Operational Semantics for the Rigorous Analysis of Distributed Systems

Mohammed S. Al-Mahfoudh[1], Ganesh Gopalakrishnan[1], and Ryan Stutsman[1]

School of Computing, University of Utah, Salt Lake City, UT 84112, USA
{mahfoudh,ganesh,stutsman}@cs.utah.edu

Abstract. The development of distributed systems based on poorly specified abstractions can hinder unambiguous understanding and the creation of common formal analysis methods. In this paper, we outline the design of a system modeling language called DS2, and point out how its primitives are well matched with concerns that naturally arise during distributed system design. We present an operational semantics for DS2 as well as results from an ongoing Scala-based implementation slated to support a variety of state-space exploration techniques. The driving goals of this project are to: (1) provide a prototyping framework within which complex distributed system protocols can be stated and modeled without breaking the primitives down to low level ones, and (2) drive the development of interesting and distributed system-relevant property checking methods (e.g., linearizability).

Keywords: Distributed Systems, Operational Semantics, Scheduling, Concurrency, Actors

1 Introduction

Distributed systems, both large scale and small, are now critically important in nearly every aspect of computing and in our lives. From embedded microcontrollers to data centers and web applications that span multiple geographic locations, more developers are building distributed systems than ever before. The ability to rapidly construct and deploy correctly functioning distributed systems is a growing necessity; unfortunately, this is hard even for experts. Nondeterminism, weak update consistency, and complex failure handling protocols push far beyond what manual reasoning can grapple with, making it hard to offer *basic* safety guarantees.

The formal methods research community has begun responding to some of these challenges associated with building distributed systems. Recent efforts have formalized several famously subtle algorithms [25] and generated correct, synthesized implementations [13],[29]. While these efforts represent significant steps forward, their wider applicability is limited. For example, approaches that rely on automated theorem proving [29] are beyond the expertise of all but a handful of academics and tend to abstract away from situation-specific complexities. Approaches that require developers to model systems using predicate logic [22] or

model-checking frameworks [12],[16] provide only limited reasoning capabilities and do not provide the requisite higher-level abstractions necessary for intuitive modeling.

Concurrently with all these developments, new languages and frameworks are continually emerging to help developers rapidly create new distributed systems. Languages such as Go and Rust and frameworks such as Akka are enabling developers to build and deploy distributed systems more easily. However, these approaches provide no formal basis for reasoning about the safety and correctness of the resulting systems. The guarantees that they do provide are informally documented, are non-portable, and low-level. Consequently, the developers' ability to build and deploy complex systems has vastly outpaced their ability to *reason* about their behaviors and detect design flaws. There is however an opportunity here: by combining a simple and expressive programming model with strong formal guarantees, one can achieve the best of both worlds. With that goal in mind, we are developing a domain-specific language for distributed systems called DS2[1].

DS2 is being developed to allow developers specify, test, verify, and synthesize correct distributed systems built on a clear operational semantics. DS2 uses an Actor-based concurrency model [3],[11],[14] that is compatible with the popular Akka distributed systems programming framework [1]. A working prototype of DS2 system written in Scala exists, and many driving examples are being ported using a front-end. Preliminary results from our effort appear in DS2's official website [9]. This paper details the aforementioned preliminary work's operational semantics, both its normal operation and its faults model.

While developing some of the central features underlying state-space exploration of distributed systems[2] expressed in DS2, we realized first-hand how treacherous the corner-cases can be. We detail some of these subtleties in §7. This gave further impetus to our work on writing down a clear operational semantics.

The need for clear guiding semantics underlying distributed systems is further exemplified by Chord's [26] erroneous operation [31] that was discovered with help of a manually written model of it in an external model checker. In short, having a semantic basis helps carry out automated semantics-guided verification, supports more objective comparisons between various efforts, and helps prepare the community to handle newer protocols being designed. A variety of formal (e.g., linearizability checking [6]) and semi-formal (e.g., lineage-driven fault-injection [24]) techniques can also build on a clear semantics.

Background: It is important to point out some central characteristics of distributed systems, over and above those present in shared-memory concurrency (e.g., P-Threads) or traditional message-passing based parallel programming (e.g., using MPI). Distributed systems ingest all these concurrency-related subtleties, and additionally present other challenges:

---

[1] Domain-Specific/Declarative-Specification of Distributed Systems.

[2] Our focus is towards networked, asynchronous, deterministic, and non-Byzantine distributed systems. However, the model is *easily* extensible to include Byzantine distributed systems, too.

- Weak (eventual [5]) consistency is the norm rather than the exception. This adds to the non-intuitiveness of distributed systems behaviors.
- Given that faulty operation is the norm rather than the exception, processes of a distributed system hover on different planes of operation at different times (i.e. fault-free or faulty planes). Each time a fault occurs, the correctness guarantees are subsetted to a core set of primitive ones. When faults are correctly handled and normalcy returns, guarantees are elevated. A designer must be empowered with the ability to simulate these fault/recovery driven transitions, and check for the right level of guarantees being delivered.

