. 3a while in Fig. 3b they are executed at the same time.

3.3 Observational refinement

We model clients and PSYNC programs as transition systems and their composition as the standard synchronized product (only the transitions labeled with common symbols are synchronized). The interface of a PSYNC program is the alphabet (of transitions labels) shared with the client.

A distributed *client* is a transition system defined by the parallel compositions of a set of n transitions systems over disjoint alphabets, each of this transition systems representing one client process in the system. This definition implies that transitions of different client processes commute, which intuitively means client processes do not synchronize with each other directly.

Definition 3 (Distributed client)**.** [1] *Let $TS_i = (\{p_i\}, V_i, A_i, s_0^i, T_i)$ be the transition system associated with a client process, with $A_i \cap A_j = \emptyset$ for all $1 \leq i \neq j \leq n$. Formally, the transitions system associated with the client is $C_{TS} = (P, V, A, s_0, T)$, where $P = \{p_1, p_2, \ldots p_n\}$, $V = \biguplus_i V_i$, $A = \bigcup_i A_i$, $s_0 = (s_0^1, \ldots, s_0^n)$, and $T \subseteq \Sigma \times A \times \Sigma$, with $\Sigma = [P \to V \to \mathcal{D}]$, such that $\Sigma(p_i) \in [V_i \to \mathcal{D}]$, and $(s, B, s') \in T$ iff for every $b \in B \cap A_i$ $(s(p_i), b, s'(p_i)) \in T_i$ and each processes takes at most one transition.*

Composition between a client and a* PSYNC *program. Let $\mathcal{P} = (P, V', A' \uplus I, s'_0, T')$ be a PSYNC program of interface I. A client $C = (P, V, A, s_0, T)$ communicates with $\mathcal{P}$ iff $I \subseteq A$ and $A \cap A' = \emptyset$. We define a program interface $I = \uplus_{p \in P} I_p$ to be the set of shared labels between $\mathcal{P}$ and C, that is I_p is the set of labels of the transitions taken by the process p, in both $\mathcal{P}$ and C.

For presentation reasons we assume that the client processes and the processes running $\mathcal{P}$ have the same identities. Otherwise the labels in the interface should include the identities of the clients and the identities of the processes running $\mathcal{P}$.

We define the behaviors of client C that interacts with a program $\mathcal{P}$ as the set of executions of a transition system denoted $C(\mathcal{P})$. The system $C(\mathcal{P})$ is obtained by (1) taking the asynchronous product between C and $\mathcal{P}$ and forcing that any transitions labeled by $b \in I$ is always taken simultaneously by C and $\mathcal{P}$, and (2) projecting out $\mathcal{P}$ from the product (we are only interested in the client), (3) after projecting out $\mathcal{P}$, we remove the transitions with no labels. We obtain a transition system with the same state space as the original client, but with fewer behaviors.

For example, a client communicates with *LastVoting* in Fig. 1 using the interface $I_{\texttt{init},\texttt{out}} = \{\texttt{init}_p(v), \texttt{out}_p(v) \mid p \in P, v \in \mathbb{Z}\}$. For the client, an $\texttt{init}_\texttt{p}(v)$ transition sends a request to process p executing *LastVoting* and makes the client wait for the reply $\texttt{out}_p(v')$. An $\texttt{init}_\texttt{p}(v)$ transitions of *LastVoting* corresponds a call to the initialization function `init` on process p with v as input parameter. Similarly $\texttt{out}_\texttt{p}(v')$ transitions are used to handle the replies between the service process p and the client process p.

In the following sections we introduce different semantics for PSYNC programs. Here, we define a general notion of refinement between different semantics.

Definition 4 (Observational Refinement)**.** *Let TS_1 and TS_2 be two transition systems and a common interface I. Then, TS_1* refines *TS_2 w.r.t. I denoted $TS_1 \sqsubseteq_I TS_2$, if for any client C,*

$$\mathsf{Runs}(C(TS_1)) \subseteq \mathsf{Runs}(C(TS_2)).$$

We say that TS_1 observationally refines TS_2 if every run of a client that uses TS_1 is also a run of the same client using TS_2.

Moreover, since we consider clients which do not impose an order between the transitions on different client processes, indistinguishability is equivalent with observational refinement.

Theorem 1. *Let TS_1 and TS_2 be two systems with a common interface I. If $TS_1 \trianglerighteq TS_2$ then $TS_1 \sqsubseteq_I TS_2$.*

Proof. (Sketch) The proof is based on the Corollary 43 from [37], which states that sequential consistency is equivalent with obser-

[1] We consider all distributed clients to be commutative by definition.

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{interface variable}^* \textit{ init phase} \\
\textit{interface} & ::= & \texttt{init}\colon \textit{type} \to () \quad (\textit{name}\colon \textit{type} \to ())^* \\
\textit{variable} & ::= & \textit{name}\colon \textit{type} \\
\textit{init} & ::= & \texttt{init}\colon \textit{type} \to () \\
\textit{phase} & ::= & \textit{round}^+ \\
\textit{round}_{\mathsf{T}} & ::= & \texttt{send}\colon () \to [P \mapsto \mathsf{T}] \quad \texttt{update}\colon [P \mapsto \mathsf{T}] \to ()
\end{array}
$$

Figure 4: PSYNC abstract syntax.

vational refinement when client transitions commute across processes. Indistinguishability is equivalent with sequential consistency, that is, $TS_1 \trianglerighteq TS_2$ iff TS_1 is sequentially consistent with TS_2. □

4. Syntax and Semantics of PSYNC

PSYNC is designed as a domain specific language embedded within a general purpose programming language. In this section we present the main programming constructs of PSYNC and its lockstep semantics.

Syntax. We give an abstract syntax for PSYNC programs in Fig. 4. A program has an interface, a number of local variables, an initialization operation `init`, and a non-empty sequence of rounds, called phase. Each process executes in a loop the sequence of rounds defined by the phase.

The interface is a set of functions, used by clients to interact with a PSYNC program. The interface includes an `init` operation, which is called by the client to send a request and multiple output functions, denoted $name$, used by the program to reply to the client's request.

Each round is parameterized by a type T which represents the payload of the messages. The messages are grouped in (partial) maps that associate processes, i.e., senders or recipients, to payloads. We use set-like notations for maps, where a map is set of pairs whose first elements are distinct. In a round, the parameters of `send` and `update` are defined over the same type T. However, different rounds can have different payload types. For instance, the payload of `Collect` in Fig 1 is `(Int,Int)` while the payload in the other rounds is `Int`.

The operations in a PSYNC program do not use directly any iterative control structure. For complex sequential computations they can use auxiliary operations implemented in the host language. The `init`, `send`, and `update` operations are assumed to terminate within a number of steps that depends on the number of processes and the input values of the initialization function. PSYNC is designed to facilitate the implementation of message passing concurrency. Proving total correctness of the sequential code executed by each process is orthogonal to the scope of PSYNC.

Lockstep semantics. Assuming a finite, non-empty set of n processes P, the state of a PSYNC program is represented by the tuple $\langle SU, s, r, msg, \mathsf{HO}\rangle$ where:

- $SU \in \{Snd, Updt\}$ indicates whether the next operation is send or update;
- $s \in [P \to V \to \mathcal{D}]$ stores the local states of the processes;
- $r \in \mathbb{N}$ is a counter for the executed rounds;
- $msg \subseteq 2^{P,\mathsf{T},P}$ stores the messages which are in transit between the SEND and UPDATE operations of a round;
- $\mathsf{HO} \in [P \to 2^P]$ evaluates the HO-sets for the current round.

The semantics of a PSYNC program is shown in Figure 5.

A transition is written as $S \xrightarrow{I,O} S'$ where S, S' are states, O is a set of labels from the interface, corresponding to observable transitions, I is a set of labels not in the interface corresponding to internal transitions. A client of a PSYNC program can only observe the transition labels in O.

$$
\textsc{Init}\quad \frac{\forall p \in P.\ * \xrightarrow{\texttt{init}(v_p)} s(p)}{* \xrightarrow{\emptyset,\{\texttt{init}_p(v_p) \mid p \in P\}} \langle Snd, s, 0, \emptyset, \mathsf{HO}\rangle}
$$

$$
\textsc{Send}\quad \frac{\begin{array}{c}\forall p \in P.\ s(p) \xrightarrow{\texttt{phase}[r].\texttt{send}(m_p)} s(p) \\ msg = \{(p,t,q) \mid p \in P \wedge (t,q) \in m_p\}\end{array}}{\langle Snd, s, r, \emptyset, \mathsf{HO}\rangle \xrightarrow{\{\texttt{send}_p(m_p) \mid p \in P\},\emptyset} \langle Updt, s, r, msg, \mathsf{HO}'\rangle}
$$

$$
\textsc{Update}\quad \frac{\begin{array}{c}\forall p \in P.\ mbox_p = \{(q,t) \mid (q,t,p) \in msg \wedge q \in \mathsf{HO}(p)\} \\ \forall p \in P.\ s(p) \xrightarrow{\texttt{phase}[r].\texttt{update}(mbox_p),o_p} s'(p) \\ r' = r+1 \qquad O = \{o_p \mid p \in p\}\end{array}}{\langle Updt, s, r, msg, \mathsf{HO}\rangle \xrightarrow{\{\texttt{update}_p(mbox_p) \mid p \in p\},O} \langle Snd, s', r', \emptyset, \mathsf{HO}\rangle}
$$

Figure 5: PSYNC semantics.

