Aspect-Oriented Language for Reactive Distributed Applications at the Edge

Ivan Kuraj
CSAIL, MIT, US
ivanko@csail.mit.edu

Armando Solar-Lezama
CSAIL, MIT, US
asolar@csail.mit.edu

Abstract

This paper presents EdgeC, a new language for programming reactive distributed applications. It enables separation of concerns between expressing behavior and controlling distributed aspects, inspired by aspect-oriented language design. In EdgeC, developers express functionality with sequential behaviors, and data allocation, reactivity, consistency, and underlying network with orthogonal specifications. Through such separation, EdgeC allows developers to change functionality and control the shape of resulting distributed behaviors without cross-cutting code, simplifying deployment to the edge. Developers can reason about and test their applications as sequential executions, whilst EdgeC automatically synthesizes low-level distributed code. It handles, with the help of the EdgeC run-time, allocation, communication, concurrency, and coordination, across the specified, potentially non-uniform, network model. We introduce the main features of EdgeC, present the new compiler design, its prototype implementation, the resulting performance, and discuss the potential of the approach for simplifying development of reactive applications over nonuniform networks and achieving performance gains, compared to existing approaches.

ACM Reference Format:
Ivan Kuraj and Armando Solar-Lezama. 2020. Aspect-Oriented Language for Reactive Distributed Applications at the Edge. In *3rd International Workshop on Edge Systems, Analytics and Networking (EdgeSys '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3378679.3394531

1 Introduction

Edge computing poses new challenges for reactive distributed applications, which go beyond black-box cloud-centric systems [8, 9]. With traditional programming methodologies, design decisions about distributed aspects, such as data distribution and reactivity need to be *woven together with the application logic*. As a result, implementations become complex even when the underlying logic is conceptually simple, and the ability to *explore different design choices is limited* because small changes to how data is distributed or how communication is orchestrated require cutting through multiple layers of code. The design choices might involve different network models, data allocation schemes, consistency and reactivity requirements. In such cases, the program developers need to write full implementations, significantly different for every combination of design decisions, making it *difficult to search for an optimal overall design, in spite of core logic being fixed.* The programming complexity further exacerbates when the system needs to operate in *heterogeneous environments*, which involve nodes of various computing capabilities and non-uniform networks.

This paper presents EdgeC[1], a framework that aims to simplify prototyping event-driven reactive distributed programs, amenable for the edge, including non-uniform deployments. EdgeC relies on the insight that distributed systems can remain *sufficiently specified, by separating the application logic and distributed aspects.* Developers write a set of operations, which specify the application logic, together with orthogonal specifications for distributed requirements, *data and computation allocation, reactivity, consistency, and networking concerns.* The system generates low-level implementation of a distributed system which allows invoking the given operations concurrently and reactively, arbitrarily distributed across the system, respecting the sequential semantics of operations according to chosen consistency (akin to distributed transactions). Specifications of behaviors, data/computation allocation, consistency, and network model, can be tweaked and changed separately, while EdgeC handles low-level code automatically. EdgeC thus allows easier *design space exploration of distributed programs*, offloading complex cross-cutting reasoning from developers. This paper presents the following contributions: 1) a *novel framework design, which synergizes program analysis and synthesis*, with a run-time, for developing optimized implementations that satisfy specifications across cross-cutting and inter-dependent concerns, applicable in new domains; 2) a prototype implementation of EdgeC and its performance.

This paper focuses on the language and compiler design, and the programming model in general, in terms of achieving expressiveness through orthogonal specifications for behaviors, and allocation, reactivity and data consistency. In turn, EdgeC assumes distributed nodes operate in reliable and trusted environments and does not address aspects related to security and failure-tolerance. We discuss these aspects later as potential avenues for future work. The goal of this paper is to propose and examine the potential of the programming model in terms of expressiveness of behaviors and performance, without engineering a general and fully optimized full-fledged system.