2 Contributions

A key feature of our approach is the adoption of the simple *strategy* OO design pattern [27], wherein the scheduler is the algorithm and the distributed system is the context. This design approach provides both flexibility and extensibility both with respect to actual algorithms, as well as implementations. Another feature is that our design of DS2 facilitates the specification of normal behaviors, and introduces – in a layered manner – various scheduling options that mimic faulty behaviors. This matches the fault/recovery-driven transitions referred to earlier. We now present specific primitives of DS2 that facilitate the creation, extension, and systematic exploration of candidate distributed system designs:

- *Locking mechanism:* It allows an agent to control when to (and when not to) receive messages, to model a single process disconnect.
- *Message dropping:* It helps model failures such as dropped packets and/or network partitioning.
- *Futures:* At the implementation level, DS2 employs a mediator agent (temporary agent) to model an *ask* pattern. The use of a mediator agent removes the need to broadcast the handle to the replier, thus freeing the asker from manually handling the future resolution. This is inspired by the actual Akka [1] implementation.
- *Mixins:* Mixins are known as traits in Scala. They enable the designer to incorporate variations to fault models into a scheduler elegantly. This approach provides flexibility for algorithm designers, while at the same time shielding users from distributed systems complexities. This approach also helps avoid making *a priori* assumptions about fault models.
- *State capture:* capturing the global state and resumption from such captured global states allows us to develop backtracking algorithms, on-the-fly scheduler switching, and parallel schedules exploration.
- *Expressive power-wise:* DS2 [4] has a model part (that facilitates state space exploration) and a language part (a DSL for concisely specifying distributed systems). In this work, we focus on the model that facilitates the state space exploration and distributed systems analyses. It provides a rich set of primitives to support flexible state-space exploration methods (as compared with more standard model-checking notations such as TLA+ and Promela).

Our approach is more in line with the needs of a real distributed systems designer. In contrast, some prior language designs (notably Actors [3]), have emphasized novel state-space exploration algorithms [28], and have not emphasized (to the same extent) the creation of a guiding operational semantics or emphasis on key primitives as we summarized.

3 Overview

In this section, we give a brief overview of how our design choices fit together synergistically. In §4, we present a more detailed example, walking through most of the operational semantics rules to address the details.

In a real world example, when a distributed system has been constructed, normally one or more agents comprising it should be bootstrapped, i.e. started. It is exactly the same for our model, however with the scheduler taking control of how the state of the system should evolve, by deciding what events happen when and where.

After bootstrapping an agent, we arrange for it to have a `Start` message in its input queue. The scheduler dequeues this message, finds the matching action from the agent's reactions map, and makes a copy of that action-template and instantiates its relevant parameters. After that, the scheduler *schedules* the action into its task queue (making it a task). The scheduler then can choose whether to schedule something from another agent or merely resume executing the scheduled task(s). Consuming a statement leads to that statement being enqueued into the scheduler's consume queue. We employ a consume queue to expose interleaving effects of statements coming from different tasks scheduled by different agents. The scheduler then executes each statement, removing it from the front of the consume queue. This cycle repeats until some specified stopping criteria are met.

The aforesaid simple design gives the scheduler the ability to model different planes/levels of faulty behaviors in isolation (or combination, if chosen). Examples of situations that can be modeled include reordering, duplication, and dropping of messages; these are modeled by manipulating the agent's queue. Disconnects of single processes can be modeled using *locking*, and network partitioning can be modeled by *message dropping* from selected agents queue's, hence simulating missing updates. The scheduler consume queue can be manipulated to simulate different interleavings as well as delays of execution, forcing different bug scenarios. Similarly, a crash in a node (agent), and injecting a fault or a message that causes a fault can be simulated.

The rest of the paper is organized in the following manner: an illustrative example is given in §4, followed by the walkthrough §5 of the operational semantics rules that are stated in Figure 2, then the faults operational semantics are explained in §6. A real example of a bug discovery in our project is discussed briefly in §7. After that, we present related work in §8 followed by the concluding remarks in §9.

4 Walk Through Example

A distributed system is constructed by the code in Listing 1.1. We begin by creating a distributed system ds, a client c and a server s. Then a reaction is added to the client in lines 6 through 10. Two reactions are then added to the server in lines 12 through 16. Finally, the client and server are added to the distributed system that, in turn, is attached to the basic scheduler (*basic* in the sense of being controlled by the user).

```
1  val ds = new DistributedSystem("Echo ack")
   val s = new Agent("Server")
3  val c = new Agent("Client")
   val act1, act2, act3 = new Action
5  // Client setup
   act1 + Statement(UNLOCK,c) // unlocks the agent incoming q
7  act1 + Statement(ASK,c,new Message("Show","Hello!"),s, "vn")
   act1 + Statement(GET,c,"vn","vn2")
9  act1 + Statement(println("I'm Happy!"))
   c.R("Start") = act1 // (Start, act1) to reactions map
11 // Server setup
   act2 + Statement(UNLOCK, s)
13 act2 + Statement(println("Greetings!"))
   act3 + Statement((m:Message,a:Agent)=>println(m.p))
15 act3 + Statement((m:Message,a:Agent)=>send(s,m(p = true),m.s))
   s.R("Start") = act2 ; s.R("Show") = act3
17 ds += Set(s,c) // adding agents to system
   ds.attach(BasicScheduler)
```