We consider the following shorthands: $|\texttt{phase}|$ is the number of rounds in a phase and $\texttt{phase}[r]$ is used to identify the $(r \bmod |\texttt{phase}|)$ round in a phase. For example, Fig. 1 has $|\texttt{phase}| = 4$ and $\texttt{phase}[3]$ identifies a `Quorum` round. The operation m of a round $\texttt{phase}[r]$ is $\texttt{phase}[r].m$. A transition $s(p) \xrightarrow{op,o} s'(p)$ says that p executes operation op in local state $s(p)$ and reaches local state $s'(p)$. The execution of op produces an observable transition o, i.e., o is in the interface.

Initially the state of the system is undefined, denoted by $*$. The first transition of every process p is to call the `init` operation upon receiving a client request via an $\texttt{init}_p$ event. The arguments of the `init` operation executed by process p match the values in the $\texttt{init}_p$ request sent by the client (see INIT in Fig. 5). The `init` operation does not return a value but initializes the state of the system. Initially, the round counter is 0, there are no messages in the system, and the first operation is Snd. An execution alternates the SEND and UPDATE transitions from Fig 5.

During a SEND transition, the messages sent by each process are added to a pool of messages msg. The messages in msg are triples of the form (sender, payload, recipient), where the sender and receiver are processes and the payload has type `T`. The triples are obtained from the map returned by `send` to which we add the identity of the process that executed `send`. The `send` operation does not affect the state of the processes. The values of the HO-sets are updated non-deterministically by the environment.

In an UPDATE step, messages are received and the `update` operation is applied locally in each process. The set of received messages is the input of `update`. A message is received only if the sender is in the receiver's HO-set. The `update` operation might produce an observable transition o_p. At the end of the round, msg is purged and r is incremented by 1.

To obtain a transition system as defined in Section 3, the fields SU, r, HO are copied locally on each process, the interface together with `send` and `update` are the labels of the transition system and the pool of messages msg is represented by a special *network* process whose only local variable is msg.

Environment assumptions. Many problems, such as consensus, are not solvable in asynchronous networks with faults [38]. Therefore, many algorithms make assumptions on the network and the faults in order to progress.

In PSYNC the network assumptions translate into assumptions on the environment actions. They are given as linear temporal logic (LTL) formulas over atomic propositions that constrain the values of the HO-sets. The classic taxonomy of distributed systems classifies in different categories the synchrony degree of the network, the reliability of the links, and the different types of process failures. The relation between the classic types of systems and the corresponding assumptions on the HO-sets is given in [23, Table 1].

In this work, we consider that the network assumptions are required only to guarantee liveness properties. Without making any assumption on the environment a program might never make any progress. For example, the environment can decide that the HO-sets are always empty and no message is delivered. In order to ensure termination *LastVoting* assumes that eventually there exists a sequence of four rounds, starting with `Collect`, where the coordinator is in the HO-set of every process, and during the `Collect` and `Quorum` rounds of this sequence the HO-set of each process contains at least $n/2$ processes. In LTL this environment assumption is expressed by the formula $\Box\Diamond(\psi \wedge \circ(\psi \wedge \circ(\psi \wedge \circ\psi)))$, where ψ is given in (1). Notice that at the end of these four rounds, all processes have decided the value proposed by the coordinator in the `Collect` round.

The lockstep semantics of a PSYNC program is defined by the set of executions of the transition system in Figure 5 that respect the environment assumptions given in the program.

Definition 5 (Lockstep execution)**.** *Given a* PSYNC *program $\mathcal{P}$ and a non-empty set of processes P, a lockstep execution of $\mathcal{P}$ is the sequence $*A_0 s_1 A_1 s_2 \ldots$ such that*

- *$*A_0 s_1$ is the result of the* INIT *rule;*
- *$\forall i.\ s_i A_i s_{i+1}$ satisfy the* SEND *or the* UPDATE *rule;*
- *the environment assumptions on* HO*-sets are satisfied.*

The set of lockstep executions of $\mathcal{P}$ is denoted by $[\![\mathcal{P}]\!]_{ls}$.

For any program $\mathcal{P}$, we consider that the environment assumptions of $\mathcal{P}$ are time-invariant, i.e., they are of the form $\Box\Diamond\varphi$, where φ is the network assumption that $\mathcal{P}$ relies on during its execution. Time invariance is important because we don't want the correctness of $\mathcal{P}$ to depend on the time it starts executing.

Any lockstep execution of *LastVoting* has a finite prefix defined by a sequence of rounds, called *bad rounds*, when the environment assigns values to the HO-set without respecting the assumptions. This prefix is followed by a sequence of *good rounds* when the environment assumptions are met. Moreover, all infinite executions of *LastVoting* are an alternation of sequences of bad, respectively good rounds. However, since consensus is a stable property, i.e., if it holds in a state than it holds in any of the following ones, and consensus is reached after the first sequence of good rounds, the local state of processes remains unchanged during the remaining sequences of bad or good rounds.

5. Runtime

In this section we define a runtime that executes PSYNC programs on asynchronous faulty networks. We describe the runtime algorithm, which induces an asynchronous semantics for PSYNC programs, and we show that a client cannot distinguish between the lockstep and the asynchronous semantics of PSYNC programs. Finally, we discuss how to tune the parameters of the runtime in relation to the network environment and liveness assumptions.

5.1 Runtime Algorithm

The runtime is responsible for executing PSYNC programs with an asynchronous semantics. Its most delicate task is to decide when a process moves to the next round. The problem is that, in an asynchronous network, it is not possible to distinguish between crashed and slow processes or between dropped and delayed messages. The lockstep semantics PSYNC deals uniformly with good and bad rounds due to the non-deterministic values of the HO-sets. So even if a round does not satisfy the liveness assumptions, the lockstep execution does not block. Therefore, the runtime cannot wait for every message as it would get stuck (some messages might be dropped or delayed forever). On the other hand, it needs to wait long enough to receive sufficiently many messages to guarantee progress of the system. Since there is no precise way to distinguish between bad and good rounds in the asynchronous semantics, the decision to switch rounds is based on approximations.

We consider an implementation of the round structure based on timeouts. Roughly, the set of messages received in one round equals the set of messages received within the time interval defined by a timeout. The timeout approach ensures that processes don't block during bad rounds, when the network behaves arbitrarily. However, if the timeout value is too small, the round switch might happen too fast and some messages are ignored at runtime even if they are properly delivered by the network. Therefore, the timeout values must be chosen carefully, to ensure that the runtime can simulate the good rounds. The values of the timeout are architecture and algorithm dependent, so the timeout is a parameter of the PSYNC runtime.

The runtime system is defined by the asynchronous composition of all processes in the network intersected with the network assumptions required to guarantee liveness properties. Given a PSYNC program $\mathcal{P}$, Fig. 6 shows the code executed by each process to run $\mathcal{P}$. Roughly, processes execute locally the same sequence of rounds as in $\mathcal{P}$, but the parallel composition is asynchronous.

The algorithm in Fig. 6 uses two while-loops. One iteration of the outer loop (line 9) executes one round. The inner loop (line 15) accumulates messages until a timeout is reached. Because the network is not synchronous, the runtime deals with messages of past or future rounds, i.e., messages tagged with round numbers strictly smaller or bigger than the process's current round number. Late messages are dropped (line 18), and a message from a future round forces the runtime to execute the outer loop until it catches up and reaches the round of the received message (line 11). The send and update operations, on line 10 and 26, are those defined in the executed PSYNC program $\mathcal{P}$. To deal with messages duplication the accumulated messages are stored in a set. The function `tryReceive(d)` tries to receive a message if one is available, or becomes available during the next `d` time units. If no message is available in this period then the method returns $\perp$.

The variable `to` has the same reference value across processes. It is used to measure locally a time interval of length `to` in reference time units. We assume that the processor's speed is not related to its clock and also we assume a bounded clock drift across processes. Therefore, processes can execute a different number of instructions while measuring the same interval. The function `currentTime` returns the value of a local clock. The implementation of `currentTime` ensures that processes can measure the elapsing of `to` reference time units. The clock drift is factored in the `currentTime` function.

Messages are tuples $(sender, payload, receiver, round)$, where $sender$, $receiver$ are the sender and respectively the receiver of the message, $payload$ is the content of the message, and $round$ is the sender's round number when the message was sent.