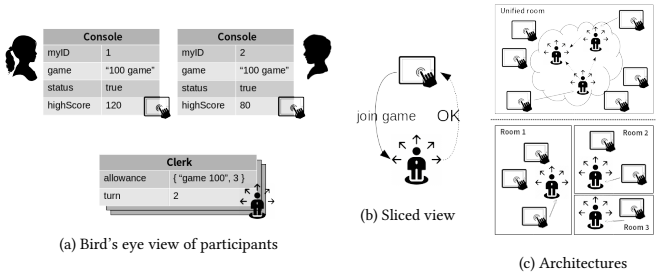
2 Overview

We demonstrate the main ideas behind EdgeC through a tutorial implementing the "100 game" (analyzed as a motivating example in [8]). In this simple game, players alternately add a chosen number to the current sum, and the first one to reach 100 wins. Fig. 1 shows a high-level view of the architecture of the application. The goal is to prototype distributed version(s) of the game, with different

EdgeSys '20, April 27, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-7132-2/20/04.
https://doi.org/10.1145/3378679.3394531

[1] EdgeC is an anagram of initial letters from "**E**vent-driven **d**istributed **g**lobal-view **c**onsistent **e**xecutions"



(a) Bird's eye view of participants

(b) Sliced view

(c) Architectures

Figure 1. Application overview, its logical "sliced view" and architectures it can capture

requirements in mind. EdgeC allows the developer to start writing operations as sequential code (without distribution in mind, viewing it as "sequential slice" Fig. 1b), and orthogonally define distributed aspects that define distributed computation shape, be it standard client-server or more specialized architecture (e.g. ones shown in Fig. 1c). Through a series of requirements on functionality and distributed concerns, we show how developers use EdgeC to incrementally explore new behaviors and/or distributed specifications, at each point producing *fully functional reactive application*. EdgeC *synthesizes Scala code and provides a JVM-based runtime*, which is then deployed on an interconnected set of machines (see § 3).

Distributed interactive behaviors in EdgeC. The main functionality is making a game move, per player turn, which adds a given number to the overall score. Fig. 2 shows EdgeC code to implement this behavior. Syntax of the language is based on Scala. Scenarios encapsulate a particular set of distributed behaviors (effectively a sub-system) and are parametrized by node instances that represent physical machines in the system. With the **node** declaration (line 1), developers declares two types of nodes, Console, which encapsulates console id (passed as a parameter when a node is created), and Server. The Game scenario defines the behavior over a given server and a set of console nodes (line 3).

In EdgeC, the core functionality is simply given with sequential code and distributed aspects are provided on top (through annotations). Developers define data and the play function, which takes a parameter and checks if it's the current player's turn: if yes, the score is increased and other variables are updated. The code for play is oblivious to any distributed aspects of the computation; it can be reasoned about and tested as any other sequential function (e.g. simple unit testing).

Distributing behaviors. Specifications of distributed aspects characterize how sequential behaviors are mapped onto the distributed system at hand. Firstly, given node instances, here server and consoles (line 3), each variable is annotated to specify allocation to a node. The annotation @loc(n) designates the declaration at hand, data or function, is allocated at instance n. With lines 4-7, developers allocate game variables on the server, and myId for every console in consoles (each console owns a copy), while notify can be executed only on a Console node. With the construct forall, developers allocate one myId for each client node and initialize it to id of the console (line 6).

In EdgeC, all function invocations (behaviors) need to resolve to specific node instances which determine the actual distributed

```
1  node Server; node Console(cId: Int)
2
3  scenario Game(server: Server, consoles: Set[Console]) {
4   @loc(server) { var turn: Int = 0; var score: Int = 0;
5    var last: Int = −1 }
6   @forAll(c in consoles) { @loc(c) var myId: Int = c.cId;
7    @loc(c) def notify(r: String) = println(r) }
8
9   @resolve[Console]
10  def play(num: Int) = {
11   if (turn == myId) {
12    score += num; last = myId
13    turn = (turn+1)%consoles.size
14    return true
15   } else return false }
16
17  trigger PlayGame[Console](num: Int)
18
19  triggering(anyOf(consoles) { (c: Console) =>
20   when (PlayGame(num)) { bind(c)(play(num)) }
21   when (onChange(turn)) { notify(``Score is: " + score) }
22   when (onTrue(score>100)) { notify(``Win: " + last) }
23  })
24 }
```

Figure 2. Distributed reactive game

execution at run time. EdgeC does this at the place of invocation, at compile time, as explained later. While game data is unambiguously bound to the single server (through @loc), developers annotate play with resolve[Console] which means behavior needs to bind a console instance to be invoked; this is due to accessing myId. (EdgeC allows means of binding different consoles to different variables, not shown here.) This in turn means: 1) access to myId resolves to an access on the bound console; 2) play accesses data spread across different nodes, i.e. server and console. EdgeC automatically splits behaviors into chunks (based on data allocation), analyzes chunk inter-dependencies, and performs the necessary communication needed for distributed execution. Thus, every behavior might be executed chunk-by-chunk, across multiple different nodes; here, console sends request to the server (sending data num and myId), the server evaluates the whole body, and a response is returned. (EdgeC splits behaviors according to a given network model; by default, it minimizes communication rounds in a uniform model.)

Invoking behaviors. Having defined the behaviors and data allocation, the next step is to define when behaviors should execute, i.e. reactivity. With **trigger**, developers define external events (e.g. interaction with the system). PlayGame is a trigger that can occur at Console instances; it represents an event that carries num. EdgeC generates API stubs for injecting events with parameters (so this can be done e.g. from a UI). The construct **triggering** invokes behaviors in response to triggers. The core of this construct is a **when** e {b} statement, which indicates that b should be executed whenever trigger e occurred. In this example, since the event PlayGame and behavior play need to bind to some console, developers first use anyOf(consoles) to quantify over all nodes c in the set consoles, and use it to bind the invocation of play. Let's consider line 20: whenever PlayGame event is fired, on any quantified console c (bound implicitly), the system starts the play behavior in the system. The caller, who injected PlayGame, then waits for the response.

Resulting distributed behaviors. EdgeC compiles this program, accounting the aspect of allocation, into a message-passing implementation which offers APIs to instantiate nodes, scenarios and inject events. By default, EdgeC uses the Akka framework [1]. Given Akka actors are obtained, developers construct nodes from actors and start the application with:

```
def init(sa: ActorRef, cas: List[ActorRef]) {
 val cs = cas.zipWithIndex.map({ case (a,i) => Console(a,i) }).toSet
 Game(Server(sa), cs).start() }
```

This creates a console node for each actor in the given list, with their ordinal as cId argument. Interaction is done by injecting events as actor messages. On the console node, developers can inject PlayGame with:

```
val console: ActorRef = ...; console ! PlayGame(5)
```

The system performs a client-server communication pattern, by communicating num to the server, executing the body of play on the node specified by sa, retrieves response, and passes it to the caller (asynchronously through a message). In EdgeC developers can implement the behavior in different patterns, by either changing the given behavior code or orthogonal specifications.

Reactivity. Unlike traditional approaches, handling reactivity is done through a separate specification, without cross-cutting code. When the game is over, i.e. score reaches 100, developers notify players by simply binding a call to notify to a trigger onTrue; with onTrue(c), whenever the condition c (on arbitrary allocated data) goes from false to true, the trigger is fires (Fig. 2, line 22). EdgeC allows implicit resolution – the quantified console c is implicitly bound to the call. This achieves the expected behavior of invoking notify on all consoles when the game is over. EdgeC makes sure that, at any point in time, if the trigger condition becomes true, the corresponding behavior is invoked. It analyzes the code and checks for all possible places the trigger can fire: in cases it can be statically determined, a call to the behavior is inserted or omitted, otherwise a run time check is emitted. (EdgeC fires at most one declared bound trigger per operation invocation; defined in § 4.) In this case, EdgeC automatically determines a check needs to be inserted after invoking play.