Asynchronous semantics In the following, given a PSYNC program $\mathcal{P}$, we define the asynchronous semantics of $\mathcal{P}$, induced by the runtime. A state is a tuple $\langle s \uplus s_r, msg\rangle$ where:

- $s \in [P \to V \to \mathcal{D}]$ is a valuation of the variables in $\mathcal{P}$;

```
1  //local variables
2  p                //initialized PSync process
3  to               //timeout
4  r := 0           //current round number
5  msg := ⊥         //last received message
6  mbox := ∅        //messages received but not yet processed
7  t := currentTime() //time at which the current round began
8
9  while (true) {
10   p.phase[r % p.phase.size].send() //send event
11   if (msg ≠ ⊥ ∧ msg.round = r) {
12     mbox := {msg}
13     msg := ⊥
14   }
15   while (msg = ⊥ ∧ currentTime() < t + to) {
16     msg := tryReceive(t + to - currentTime())//receive event
17     if (msg ≠ ⊥) {
18       if (msg.round < r) {
19         msg := ⊥
20       } else if (msg.round = r) {
21         mbox := mbox ∪ {msg}
22         msg := ⊥
23       }
24     }
25   }
26   p.phase[r % p.phase.size].update(mbox) //update event
27   r := r + 1
28   t := currentTime()
29   mbox := ∅
30 }
```

Figure 6: Algorithm to implement the round structure

- $s_r \in [P \to V_r \to \mathcal{D}]$ is a valuation of the variables introduced by the runtime;
- $msg \in [(P, \mathsf{T}, P, \mathbb{N}) \to \mathbb{N}]$ is a multiset of messages in transit.

In Fig. 7 we define the semantics of the most important instructions of the algorithm in Fig. 6. Each transition has the form $s \xrightarrow{I,O} s'$, where s, s' are global states and I, resp. O, are the internal, resp. observable labels of the transition. The runtime of a program $\mathcal{P}$ interacts with a client using the interface of $\mathcal{P}$ defining the observable transitions. By considering msg a specific *network* process, we obtain a transition system as defined in Section 3.

Local process transitions. The SEND rule states that the messages sent by one process are tagged with the current round of the sender, i.e., $s(p).r$, and added to the global pool of messages, msg. The messages sent by one process in round r are defined by the `send` operation of the same round from $\mathcal{P}$. The RECEIVE1 and RECEIVE2 rules define the reception of a message. The RECEIVE2 rule describes a failed reception due to a timeout. The WAIT rule models a process waiting for a message. The UPDATE rule states that the semantics of `update` when called by a process whose current round is r is the the semantics of the `update` operation of round r from $\mathcal{P}$.

The CRASH, CRASHED, DUPLICATE, DROP describe the fault model. Progresses can crash, but do not recover. Messages can be duplicated and dropped by the network. The CRASHED rule states that crashed process do not modify the global state.

The runtime does not include HO-sets. To evaluate the environment assumptions from $\mathcal{P}$ on asynchronous executions, the HO-sets are replaced with the set of received messages. That is, any constraint on $\mathsf{HO}(p)$ in round r, is replaced by the same constraint over $\texttt{mbox}_p$, the set of messages received by process p in round r. The `update` called by process p in round r takes $\texttt{mbox}_p$ as argument. The environment assumptions must hold only for the non-crashed processes.

Local transitions

SEND

$$\frac{p \xrightarrow{\texttt{send}(ms)} p' \qquad msg' = \{(p, m, q, p.r) \mid (m, q) \in ms\} \cup msg}{\langle p, msg\rangle \xrightarrow{\{\texttt{send}_p(ms)\},\emptyset} \langle p', msg'\rangle}$$

RECEIVE1

$$\frac{m \in msg \qquad m.receiver = p \qquad p \xrightarrow{receive(m)} p' \qquad msg' = msg \setminus m}{\langle p, msg\rangle \xrightarrow{\{receive_p(m)\},\emptyset} \langle p', msg'\rangle}$$

RECEIVE2

$$\frac{\forall m \in msg.\ m.receiver \neq p \qquad p \xrightarrow{receive(\bot)} p'}{\langle p, msg\rangle \xrightarrow{\{receive_p(\bot)\},\emptyset} \langle p', msg\rangle}$$

UPDATE

$$\frac{m = p.mbox \qquad p \xrightarrow{\texttt{update}(m),\alpha_p} p'}{\langle p, msg\rangle \xrightarrow{\{\texttt{update}_p(m)\},\{\alpha_p\}} \langle p', msg\rangle}$$

DROP

$$\frac{ms' \subset ms}{\langle p, ms\rangle \xrightarrow{\emptyset,\emptyset} \langle p, ms'\rangle}$$

CRASH

$$\frac{p \neq \bot \qquad p' = \bot}{\langle p, msg\rangle \xrightarrow{\emptyset,\emptyset} \langle p', msg\rangle}$$

WAIT

$$\frac{p \neq \bot}{\langle p, msg\rangle \xrightarrow{\{wait_p\},\emptyset} \langle p, msg\rangle}$$

CRASHED

$$\frac{p = \bot}{\langle p, msg\rangle \xrightarrow{\{\bot_p\},\{\bot_p\}} \langle p, msg\rangle}$$

DUPLICATE

$$\frac{m \in ms \qquad ms' = ms \cup \{m\}}{\langle p, ms\rangle \xrightarrow{\emptyset,\emptyset} \langle p, ms'\rangle}$$

Global transitions

INIT

$$\frac{\forall p \in P.\ * \xrightarrow{\texttt{init}(v_p)} s(p)}{* \xrightarrow{\emptyset,\{\texttt{init}_p(v_p) \mid p \in P\}} \langle s, \emptyset\rangle}$$

PARALLEL COMPOSITION

$$\frac{\begin{array}{c}P' \subseteq P \qquad msg = \uplus_{p\in P'} msg_p \qquad msg' = \cup_{p\in P'} msg'_p \\ I = \cup_{p\in P'} I'_p \qquad O = \cup_{p\in P'} O'_p \qquad \forall p \in P \setminus P'.\ s(p) = s'(p) \\ \forall p \in P'.\ \langle s(p), msg_p\rangle \xrightarrow{I_p,O_p} \langle s'(p), msg'_p\rangle\end{array}}{\langle s, msg\rangle \xrightarrow{I,O} \langle s', msg'\rangle}$$

Figure 7: Asynchronous semantics of a PSYNC program $\mathcal{P}$. The local, respectively global, transitions are defined over local, respectively global states. We give the important local transition of a process executing the algorithm in Fig. 6. The rules SEND, RECEIVE, and UPDATE correspond to Line 10, 16, and 26 of the algorithm. The semantics of `send` and `update` are given by the program $\mathcal{P}$.

Definition 6 (Asynchronous executions)**.** *The set of asynchronous executions of a* PSYNC *program $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]_a$, is the set of executions of the transition system in Fig. 7 which satisfy the environment assumptions defined in $\mathcal{P}$.*

5.2 Correctness

We have introduced in Section 4 a lockstep semantics for PSYNC programs that helps the developer to focus on the algorithmic task. However, PSYNC programs are not executed only on top of synchronous networks. The runtime algorithm in Section 5.1 induces an asynchronous semantics for PSYNC programs which allows executing PSYNC programs on a wide variety of networks. In this section we state one of the main results of the paper which shows that the lockstep semantics models the asynchronous semantics faithfully.

Theorem 2. *For any* PSYNC *program $\mathcal{P}$, the transition system defining the asynchronous semantics of $\mathcal{P}$ is indistinguishable from the transition system defining the lockstep semantics of $\mathcal{P}$, i.e., $\forall \mathcal{P}.\ [\![\mathcal{P}]\!]_a \trianglerighteq [\![\mathcal{P}]\!]_{ls}$.*

Proof. (Sketch) Given an execution π from $[\![\mathcal{P}]\!]_a$, we define an execution π' such that $\pi' \in [\![\mathcal{P}]\!]_{ls}$ and $\pi \simeq \pi'$. We recall that the local variables V declared in $\mathcal{P}$ are modified only by the `update` operation, which has the same semantics in both $[\![\mathcal{P}]\!]_{ls}$ and $[\![\mathcal{P}]\!]_a$. Therefore, in define π' we need to show that (1) we can associate to every round r of π an execution of the same round under the PSYNC semantics, with the same output for the send operation and the same input for the `update` operation and (2) this sequence of PSYNC rounds satisfies the environment assumptions.

The execution π' is build from π by eliminating all receive steps, and shifting `send` and `update` operations to the left and right such that they execute in lockstep. Shifting the `send` and `update` of a process preserves the order of the operations within that process. For each round r the environment defines $\mathsf{HO}(p)$ to be the identity of the senders of the set of messages delivered to p tagged by round r. Without loss of generality, in $[\![\mathcal{P}]\!]_a$ we assume that at each round every process sends a message to every other process. If the algorithm sends less messages we can introduce additional *ghost* messages for the proof purpose.

Notice that the asynchronous executions π obeys to the environment assumptions, translated as constraints on mailboxes, i.e., the set of received messages of each round. Since the the HO-sets are defined from the processes mailboxes the execution π' respects also the environment assumption given in $\mathcal{P}$.