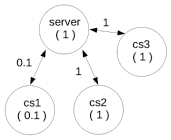
However, imagine developers introduced another operation besides play (and bound it to nodes), to reset the game:

```
def reset() { score = 0; turn = 0; last = −1 }
```

Then, EdgeC does not insert any checks after reset, as analysis confirms score cannot become 100 after executing reset.

Now, imagine developers want also to notify players of the new score, whenever it changes. In this case, the developers bind appropriate notify call to the onChange(e) trigger, which triggers every time the expression e changes (used in Fig. 2, line 21). EdgeC automatically injects approapriate calls, but also optimizes communication: namely, after the score reaches 100 both triggers are fired but the server sends only one message to consoles. (A naive event handling would fire two behaviors through two separate messages from the server to consoles, i.e. 2n messages for n consoles.) EdgeC statically analyzes control flow to minimize communication; here, it consolidates messages and emits an additional check (and receive handler on consoles), to cover this case.

Handling consistency. When developing applications with multiple distributed behaviors occurring at differences places in the system, developers must often coordinate messages to avoid inconsistent results. Unlike uniform transactional systems, EdgeC handles consistency at different nodes by employing different low-level coordination protocols. It analyzes program semantics, places of invocation, as well as network model to try to optimize and avoid unnecessary coordination. Our prototype supports strong and weak consistency modes (given as a flag to our prototype compiler; not shown).




```
val g = Graph[Node]()
g.node(server, 1.0)
val edges = for (c <− consoles) yield {
  g.node(c, 1.0)
  g.edge(c, server, 1.0) } /* bidirectional */
g.node(cs1, 0.1); g.edge(cs1, server, 0.1)
initNetwork(g)
```

Figure 3. Network model specifications

Under weak consistency, EdgeC simply executes all behaviors when they get invoked as a sequence of message passing communication across the system (i.e. more specifically a graph, based on data dependencies across nodes), without coordination. For strong consistency, EdgeC analyzes any two behaviors in the system for potential conflicts that could violate serializability. (EdgeC performs cycle graph conflict analysis based on prior work [13].) In the running example, since the game data is allocated on the server node, strong consistency is preserved if all executions (data modifications) on server and notification behaviors resulting from those executions on the consoles are observed in the same order on all nodes. Under Akka implementation, which uses TCP/IP, this is automatically satisfied (as message ordering between two nodes is preserved). However, EdgeC supports backends with weaker assumptions (e.g. UDP or MQTT under low QoS) and implements such ordering automatically.

EdgeC performs pattern matching on the conflicting behaviors and underlying network and applies optimizations, such as ordering, if possible. In case such optimizations are not possible, EdgeC emits two-phase consistency protocol to ensure strong consistency. In the given example, imagine developers split data between two servers, allocate turn on a different server node server2, and redefine reset:

```
scenario Game(server: Server, server2: Server, ...) {
 @loc(server2) var turn: Int = −1
 def reset() { /* same as before */ }
```

EdgeC emits two-phase commit for all invocations of play and resetTurn (regardless of places of triggering) to preserve same ordering of observing two operations on server and server2, due to potential conflicts.

EdgeC guarantees the chosen level of consistency, however, unlike traditional data management systems, it optimizes distributed behaviors based on network topology and operation semantics, making it more amenable for the edge. The optimization is sound, but incomplete, as EdgeC might fail to recognize a case where optimization is valid and emit a costly consistency protocol.

Adapting implementations with network models. EdgeC allows developers to model the network, specifying a cost of communication as well as execution on individual nodes. Developers specify custom models with **network**. If left unspecified, EdgeC assumes the default uniform network model with costs of execution and communication equal to 1 across all nodes and edges. Network

takes a directed weighted graph of the network as an argument, as shown in Fig. 3[2].

In the current example, without network model specification, for play, EdgeC generates behavior following the client-server pattern, as mentioned. If the depicted network model is specified, however, since cs1 has lower cost of computing and connectivity (0.1, as depicted on the left), EdgeC will allocate computation on the node, incurring more communication rounds, but still overall lower cost, according to the given model.