Finally, the crashed processes, represented by $\perp$ in π, can be matched to any lockstep execution where the corresponding processes do not appear in the HO set of any process after the crash.

The execution π is indistinguishable from π' because (1) for every round the state preceding the `send`, resp. `update`, operations is the same in π and π'; (2) for every round the `update` operations have the same inputs in both π and π'; (3) $receive$ steps in the runtime trace correspond to stuttering in the lockstep trace. □

Corollary 1. *For any* PSYNC *program $\mathcal{P}$, $[\![\mathcal{P}]\!]_a \sqsubseteq [\![\mathcal{P}]\!]_{ls}$.*

Theorem 2 ensures that the asynchronous semantics of PSYNC is a refinement of the lockstep one, independently of the timeout values. However, if the timeout is not carefully chosen the set of asynchronous executions in the semantics of PSYNC might be empty because the timeout is too short and it does not allow the environment assumptions to be satisfied. The asynchronous executions which don't respect the environment assumptions loose the progress guarantees of the PSYNC program.

Nevertheless, if we include in the asynchronous, respectively the lockstep, semantics of a program $\mathcal{P}$, the executions which do not respect the environment assumptions, the refinement relation between the two semantics still holds. This result is a corollary of Theorem 2.

Corollary 2. *For any asynchronous execution of a program $\mathcal{P}$, that does not respect the environment assumptions there exists an indistinguishable lockstep execution of $\mathcal{P}$ that does not respect the environment assumptions.*

5.3 Timeout Estimation

In this section we define an under-approximation of the timeout values ensuring that the runtime produces a non-empty set of asynchronous executions that respect the environment assumptions. To compute this estimation we consider partial synchronous networks.

Partial synchrony means that the network alternates between bad and good time periods, where during a good period the communication and the processes are synchronous while through a bad period they are asynchronous.

Definition 7 (Synchronous network)**.** *A network is synchronous if there exists Θ and Δ two positive integers, such that:*

- *Θ is the minimal time interval in which any process is guaranteed to take a step;*
- *Δ is the maximal transmission delay between any two processes.*

Intuitively, Θ corresponds to process synchrony and Δ to communication synchrony. Their values can be obtained by measuring the latency of processors and the bandwidth of the network. Both are required in order to solve problems like consensus [31]. When the network is asynchronous there are no bounds on the communication delay and relative speed.

Remark 1. *The transmission delay is typically much larger than the time required to ensure a computation step, i.e., $\Delta \gg \Theta > 0$.*

Definition 8 (Partially synchronous network [34][2])**.** *A network is* partially synchronous *if the constants Θ, Δ exist, are known, and eventually hold after a time gst, called the global stabilization time.*

Def. 8 characterizes the partial synchrony assumption w.r.t. an initial time. *Wlog* we assume this time coincide with the start of the program execution to avoid depending on the time processes start. A network is partially synchronous if it alternates between good and bad time periods, where the good periods are as long as needed. The required length of good periods depends on the program. Def. 8 takes the supremum of good periods for any program to allow a general statement about liveness of the system.

To characterize the partial synchronous executions of the runtime we need to reason about the message delays and the speed of the processes. Therefore, we add a reference time to the executions generated by the system of transitions defined in Fig. 7 using a function τ. τ maps each state s_i in π to a time in $\mathbb{N}$, it is monotonic: $\forall i, j.\ i \leq j \Rightarrow \tau(s_i) \leq \tau(s_j)$, and in any finite time interval each process takes only a finite number of steps.

An execution $\pi = s_0 A_0 s_1 A_2 \ldots$ of the transition system in Fig. 7 satisfies the partially synchrony assumption iff there exits a global stabilization time, gst, s.t. $\pi = \pi_a \pi_s$ and

- for any $s \in \pi_s$, $\tau(s) > gst$;
- for any i, j such that $\tau(s_i) > gts$ and $\tau(s_j) - \tau(s_i) \geq \Theta$, every process takes at least one step in the sequence $s_i \ldots s_j$;
- for any i,j such that $\tau(s_i) > gst$, $\mathtt{send}_p(ms) \in A_i$, and $receive_q(m) \in A_j$ with $m \in ms$, either the delay is $0 < \tau(s_j) - \tau(s_{i+1}) \leq \Delta$ or if $k \in [i, j]$ s.t. s_k is the first state with $\tau(s_k) - \tau(s_{i+1}) > \Delta$ then q does not take any WAIT or RECEIVE2 step in the $s_k \ldots s_j$ interval;
- there is no message duplication after gts[3].

Remark 2. *Because the environment assumptions are time invariant, i.e., they are LTL formulas of the form $\Box\Diamond\varphi$, we consider that $\pi = \pi_a \pi_s \in [\![\mathcal{P}]\!]_a$ satisfies the environment assumption iff π_s satisfies $\Box\Diamond\varphi$.*

[2] In [34] a second definition is proposed where Θ and Δ, are unknown but they hold from the beginning. Our results hold under both definitions of partial synchrony. We have chosen Def. 8 for performance reason.

[3] Limiting duplication only simplifies the proof. It is possible to tolerate a bounded amount of duplication after gts. The time to process the duplicate messages has to be factored in the timeout.

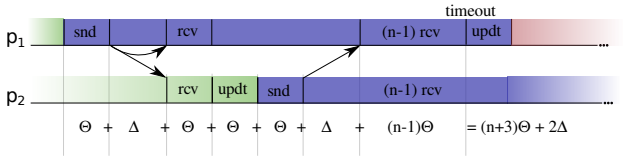


Figure 8: Processes synchronizing. The colors represent rounds.

In the following we define the timeout values using polynomial functions depending on Θ, Δ, and n (the number of processes). We recall that the HO-sets are an abstraction of (1) the synchrony degree and (2) the types of faults. The computed timeouts ensure that the degree of synchrony required by the environment assumptions is captured by the runtime (the round structure is correctly implemented). The values of the timeout which include an estimation for faulty behaviors are algorithm dependent, and have larger values. Since the timeout is a parameter of the runtime it can be tuned to meet the algorithms specificity. First, we derive a sufficiently large timeout value to ensure liveness of any algorithm when the network is synchronous and then we compute the timeout for partially synchronous networks. We assume that every process periodically communicates with every other process. If the algorithm does not send messages, the runtime inserts heatbeat messages. Communication is needed for synchronization.

Definition 9 (Synchronous execution)**.** *An execution of $[\![\mathcal{P}]\!]_a$ is called* synchronous *if in any time interval of length Θ each process takes at least a* `send`*,* `update`*, or* receive *step and every message sent in a given round is received in the same round, dropped by the network, or the receiver has crashed.*

Lemma 1. *If the network is synchronous any execution of $[\![\mathcal{P}]\!]_a$ is synchronous if the value of the timeout is at least $(n+1)\Theta + \Delta$.*

When the network is synchronous the slowest process takes exactly one step within Θ time units. Therefore, the maximal duration of a round is given by the slowest process and it equals $(n+1)\Theta + \Delta$ reference time units. $(n+1)\Theta$ is needed to receive $n-1$ messages from the other processes (excluding itself) and execute the send, and the update method, and Δ is the maximal duration messages are in transit. The timeout needs to be at least $(n+1)\Theta + \Delta$. This timeout slows down the faster processes such that the slower processes have enough time to process all the messages sent in the current round before the faster processes move to the next round.

Lemma 2. *If the timeout is at least $(n+3)\Theta+2\Delta$ and the network is partially synchronous, any execution π of $[\![\mathcal{P}]\!]_a$ is of the form $\pi = \pi_a\pi_s$ where π_a is finite and π_s is synchronous.*

Proof. Let s be the first state after gts, i.e., $\tau(s) \geq gst$. We divide the asynchronous prefix in two parts $\pi_a = \pi_1\pi_2$, where π_2 states with the state s. We show that the length of π_2 is bounded.

The actual computation time for a round is $(n+2)\Theta$, the remaining time $\Theta + 2\Delta$ is the slack required to take into account message delays and relative speed. After gts, the processes can use the slack time in each round to process more messages than the number of messages sent during that round. Because π_1 is finite (follows from the definition of partial synchrony), the number of messages that separate the fastest process from the slowest is also finite. Thus, the slowest process will eventually consume all these messages. When there are no more pending messages, the process are within one round of each others, but their respective round start might still be offset by more than $3\Theta + \Delta$.

Consider the case when the slowest process is in round r and receives a message sent by a process in round $r+1$. This message will be received in at most $\Theta + \Delta$ time units after it was sent. The catch-up mechanism of the runtime will force the slowest process to directly call `update` and move to the next round, taking 2Θ. Notice that the slowest process call `send` $3\Theta + \Delta$ time units after the fastest process called the `send` operation of the same round. When the slowest process starts receiving messages for the round $r+1$, it will be at most $3\Theta + \Delta$ behind the fastest process. Therefore, the fastest process will receive the messages from the slowest process in at most $4\Theta + 2\Delta$ time units after it started the round $r+1$. As their might be $(n-1)$ such messages, the timeout needs to be at least $(n+3)\Theta + 2\Delta$ to receive the remaining messages for the round $r+1$. Fig. 8 illustrates how the synchronizations happens.