Replication and aliasing. EdgeC supports replication as experimental feature in the prototype, limited to certain forms of scenarios (programs outside the supported class will fail to compile). In the running example, developers might decide to replicate data across multiple server instances (as shown in Fig. 1). Replication is supported in EdgeC prototype by providing a special annotation:

```
1 @replicated(3) node Server
2 def resolveReplica(c: Console): Int = ...
3 @resolver(resolveReplica) def play( /* as before */ )
```

Here, EdgeC replicates server, implicitly, across 3 instances. It resolves accesses to replicated variables, e.g. in play, based on an externally defined function resolveReplica. Modifying server variables under strong consistency now involves 2PC across all server replicas. Replication is supported on the level of a node type; we plan to extend it to apply to specified node groups and specific variables.

Peer-to-peer. Let's assume developers want to fit their application into a p2p model. In this setting, there are only console nodes, while every console has its designated console node acting as the server. This can be achieved by declaring another scenario, similarly as before, but specifying the server to be one of the console nodes.

```
1 scenario P2PGame(server: Console, cs: Set[Console]) {
2  require(cs.contains(server))
3  ... /* rest is as before */ }
```

Without any other changes to our running example, EdgeC recognizes the precondition to conclude that the server is also one of the console nodes. The main difference with respect to the previous case is that a console node now stores all the game variables (in addition to myId) and play is instantiated for two cases: 1) originating console is also the server and play incurs no communication (as the behavior executes locally on the single console); 2) originating console is not the server, in which case the behaviors are the same as discussed before.

3 EdgeC compilation

EdgeC contains two parts: the compiler and the runtime. EdgeC employs a novel compiler design which incorporates multiple program analyses. The compiler compiles EdgeC programs to Java bytecode, while the runtime implements communication primitives and consistency protocols (using Akka [1]). The runtime exposes APIs which are called by the generated code.

Here, we focus on the main tasks of the compiler. The overview of the compilation is given in Fig. 4. The core parts are the behavior graph synthesizer and splicer, which analyze behaviors and specifications, and incrementally splice appropriate code into the distributed implementation. The compilation process includes helper

[2]Our prototype currently does not use reflection, thus graphs have to be static instances, known at compile time.

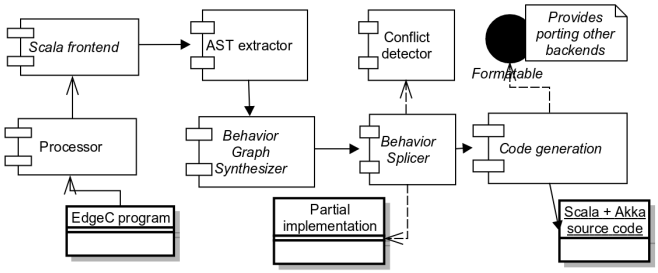


Figure 4. EdgeC compiler

components: EdgeC preprocessor; AST extractor, which extracts behaviors and EdgeC specifications; implicit resolver, which binds nodes at the places of behavior invocations; and code generator, which transforms the internal representation of programs into bytecode. Our prototype compiler is implemented as a Scala compiler plugin.

High-level compilation loop. The compilation process loops over all triggers, identifying all the places they might trigger, instantiates bound behaviors, and analyzes the current set of behaviors for consistency levels under which the behaviors need to be executed. Effectively, this is done until a fixed point is reached, i.e. no new changes are made to the resulting implementation.

Behavior splitting and allocation. Given an operation invocation and its starting node, a behavior graph designates the shape of a distributed computation: its nodes represent executions of operation chunks on particular nodes, while edges represent data or control-flow dependencies between chunks. For each invoked behavior, the process splits the operation and assigns individual chunks to be executed on particular nodes, producing a behavior graph. This is done with minimizing the overall cost of mapping (usually collocating data with computation). A behavior graph effectively encodes an execution graph: following the topological sort of such a graph, the system can execute the operation, incrementally distributed across the system, while propagating data dependencies according to its edges. (For example, a graph of play in the running example, reflects the simple client-server model; it has three nodes, with edges encoding data dependencies for the request and response.) A set of behavior graphs serves as intermediate representation of the final implementation and it captures enough information to allow emitting low-level code (and allow optimizations based on the given network model and interactions with other behaviors, e.g. the aforementioned message consolidation).