After the system reaches synchrony. The remaining suffix π_s stays synchronous, the timeout is greater than the sum of $(n+1)\Theta + \Delta$ (the timeout for the synchronous case) and $3\Theta + \Delta - 1$ (the maximal slack minus the message send by a process to itself which can be processed more quickly). □

5.4 Extensions

In the above, we focus on the case of partially synchronous systems where crashed processes do not recover. However, PSYNC is not restricted to this assumption and the runtime can be adapted to a wide variety of systems. We give a short overview of the changes required to implement a round structure under different network assumptions.

Alternative definition of partial synchrony. The runtime algorithm also works with the definition of partial synchrony where the bounds Θ and Δ exist during the entire execution but are not known. In this setting, the runtime must use incremental timeout. More precisely, the initial value of the timeout is greater than 0 and when a message from a previous round is received, i.e., on line 19 in Fig. 6, the timeout is incremented with a fixed value.

Crash-recovery. From the perspective of the HO-model crash-recovery is similar to network partition. A process that has crashed but not yet recovered has an empty HO-set and is not in the HO-sets of any other process. However, from an implementation perspective, to allow recovery, the runtime must store in stable memory, i.e., hard disk, the state of the system before sending messages (Fig. 6, line 10). Upon recovery, the system resumes from the last state in the stable memory.

Byzantine failures. Byzantine failures require a more complex algorithm to implement the round structure. In that setting, the round structure is akin to clock synchronization and supporting arbitrary transient faults can be done through the self-stabilizing *pulsing* algorithm [30]. The decision moving to the next round requires receiving enough messages ($2/3$ majority). The messages need to be signed (HMAC) to avoid one process trying to impersonate another [18] and more complex message filtering strategies are needed to prevent a few malicious processes from flooding the correct processes [26]. We plan to extend the PSync runtime to support Byzantine faults in our future work.

6. Verification

In this section we identify a class of specifications such that if the lockstep semantics of a PSYNC program satisfies the specification then its asynchronous semantics satisfies it as well. Roughly, the refinement relation between these semantics preserves the local properties of processes. We underline the advantages of PSYNC for automated verification. Finally, we present a deductive verification engine for PSYNC.

6.1 From Verified PSYNC to Verified Runtime Executions

We develop a state-based verification engine for PSYNC which considers that specifications are sets of program runs. A *state-based specification Spec* for a program $\mathcal{P}$ is a set of sequences of states in $\Sigma = [P \to V_p \to D]$, where V_p is a subset of the local variables declared in the program. The specification does not talk about the HO-sets.

For example, consensus is defined by the conjunction of four properties: (1) Agreement, all processes decide on the same value; (2) Validity, the decision is the initial value of a process; (3) Irrevocability, a process cannot change its decision; (4) Termination, all correct processes eventually decide. These properties correspond to the following set of runs, denoted $Consensus$:

$$
\begin{aligned}
&*s_0 s_1 s_2 \ldots \in Consensus \Leftrightarrow \\
&\quad \exists q.\, \forall p.\, \forall i.\ s_i(p).\texttt{decided} \Rightarrow s_i(p).\texttt{decision} = s_0(q).\texttt{x} \\
&\wedge\, \forall p, i.\ s_i(p).\texttt{decided} \Rightarrow s_{i+1}(p).\texttt{decided} \\
&\wedge\, \forall p, i.\ s_i(p).\texttt{decided} \Rightarrow s_i(p).\texttt{decision} = s_{i+1}(p).\texttt{decision} \\
&\wedge\, \forall p.\ \exists i.\ s_i(p).\texttt{decided},
\end{aligned}
$$

where p, q denote a process, `decided`, `decision` are variables declared in the program, and s_0, s_i are states in Σ.

A program $\mathcal{P}$ satisfies a state-based specification *Spec* if all the runs of $\mathcal{P}$ under the lockstep semantics are included in *Spec*, i.e., $\mathsf{Runs}^{ls}(\mathcal{P})\lfloor_{V_{Spec}} \subseteq$ *Spec*, where $\mathsf{Runs}^{ls}(\mathcal{P})$ is the set of runs of $\mathcal{P}$ under the lockstep semantics, and $\lfloor_{V_{Spec}}$ denotes the projection on the variables in V_{Spec} used in *Spec*. We associate to each set of runs a transition system that produces them.

Transition system associated with a specification. Given a state-based specification *Spec* we build $TS(Spec) = (P, V_{Spec}, A, s_0, T)$ the transition systems associated with *Spec*, where $A = \emptyset$ and $s_0 = *$, such that $\mathsf{Runs}(TS(Spec)) = Spec$.

We consider state-based specifications, therefore to prove that the lockstep, respectively the asynchronous, semantics refine the specification, we need to relate sequences of observable program transitions to the specification. To achieve this, we introduce input-output interfaces which associate the labels of observable process transitions to sets of process local states.

Definition 10 (Input-Output Interface)**.** *Let $TS = (P, W, A, s_0, T)$ be a transition system. Let $I = \cup_{p \in P} I_p$ be a subset of A and f be a mapping from I to sets of states over the variables $W = \cup_{p \in P} W_p$. The pair (I, f) is an* input-output interface *of system TS if for any process p, for any $a \in I_p$, and for any $(s, B, s') \in T$, $a \in B$ iff $s(p)\lfloor_{W_p} \notin f(a) \wedge s'(p)\lfloor_{W_p} \in f(a)$.*

Given a program $\mathcal{P}$ with interface I and a state-based specification *Spec* for $\mathcal{P}$ let $TS(Spec, f)$ be the transition system obtained from $TS(Spec)$ by setting $A = I$ and adding labels in I to the transitions in $TS(Spec)$ such that (I, f) is an input-output interface of $TS(Spec, f)$. In this case, the pair (I, f) is also called the input-output interface of *Spec* w.r.t. the program $\mathcal{P}$.

The input-output interface of $Consensus$ w.r.t *LastVoting*, called (I_C, f_C), associates, for every $p \in P$, $\texttt{init}_p(v_p)$ with the first initialized state when $s(p).\texttt{x} = v$, and $\texttt{out}_p(v_p)$ to the states where `decided` is set to true and `decision` is v. The mapping f_C relates $(\texttt{init}_p(v)$ to $\{s \mid s(p).\texttt{x} = v \wedge s(p).\texttt{ts} = -1\}$ and $\texttt{out}_p(v)$ to $\{s \mid s(p).\texttt{decided} \wedge s(p).\texttt{decision} = v\})$.

Proposition 1. *For any* PSYNC *program $\mathcal{P}$, specification Spec, and input-output interface (I, f) of Spec w.r.t. $\mathcal{P}$:*

$$\mathsf{Runs}^{ls}(\mathcal{P})\lfloor_{V_{Spec}} \subseteq Spec \Rightarrow [\![\mathcal{P}]\!]_{ls}\lfloor_{V_{Spec},I} \subseteq [\![TS(Spec, f)]\!],$$

where $\lfloor_{V_{Spec},I}$ keeps only transition labels in I and projects the states on the variables in V_{Spec}.

The asynchronous semantics contains more executions than the lockstep semantics. However, these executions are indistinguishable from the lockstep ones. Therefore, if the specification is closed under indistinguishability, then the specification is preserved from the lockstep semantics to the asynchronous one. Formally, given a transition system TS we denote by $Closure_{\simeq}(TS)$ its closure w.r.t. the indistinguishability relation, i.e., $[\![Closure_{\simeq}(TS)]\!] = \{\pi \mid \exists \pi' \in [\![TS]\!].\ \pi \simeq \pi'\}$.

Theorem 3. *For any* PSYNC *program $\mathcal{P}$, specification Spec, and input-output interface (I, f) of Spec w.r.t. $\mathcal{P}$, if $\mathcal{P}$ satisfies Spec and $TS(Spec, f)$ is closed under indistinguishability, then*

$$[\![\mathcal{P}]\!]_a \trianglerighteq [\![TS(Spec, f)]\!] \text{ and } [\![\mathcal{P}]\!]_a \sqsubseteq_I TS(Spec, f).$$

Proof. Let $\pi \in [\![\mathcal{P}]\!]_a$ be an asynchronous execution of $\mathcal{P}$. Th. 2 implies that there exists $\pi' \in [\![\mathcal{P}]\!]_{ls}$ s.t. $\pi \simeq \pi'$. Moreover, since V_{Spec} is included in the set of program variables in $\mathcal{P}$ and the interface I is common for $\mathcal{P}$ and its runtime, (and indistinguishability between the runtime and PSYNC holds w.r.t. all the variables in $\mathcal{P}$ and all common labels), it implies that $\pi\lfloor_{V_{Spec},I} \simeq \pi'\lfloor_{V_{Spec},I}$.

Since $\mathcal{P}$ satisfies the specification, from Prop. 1 it follows that $\pi' \lfloor_{V_{Spec},I} \in [\![TS(Spec, f)]\!]$. Finally, since $[\![TS(Spec, f)]\!]$ is closed under indistinguishability $\pi \lfloor_{V_{Spec},I} \in [\![TS(Spec, f)]\!]$. Therefore, $[\![\mathcal{P}]\!]_a \trianglerighteq [\![TS(Spec, f)]\!]$ and, using Th. 1, it follows that $[\![\mathcal{P}]\!]_a \sqsubseteq_I TS(Spec, f)$. □

In the asynchronous semantics the specification applies only to correct processes, e.g., showing that every process decides means that every correct process decides in the runtime executions. For clients interacting with a PSYNC program, we assume that the failure of a process carries over to the client.

Proposition 2. *Consensus is closed under indistinguishability and if a program $\mathcal{P}$ satisfies Consensus then $[\![\mathcal{P}]\!]_a \sqsubseteq_{I_C} [\![\mathcal{P}]\!]_{ls} \sqsubseteq_{I_C} TS(Consensus, f_C)$.*

Roughly, consensus is closed under indistinguishability because it does not impose an order on the updates performed on different processes: processes can decide on a value in any order and at any time. The same reasoning also holds for other agreement specifications, like the k-set agreement [25], or the lattice agreement [36].

6.2 Benefits of PSYNC for Verification

Distributed algorithms are challenging to verify because of several sources of unboundedness. Messages come from unbounded domains, the number of processes is a parameter, and channels may also be unbounded. Using communication-closed rounds and a lockstep semantics helps mitigate or avoid these challenges.

Model checking. Model checking techniques are based on algorithms that explore the system's reachable states. It requires a fixed number of finite state processes. With an asynchronous semantics, a model checker explores all the possible interleavings of processes transitions and suffers from combinatorial explosion. The lockstep semantics of PSYNC abstracts away many of these interleavings. Another difficulty comes from the communication channels. Unbounded FIFO channels cause undecidability even for two processes [15]. Making the channels lossy [1] and fixing the number of processes makes the problem non-primitive recursive [68]. Weaker channel models are usually at least EXPSPACE-hard for verification. Communication-closed rounds sidestep this difficulty.

Deductive verification. Deductive verification relies on user provided inductive invariants and ranking functions. The invariants describe an over-approximation of the set of reachable states which is inductive w.r.t. the program transitions. Ranking functions show progress toward satisfying the program goals. However, finding

these annotations is not easy even for an expert. Automated techniques, such as static analysis, are far from being able to generate these annotations automatically for our targeted class of systems.

The lockstep semantics leads to much simpler invariants, because they are required to describe the set of reachable states only at the boundaries between rounds.

In the literature, the HO-model has been shown to be suited for verification using bounded state-space exploration [21, 71–73] and interactive theorem provers [22, 24, 27, 58].

6.3 A Verifier for PSYNC

We consider specifications that include both safety and liveness properties written in LTL with state properties in the logic $\mathbb{CL}$ [33]. The verifier inputs are the specification and a PSYNC program annotated with inductive invariant candidates. The verifier checks the validity of the invariants and that they imply the specification by generating verification conditions that can be discharged using an SMT solver.

Expressiveness. $\mathbb{CL}$ is a first-order logic over sets of program states. The variables are interpreted over the different types declared in the program. The value of the program variable `x` of type `T` of a process p is denoted in the logic by the term $\mathtt{x}(p)$, where `x` is a function of type $P \to \mathtt{T}$ with P being the type of process ids. To characterize global states, $\mathbb{CL}$ uses universal quantification over variables of type P, set comprehensions, and cardinality constraints.

The programmer provides the inductive invariant candidates and the pre/post conditions of the `send`, and `update` functions. Typically the correctness argument for consensus solving algorithms, which are captured in the inductive invariants, is centred around the existence of a *majority of processes* that support a decision. For example, the formula $\exists v.\ |\{p \mid \mathtt{x}(p) = v\}| > \mathtt{n}/2$ defines a majority ($> \mathtt{n}/2$) of processes that agree on value v using a comprehension, where `x` is the function symbol associated with the local variable `x`.

The inductive invariant that shows agreement in *LastVoting* is

$$\begin{array}{lll} & \forall p. & \neg\mathtt{decided}(p) \wedge \neg\mathtt{ready}(p) \\ \vee & \exists v, t, A. & A = \{p \mid \mathtt{ts}(p) \geq t\} \ \wedge \ t \leq r/4 \\ & \wedge & |A| > n/2 \ \wedge \ \forall p.\, p \in A \Rightarrow \mathtt{x}(p) = v \\ & \wedge & \forall p.\, \mathtt{decided}(p) \Rightarrow \mathtt{decision}(p) = v \\ & \wedge & \forall p.\, \mathtt{commit}(p) \vee \mathtt{ready}(p) \Rightarrow \mathtt{vote}(p) = v \\ & \wedge & \forall p.\, \mathtt{ts}(p) = r/4 \Rightarrow \mathtt{commit}(\mathtt{coord}(r/4)). \end{array}$$

The invariant is a case split characterizing the states in which processes can safely decide. A process decides when there is a majority of processes agreeing on a proposal with timestamps more recent than t. The additional clauses are required to make the invariant inductive and to relate it to the specification. For instance, if a process is `ready` then its vote is v and it agrees with the majority. Also the decision of any process that has decided is v.

Methodology. To prove safety properties we implement a standard verification conditions generator for PSYNC programs. For the round R, the generator builds a $\mathbb{CL}$ formula corresponding to the transitions relation as follows. Let s, s' be the primed and unprimed function used to represent the global state of the system. The transition relation associated with a local `send`, resp. `update`, of a round R is $\mathtt{send}_R(s(p), m)$ resp. $\mathtt{update}_R(m, s(p), s'(p))$.

Then the transition relation of round R, $TR_R(s, s')$ is

$$\begin{array}{lll} & \forall p. & \mathtt{send}_R(s(p), m_s(p)) \\ \wedge & \forall p, q, t. & (t, q) \in m_u(p) \Leftrightarrow (t, p) \in m_s(q) \wedge q \in \mathsf{HO}(p) \\ \wedge & \forall p. & \mathtt{update}_R(m_u(p), s(p), s'(p)) \ \wedge \ r' = r + 1. \end{array}$$

Safety. We generate verification conditions that imply partial correctness: (1) the invariant contains the initial state ($TR_{\mathtt{init}}(s) \Rightarrow Inv(s)$), (2) for any round R the invariant is inductive ($Inv(s) \wedge TR_R(s, s') \Rightarrow Inv(s')$), (3) the safety specification φ is implied by the invariant ($Inv(s) \Rightarrow \varphi(s)$).

Liveness. We also prove liveness properties, such as, every process eventually decides in *LastVoting*. Showing these properties typically requires ranking functions. However, in many cases we can simplify the proof. We show that there exists a fixed number of good rounds, i.e., rounds when the environment assumptions hold, such that after the execution of the last good round the program reaches a set of good states, e.g., processes have decided. To prove that the program makes progress after each good round the user provides additional invariants, expressing how a good round strengthens the safety invariant. For instance, in the *LastVoting* the program makes a decision if the formula (1) holds during one complete phase of the algorithms. An intermediate invariant specifies that between round `Collect` and `Candidate` the formula $\mathtt{commit}(\mathtt{coord}(r))$ holds on top of the safety invariant.

7. Evaluation

We have implemented PSYNC as an embedding in the SCALA programming language. The runtime of PSYNC is built on top of the NETTY framework [69]. For the transport layer, we use UDP. The serialization of messages uses the pickling library [60]. The implementation is available at `https://github.com/dzufferey/psync` under an Apache 2.0 license.

The set of processes executing a PSYNC program is specified in a configuration file. The runtime manages the interface between a PSYNC program $\mathcal{P}$ and the client application, and also the communication between the different processs running $\mathcal{P}$.

The execution of $\mathcal{P}$ is launched from a client application, using $\mathcal{P}$'s interface. More specifically, the interface contains callback methods provided by the application. Regarding termination, many consensus algorithms presented in the literature assume that processes continue executing the algorithm after a decision is taken, because not all processes decide simultaneously. For example, in a case of a network partition, some processes learn the decision value much later. To safely free the memory allocated by the runtime when a process decides, each process stores only the decision value in a log. In our experiments we implement a key-value store using *LastVoting* iteratively. Each process terminates an execution of *LastVoting* once it decides, but the process keeps a log of the most recent decisions. Any process detects the messages send by late processes, e.g., those that got disconnected or slowed down temporary, and send them the decision from the log (if present).

To achieve the good performances, the timeout is an important parameter determined empirically. Δ can easily be measured, i.e., latency and bandwidth. Θ is harder to measure but can easily be over-approximated. To decrease the reliance on an accurate timeout, we implemented several optimizations that allows the runtime to progress before the timeout occurs.