Trigger splicing. Trigger splicing identifies the places where triggers get activated (and operations invoked). They are can be at the originating nodes, in case of external triggers, or inside existing behaviors that enable conditional triggers during execution (e.g. for onChange(score), it's on the server, where play modifies score). EdgeC analyzes the current set of behavior graphs to discover possible places where these might get enabled and emits necessary run time checks.

Consistency analysis. EdgeC, when parametrized with strong consistency, checks for conflicts by analyzing each pair of behavior graphs. It employs an analysis based on interference conflict analysis, which allows detecting possible conflicting transactions statically [13]. We modify the original algorithm to handle distributed

Algorithm 1 Execution model as an interpreter

Require: start state s_0, condition triggers T_c

$s \leftarrow s_0, T \leftarrow \emptyset, E \leftarrow \emptyset$ ▷ state, triggers, execution graph
loop
  $T \leftarrow T \cup$ newEvents() ▷ include new external events
  $E \leftarrow E \cup$ chooseAndInstantiate(T, s) ▷ start new behaviors
  $el \leftarrow$ removeTop(E) ▷ next element in topological order
  if $el = node(e, t)$ **then** ▷ node case
    $res(el) \leftarrow$ eval(s, $env(el)$); $s \leftarrow s \cup res(el)$ ▷ evaluate chunk
    if el is last chunk in operation **then**
      $E \leftarrow E \cup$ instantiateAll(T_c, s) ▷ check condition triggers
    else if el is last chunk in condition trigger t **then**
      $T \leftarrow T \cup$ checkEnabled(t) ▷ new trigger enabled
  else $el = edge(n_1, n_2)$ ▷ edge case
    $env(n_2) \leftarrow env(n_2) \cup res(n_1)$ ▷ communicate dependencies

executions, accounting the network model (and single-threaded execution per each node, which our prototype currently employs).

Scalability of analyses. All the analyses run in polynomial time with respect to the program size, except the splitting phase which run in exponential time. (A faster, heuristic-based, allocation method is left for future work.)

4 Language Semantics

We formalize the semantics of EdgeC programs with an event-driven model of execution, enriched with specifications of distribution, reactive events, and consistency concerns. It defines the requirements of the execution model, while allowing different concrete scheduling, consistency models, and optimizations.

Event model semantics. We define dynamic semantics by interpretation [12]. The interpreter is given in Algorithm 1. Inputs to the interpreter include the starting state and a set of all condition triggers (such as onChange, which need to be checked during execution). The system maintains the current state (across all nodes), the set of active triggers T and an execution graph E. The execution graph represents all active behaviors, waiting to get executed. Whenever an operation is invoked, its behavior is instantiated and added to the main graph E. These behaviors might be either operation invocations or condition checking (which need to evaluate an expression over current state, e.g. onTrue(score>100)).

The interpreter initializes variables and starts looping. It first collects new external triggers to the set of active triggers T. Then, it chooses a subset of T to instantiate behaviors (chooseAndInstantiate), adding them to the main execution graph E. The instantiated set depends on the assumed consistency model: for strong consistency it only instantiates behaviors which do not conflict with any of the active behaviors; for weak, it instantiates all behaviors of enabled triggers. Next, the algorithm picks a node (to execute) or an edge (to perform communication step) from E: it picks either a non-visited node from E which has all incoming edges visited, or a non-visited edge which has the source node visited. (This is akin to topological order, but generalized to traverse edges as well.) If it visits a node, it executes it's behavior chunk, with it's environment ($env(el)$), for a result ($rel(el)$), and updates the state (s). If the node is the last non-visited node belonging to: 1) an operation invocation, it instantiates all condition triggers (to be checked, since some of them might trigger as a result of the current execution); 2) condition behavior, the evaluation result represents a Boolean which determines if the given trigger should fire, enabling new behaviors. If it visits an edge (from n_1 to n_2) it processes the communication step by updating environment for the chunk n_2.

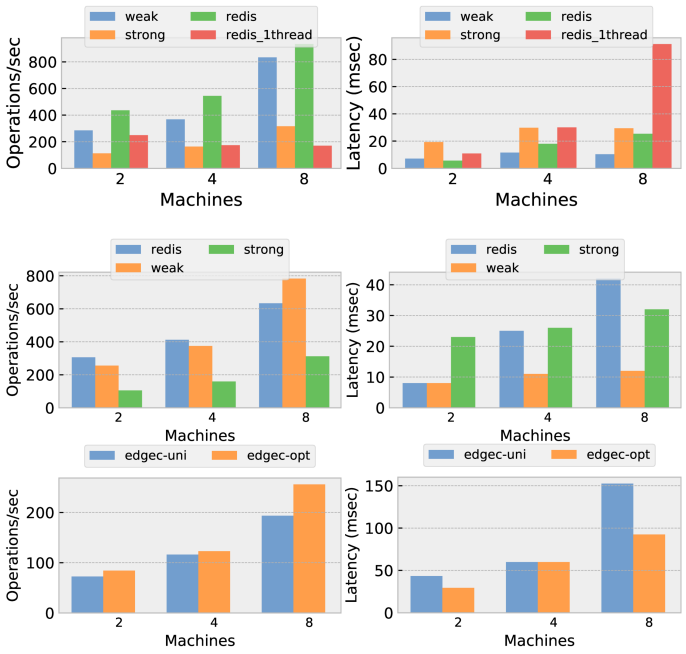


Figure 5. Performance evaluation

5 Evaluation

This section evaluates EdgeC prototype showing potential in performance gains due to implementation tweaking allowed by the expressiveness of the language. We evaluated the JVM implementation of EdgeC on OpenStack Compute Cluster using 8 machines (3GHz clock speed and 2GB of RAM).The benchmarks include: 1) the standard Retwis benchmark ([8]); 2) "reactive Retwis", with added reactive behaviors; 3) play from the running example, over a non-uniform network. The results are shown in Fig. 5. EdgeC finds all optimizations of behaviors for the given benchmarks; we thus believe it performs similarly to manual implementations that rely on the same optimizations.

Redis over uniform network. The first row of graphs shows the Retwis benchmark, specifically: EdgeC implementations with strong and no consistency (*strong* and *weak*); Redis-based (no consistency) implementation with and without concurrent handler (*redis* and *redis_1thread*). EdgeC's performance is comparable to that of standard Redis in the weak consistency model. Performance penalty in EdgeC occurs for strong consistency, as expected, as around 3/4 of operations require coordination to maintain consistency.

Reactive behaviors. We added a new operation to the Retwis benchmark for notifications of new post or likes. In Redis, clients poll after each operation to check for new notifications. The results

are shown in the second row of Fig. 5. The experiments confirm expected performance penalty due to polling in Redis; EdgeC exhibits better latency and throughput. The reason is direct splicing of triggers that avoids polling.

Non-uniform networks behaviors. We evaluated benefits of leveraging the network model in EdgeC, under a non-uniform network. We ran play operation from the running example, with the network model from § 2; we simulated this by adding delay of 15ms to both computation and message receives on "slow" nodes. The results are shown in the third row of Fig. 5. EdgeC outperforms the uniformly communicating system, in both the throughput and latency, since the adjustments in computation based on the given network avoids the unnecessary latency penalties. This becomes more significant with 8 nodes (with slowdown of around 25%).