7.1 Implementing Algorithms in PSYNC

PSYNC can implement a wide variety of fault-tolerant distributed algorithms. Table 1 lists several implementations in PSYNC of algorithms that solve different agreement problems. For each algorithm, we indicate if it is designed for a synchronous or an asynchronous network and if the presentation uses some form of rounds. Many asynchronous algorithms are tagging messages with information that implicitly structures programs in rounds (not necessarily executing in lockstep). However, even when the original algorithm presentation is event-driven, they can be encoded in PSYNC.

The first three algorithms focus on the traditional consensus problem, the others are weaker agreement problem. Ben-Or algorithm [11] solves binary consensus and almost surely terminates. The k-set agreement [25] is a weaker version of consensus that

Algorithm implemented in PSYNC	LOC	Use rounds	Async.
Last voting [23]	89	✓	✓
One third rule [23]	50	✓	✓
Flood min consensus [55]	22	✓	×
Ben-Or randomized consensus [11]	58	✓	✓
k-set agreement [25]	39	✓	✓
k-set agreement early stopping [65]	30	✓	×
Lattice agreement [36]	30	×	✓
ϵ-agreement [50]	49	✓	✓
Two phases commit [41]	53	✓	×
Eager reliable broadcast [17]	27	×	×

Table 1: Fault-tolerant algorithms implemented in PSYNC

Paxos implemented in	LOC	Executable	Verification
PSYNC	89	✓	semi-automated
DistAlgo	43	✓	×
Distal	157	✓	×
Overlog	107	✓	×
TLA+	53	×	mechanized
IO Automata	142	×	mechanized
EventML	1729N	✓	mechanized
Verdi (Raft algorithm)	520	✓	mechanized
Bloom	224	✓	×

Table 2: Comparison of the code size and verification of Paxos in different languages. For EventML, Schiper et. al. [67] report the number of AST nodes.

allows processes to decide on k-different values. The generalized lattice agreement [36] asks processes to choose values in a lattice, such that these values form a chain. ϵ-agreement is a form of consensus over $\mathbb{R}$ in which all the decision values lie in an interval of size ϵ. Two phases commit is a degenerate version of the binary consensus where the decision value $true$ is allowed only if all processes propose $true$. Reliable broadcast guarantees that if a correct process delivers a value, then all correct processes deliver that value.

We compare PSYNC against other high-level languages for distributed algorithms. Table 2 shows a comparison between different implementations of Paxos in different programming and specification languages. For PSYNC we used *LastVoting*. For each implementation we count the number of lines of code without comments or blank lines. Also we focus on the algorithm itself and remove boilerplates like include statements.

We compare against the following languages: DistAlgo [54] is a programming language for distributed algorithm that uses incrementalization to compile a high-level specification into Python code. Distal [14] is designed to express fault-tolerant distributed algorithms in a pseudo-code-like manner. It is built as a library on top of Scala. Bloom [5] is a programming language based on a monotonic logic for building consistent distributed systems. Overlog [4] is a logic programming language for distributed systems inspired by Datalog. EventML [59] is a programming and verification language for distributed algorithms connected to the Nuprl interactive theorem prover [3]. Verdi [75] is a Coq framework for implementing and proving distributed systems correct. TLA+ [52] is a logic-based specification language designed to describe concurrent and distributed systems. IO Automata [56] is a specification language with automata theoretic foundations to describe asynchronous concurrent and distributed systems. Except for TLA+ which can encode both synchronous and asynchronous programs, and Verdi that starts with a synchronous model and transforms it into an asynchronous one, the other languages have an asynchronous semantics. Currently only mechanized proofs in Nuprl or Coq exist for Paxos, when implemented in EventML or Verdi.

Implementation	Source	Year	Throughput $\times$1000 req/s
Last Voting (Batching)		2015[4]	170
Egalitarian Paxos	[63]	2013	450
Paxos in Distal	[14]	2013	150
JPaxos / SPaxos	[13]	2012	75 / 300
Paxos for system builder	[6]	2008	40

Table 3: Performance of Paxos implementations with 3 processs

Limitations of the model PSYNC keeps an order between messages only if they are sent in different rounds. It is oblivious to the order in which messages arrive in one round. As a consequence, one cannot implement the runtime system of PSYNC in PSYNC itself. Also, for the moment we don't have an efficient way of composing PSYNC programs to create other programs. The composing round based models is a problem that currently receives attention.

7.2 Comparing PSYNC to Existing Paxos Implementations

We evaluate our PSYNC implementation of Paxos (*LastVoting* from Fig. 1) and compare it to existing Paxos implementations. We use *LastVoting* to order write requests in a simple key-value store. The submitted requests are collected into batches of about 300 requests, then *LastVoting* is used to make all processs agrees on the next set of writes. Adding batching to *LastVoting* requires only changing the type of the messages sent in each round, i.e., modifying only a few lines of code ($<$10) because batching does not interfere with the control structure of the algorithm.

Table 3 shows throughput numbers for the different implementations of Paxos we considered. All the algorithms try to maximize the throughput when dealing with requests of small sizes. However, the exact settings is sightly different for every experiment. Due to the difficulty of processting published results, we report the published numbers. We run PSYNC on three servers with Intel Xeon X5460 cpu and 8 GB ram [4], running Linux 2.6.32 and the JRE 1.8. We also ran experiments incorporating crashes. The throughput of the system roughly halves after one crash. The point of this comparison is to show that the overhead of the runtime to implement the round structure is acceptable and does not preclude PSYNC adoption. We believe that the benefits of PSYNC, i.e., an intuitive semantics, simple control structures, and the ability to use automated verification tools, make it a compelling language.

PSYNC and Distal both are based on SCALA. JPaxos and SPaxos [13] are written in Java. JPaxos is Java implementation of Paxos and SPaxos is an improved algorithm to achieve higher-throughput when the coordinator is CPU bound. Egalitarian Paxos [63] is implemented in Go and improves over Paxos by processing independent requests in parallel. Paxos for system builder [6] is an implementation of Paxos in C. To achieve high throughput, all the implementations use batching.

7.3 Verification in PSYNC

We implemented a verification conditions generator for PSYNC, based on the logic $\mathbb{CL}$. To compute the verification conditions the tool computes, at compile time, (1) the transition relation of the program and (2) transforms the program assertions given in the SCALA language in $\mathbb{CL}$ formulas.

We implemented the semi-decision procedure for $\mathbb{CL}$ [33] on top of the SMT solver Z3 [64] and using the VC generator we verified the PSYNC programs One third rule [23] and *LastVoting*. Their specification and invariants are provided by the user. We used the invariants from [33]. For the One Third Rule, we need for 4 invariants (23 LOCs), 27 VCs are generated, solved in 5s. For

[4] We use 5 years old machines which makes the comparison with older results relevant

the *LastVoting*, we need for 8 invariants (35 LOCs), 45 VCs are generated, solved in 16s.

The verification of programs solving weaker agreement problem requires reasoning about sets of data. For example, k-set consensus needs to reason about the cardinality of the set of decision values. $\mathbb{CL}$ supports only reasoning about sets of processes. We are working on extending the scope of our implementation to verify a larger class of examples.

8. Related Work

In this section we compare PSYNC to the related work from a programming language and verification perspective.

Formalizations of distributed algorithms. Distributed algorithms are typically defined in English or pseudo-code [42] using different computational models. Synchronous and asynchronous models are the most frequent ones. Synchrony allows solving a larger class of problems, while asynchrony is close to the network behavior. Finding an uniform model is still an open problem in the distributed algorithms community. Multiple models that abstract uniformly faults and (a)synchrony have been introduced [2, 12, 23, 34, 40, 46, 66, 74]. We have chosen the HO-model [23] because of its simplicity. It handles asynchrony, host and network failures uniformly.

In the classic setting, distributed systems are formalized using the π-calculus [61, 62], CSP [45], and I/O-automata [56]. These formalisms are used to give a formal semantics to message-passing systems and to analyze them, but not as programming languages.

The Actor model [44] is probably the most successful high-level programming abstraction for message-passing systems. Actors are either built-in languages, e.g., Erlang [8], or supported through libraries, e.g., Scala [43]. Erlang via the OTP library [70] has support to handle faults in distributed systems. Faults are handled using a supervision hierarchy: when a replica fails its superior in the hierarchy is notified and takes action. PSYNC allows reasoning about faults when processes are not organized in a strict hierarchy.

Several domain specific languages for distributed algorithms have been developed [4, 5, 9, 14, 49, 54, 59]. Languages like Meld [9], Overlog [4], and Bloom [5], which are based on Datalog, do not have a formal operational semantics, and do not support automated verification. The closest domain specific languages to PSYNC are Mace [49], DistAlgo [54] and Distal [14]. However, they all have an asynchronous semantics and lack high-level programming construct to reason explicitly about faults.

Verification. The verification of parametric systems is in general undecidable [7]. Therefore, mainly bug finding tools are developed. They are based on state-space exploration up to a bounded, generally small, number of processes or messages [32, 39, 49, 71, 72], or uses specialized abstractions [47]. Static analysis techniques [29] are used to prove only simple properties, such as type errors or dead code detection, not for complex functional properties.