5.1 Comparison to other programming models

While existing expressive programming models leverage the sequential computation model to some extent, the fundamental difference lies in the necessity for additional abstractions, such as remote procedure calls, reactive values, and conflict-free replicated data type, for handling distributed aspects of the system [2, 6, 7]. (Transaction-based models, on the other hand, do not break the sequential model, but lack expressiveness for specifications of fine-grained allocation, reactivity, and optimizations [8].) The additional abstractions often cross-cut different parts of the program and break the conceptual model of the intended high-level behavior (e.g. by forcing a split into smaller computational parts which explicitly handle communication through messages). While these models abstract away some of the complexity, due to the close match between the program and the final distributed implementation, expressing certain complex behaviors requires low-level reasoning and careful structuring of the program [14].

A representative of a general-purpose model is the actor model [3, 10]. Despite being concise, (the running example is implemented in a similar number of lines of code, modulo message declarations) by providing abstractions which encapsulate sequential computation as responses to messages, the actor model suffers from cross-cutting concerns, similarly to other lower-level models. Programmers need to manually match the structure of the program to the resulting flow in the system: high-level behaviors which span across different nodes have to be split into behavior chunks, data explicitly partitioned and forwarded between nodes, and communication orchestrated through message sending and handling, depending on the orthogonal requirements, such as data consistency.

Notably, the design decision of separation of concerns afforded by our programming model inevitably comes with a loss of expressiveness for controlling communication. For example, our model takes away control from the programmer over how messages are sent across the system, so it cannot be used to implement distributed algorithms which require specific orchestrations of messages. However EdgeC leverages message passing, as well as various lower-level protocols, in the resulting implementations, to achieve the specified higher-level behavior.

6 Related and Future Work

EdgeC is inspired by prior frameworks, but aims to lift abstractions to a higher level [5]. Approaches that extend sequential model with new abstractions, such as RPCs and reactive values, handle distributed aspects specific to a domain, such as client-server and computation parallelization, but with no support for reactivity and consistency [4, 6]. Prior work presented systems that simplify development of reactive applications, but are not tailored for aspects needed for the edge, such as computation adaptation [8, 9].

The nature of our approach allows it to be extended with new specifications while hiding the reasoning complexity in the compiler: different specialized backends, such as MQTT, with necessary adjustments in the code generator; eventual consistency, with reasoning about CRDTs and general commutativity with SMT solvers; dynamic load-balancing, where behavior shapes from different specified network models are switched at run time, based on the current load. Aspects such as fault-tolerance could be supported by leveraging appropriate protocols inside the compiler (e.g. using protocols which achieve strong consistency of behaviors, but also fault-tolerance [11]). However, an interesting direction would be to extend the language to allow programmers to specify fine-grained fault tolerance and security aspects, providing the compiler more information for optimization.

References

[1] [n. d.]. Akka – actor toolkit and runtime. ([n. d.]). http://akka.io/
[2] [n. d.]. Meteor - Pure JavaScript web framework. ([n. d.]). http://meteor.com
[3] Gul Agha. 1986. Actors: a Model of Concurrent Computation in Distributed Systems. *MIT Press* (1986).
[4] Jean-Pierre et al. Briot. 1998. Concurrency and Distribution in Object-oriented Programming. *Comput. Surveys* (1998).
[5] Sergey Bykov et al. 2011. Orleans: Cloud Computing. In *SoCC*.
[6] Adam Chlipala. 2015. Ur/Web. In *POPL*.
[7] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *PLDI*.
[8] Irene Zhang et al. 2016. Diamond: Automating Data Management and Storage for Wide-Area, Reactive Applications. *OSDI* (2016).
[9] Rajeev Alur et al. 2016. Systems Computing Challenges in the Internet of Things. *white paper CCC* abs/1604.02980 (2016).
[10] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*.
[11] Brian M Oki and Barbara H Liskov. 1988. Viewstamped replication. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. 8–17.
[12] John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *ACM '72*.
[13] Dennis et al. Shasha. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.* (1995).
[14] Andrew Stuart Tanenbaum and Robbert van Renesse. 1987. *A critique of the remote procedure call paradigm.* Technical Report. Vrije Universiteit.