Recently a few programming languages were designed with build-in support for verification. Mace [49] has an integrated model checker which cannot prove total functional correctness. The most successful programming languages from a verification perspective are EventML [59] and Verdi [75]. EventML is a functional programming language and Verdi is a Coq framework for implementing and proving correct distributed algorithms. Verdi starts with a synchronous implementation and progressively transforms it using refinement into an asynchronous fault-tolerant one. The correctness of the implemented programs is mechanically proved using theorem provers: EventML uses Nuprl and Verdi uses Coq. These two languages have a small trusted base but reduced performances. For example, the throughputs of PSYNC implementations are several orders of magnitude faster than corresponding ones in Verdi. Moreover PSYNC's verifier is designed for automated verification, currently based on SMT solvers.

In the algorithms community specification languages like +Cal [53] and TLA+ [52] are used to write formal specification of distributed algorithm and to model check or to mechanically prove them. However, these specifications are not executable.

Finally, the exploration of synchronous behavior of asynchronous systems for verification purposes has been investigated in [10, 28]. Our approach starts with a (partially) synchronous abstraction rather than retrofitting synchrony in existing asynchronous systems. The approach in [10, 28] makes the verification more complex without offering any guarantees about its applicability.

9. Conclusion

We have presented PSYNC a domain specific language for fault-tolerant systems that strikes a balance between high-level constructs, performance, and automated verification. PSYNC offers a simple lockstep semantics that is indistinguishable from its runtime asynchronous executions. We have implemented a prototype runtime for PSYNC for partial synchronous networks and shown that it performs within a constant factor from highly optimized low-level implementations. For future work we intend to enlarge the application domain of PSYNC and to raise the automation level of the verification engine by developing static analysis that generate inductive invariants. We plan to generalize the runtime to cover more fault-models, such a Byzantine faults.

Acknowledgments

We thank Josef Widder for the discussion about modeling distributed systems, Shaz Qadeer for his support and suggestions for improving the paper, and the anonymous reviewers for their helpful feedback. Thomas A. Henzinger was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award). Damien Zufferey was supported by DARPA (Grants FA8650-11-C-7192 and FA8650-15-C-7564) and NSF (Grant CCF-1138967).

References

[1] P. A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *FMSD*, 25(1):39–65, 2004.

[2] Y. Afek and E. Gafni. Asynchrony from synchrony. In D. Frey, M. Raynal, S. Sarkar, R. K. Shyamasundar, and P. Sinha, editors, *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, pages 225–239, 2013.

[3] S. F. Allen, R. L. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The nuprl open logical environment. In D. A. McAllester, editor, *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, pages 170–176, 2000.

[4] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I do declare: consensus in a logic language. *Operating Systems Review*, 43 (4):25–30, 2009.

[5] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[6] Y. Amir and J. Kirsch. Paxos for system builders: An overview. In *Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008), Yorktown, NY, September 2008*, 2008.

[7] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[8] J. L. Armstrong. The development of erlang. In S. L. P. Jones, M. Tofte, and A. M. Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, pages 196–203. ACM, 1997.

[9] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. Campbell. A language for large ensembles of independently executing nodes. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, pages 265–280, 2009.

[10] S. Basu, T. Bultan, and M. Ouederni. Synchronizability for verification of asynchronously communicating systems. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 56–71, 2012.

[11] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC*, pages 27–30. ACM, 1983.

[12] M. Biely, B. Charron-Bost, A. Gaillard, M. Hutle, A. Schiper, and J. Widder. Tolerating corrupted communication. In *PODC*, pages 244–253, 2007.

[13] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *IEEE 31st Symposium on Reliable Distributed Systems, SRDS 2012, Irvine, CA, USA, October 8-11, 2012*, pages 111–120, 2012.

[14] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–8, 2013.

[15] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

[16] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

[17] C. Cachin, R. Guerraoui, and L. Rodrigues. Reliable broadcast. In *Introduction to Reliable and Secure Distributed Programming*, pages 73–135. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-15259-7.

[18] M. Castro and B. Liskov. Practical byzantine fault tolerance. In M. I. Seltzer and P. J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. .

[19] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[20] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5.

[21] M. Chaouch-Saad, B. Charron-Bost, and S. Merz. A reduction theorem for the verification of round-based distributed algorithms. In *RP*, volume 5797 of *LNCS*, pages 93–106, 2009.

[22] B. Charron-Bost and S. Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.

[23] B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1): 49–71, 2009.

[24] B. Charron-Bost, H. Debrat, and S. Merz. Formal verification of consensus algorithms tolerating malicious faults. In *SSS*, pages 120–134. Springer, 2011. ISBN 978-3-642-24549-7.

[25] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132 – 158, 1993. ISSN 0890-5401.

[26] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In J. Rexford and E. G. Sirer, editors, *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 153–168. USENIX Association, 2009.

[27] H. Debrat and S. Merz. Verifying fault-tolerant distributed algorithms in the heard-of model. *Archive of Formal Proofs*, 2012.

[28] A. Desai, P. Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 709–725, 2014.

[29] Dialyzer. `http://www.erlang.org/doc/man/dialyzer.html`.

[30] D. Dolev and E. N. Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In T. Masuzawa and S. Tixeuil, editors, *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, SSS 2007, Paris, France, November 14-16, 2007, Proceedings*, pages 234–252. Springer, 2007. .

[31] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. In *Foundations of Computer Science, 1983, 24th Annual Symposium on*, pages 393–402, 1983. .

[32] E. D'Osualdo, J. Kochems, and C. L. Ong. Automatic verification of erlang-style concurrency. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 454–476, 2013.

[33] C. Dragoi, T. A. Henzinger, H. Veith, J. Widder, and D. Zufferey. A logic-based framework for verifying consensus algorithms. In K. L. McMillan and X. Rival, editors, *VMCAI*, pages 161–181. Springer, 2014.

[34] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, Apr. 1988.

[35] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.

[36] J. M. Falerio, S. K. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Generalized lattice agreement. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 125–134, 2012.

[37] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010. .

[38] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[39] L. Fredlund and H. Svensson. Mcerlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 125–136, 2007.

[40] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In B. A. Coan and Y. Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 143–152, 1998.

[41] J. Gray. Notes on data base operating systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin Heidelberg, 1978. ISBN 978-3-540-08755-7.

[42] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540288457.

[43] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009. .

[44] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd In-*

ternational Joint Conference on Artificial Intelligence. Standford, CA, August 1973, pages 235–245, 1973.

[45] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[46] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *DSN*, pages 92–101, 2007.

[47] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD*, pages 201–209, 2013.

[48] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 245–256. IEEE, 2011.

[49] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distributed systems. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 179–188, 2007.

[50] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

[51] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16 (2):133–169, May 1998. ISSN 0734-2071.

[52] L. Lamport. Distributed algorithms in TLA (abstract). In G. Neiger, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. ACM, 2000.

[53] L. Lamport. The pluscal algorithm language. In *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, pages 36–60, 2009.

[54] Y. A. Liu, S. D. Stoller, B. Lin, and M. Gorbovitski. From clarity to efficiency for distributed algorithms. In G. T. Leavens and M. B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 395–410, 2012.

[55] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[56] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In F. B. Schneider, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 137–151, 1987.

[57] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[58] O. Marić, C. Sprenger, and D. Basin. Consensus refined. In J. Karlsson and Y. Amir, editors, *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

[59] V. R. Mark Bickford, Robert L. Constable. Logic of events, a framework to reason about distributed systems. In *2012 Languages for Distributed Algorithms Workshop*, Philadelphia, PA, 2012.

[60] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *OOPSLA*, pages 183–202, 2013.

[61] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[62] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[63] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8.

[64] L. Moura and N. Bjorner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78799-0.

[65] P. Parvedy, M. Raynal, and C. Travers. Early-stopping k-set agreement in synchronous systems prone to any number of process crashes. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 3606 of *Lecture Notes in Computer Science*, pages 49–58. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28126-9.

[66] N. Santoro and P. Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theor. Comput. Sci.*, 384(2-3):232–249, 2007.

[67] N. Schiper, V. Rahli, R. van Renesse, M. Bickford, and R. L. Constable. Developing correctly replicated databases using formal tools. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 395–406, 2014.

[68] P. Schnoebelen. Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets. In *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*, pages 616–628, 2010.

[69] The Netty project. `http://netty.io/`.

[70] S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.

[71] T. Tsuchiya and A. Schiper. Model checking of consensus algorithms. In *26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, October 10-12, 2007*, pages 137–148. IEEE Computer Society, 2007.

[72] T. Tsuchiya and A. Schiper. Using bounded model checking to verify consensus algorithms. In G. Taubenfeld, editor, *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 466–480, 2008.

[73] T. Tsuchiya and A. Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6):341–358, 2011.

[74] J. Widder and U. Schmid. The Theta-Model: Achieving synchrony without clocks. *Distributed Computing*, 22(1):29–47, Apr. 2009.

[75] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In D. Grove and S. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015.