Listing 1.1: An example distributed system, echo server-client interaction with additional blocking for an acknowledgement on client

One execution of such a distributed system is shown in Listing 1.2. We deliberately chose a schedule that leads to a large number of rule-firings. The listing is explained via the comments, and Figure 1 visualizes the resulting sequence of states step-by-step.

```
   val sch = ds.scheduler
2  sch.boot(s); sch.boot(c) // sends Start msg to s and to c
   sch.schedule(s) // schedule start task from s
4  sch.schedule(c) // schedule start task from c
   sch.consume(s)  // consume UNLOCK stmt from s task
6  sch.consume(s)  // consume "greeting" stmt from s task
   sch.consume(c)  // consume UNLOCK stmt from c task
8  sch.consume(c)  // consume ASK stmt from c task
   sch.executeOne  // UNLOCK s stmt, IsLocked(s) == false
10 sch.executeOne  // "greeting" s stmt
   sch.executeOne  // UNLOCK c stmt, IsLocked(c) == false
12 sch.executeOne  // ASK s stmt, T = {t} temporary agent
                   // and s.q == [Show("Hello",s=t)]
14 sch.schedule(s) // schedule "Show" task from s
   sch.consume(s)  // consume print("Hello") stmt
16 sch.consume(c)  // consume GET stmt from c task
   sch.consume(s)  // consume resolving send(..) stmt
18                 // note GET blocks, then it is resolved
   sch.consume(c)  // consume "happy" stmt from c task
20 sch.executeOne  // s print("Hello")
   sch.executeOne  // c blocks on GET, does not progress
22                 // putting back all stmts after it
                   // from cq back to front of task.xq in order
24 sch.executeOne  // resolving send(..), t.q != empty
                   // things happen to t.L("vn") future resolved
26                 // and then c.q = [RF(f,s=s)], note sender
                   // is s, not t
```

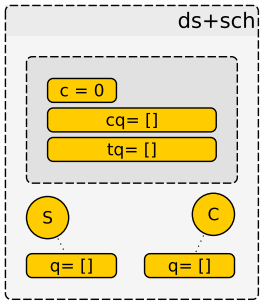
```
28 sch.handle(c)    // handling the RF message, unblocking c
   sch.consume(c)   // consuming GET from c again
30 sch.consume(c)   // consuming "happy" stmt from c
   sch.executeOne   // R GET c stmt, won't block (resolved)
32                  // c.L("vn2") = c.L("vn").val
   sch.executeOne   // print("I'm happy")
34 // DONE happy schedule, other schedules are not this happy
```

Listing 1.2: Example schedule invoking SCHEDULE, CONSUME, ASK, BLK-GET, R-SEND, and R-GET rules. The equivalent visualization of this execution is shown in Figure 1

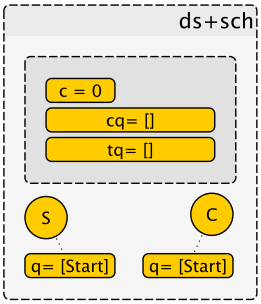
A design-time question might be: what schedule might lead to a client avoiding blocking on a future? The answer is to swap the consuming of the resolving send (on line 17 of Listing 1.2), with consuming the blocking get (on line 16 of the same listing). Another intent may be to exhibit a deadlock by the client. We then keep the same schedule, but we remove all statements and actions, consumed and scheduled respectively, from the server, and reset the server state (i.e. locking the server, then making $q = \epsilon$, and making its local state $\mathscr{L}$ empty) before it executes the resolving send. By doing the latter, we simulated a crash of the server and there is no way that client gets its future resolved, ending it in the simplest forms of a deadlock. One last attempt is to simulate a message drop for a message sent by the same resolving send to resolve the client's future (say we drop RF from client's queue, since getting the messages into the queue does not mean it is delivered, but rather means it is *in flight* and only considered delivered after it gets scheduled and/or handled by the receiving agent).

In addition, fault assumptions can be enabled/disabled selectively to affect how a scheduler explores a distributed system, i.e. the operation of a scheduler is *not* disconnected from these assumptions. In our implementation, we have these assumptions implemented in the form of mixin Scala traits. When these traits are mixed into a scheduler they change what different faults are/not allowed to be simulated by the scheduler. Even better, a developer can replace, extend, re-use and/or override these mixins by providing their own scheduler-specific implementations that, in turn, enable their schedulers inject/simulate faults in a very specific manner with respect to their algorithms.

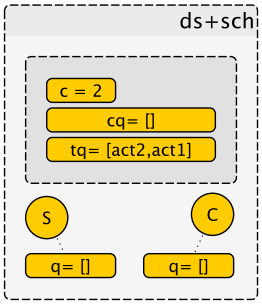
All these are illustrations that highlight the flexibility offered by the DS2 approach in creating faulty situations, and forcing a plethora of semi-formal state exploration methods to cover them.



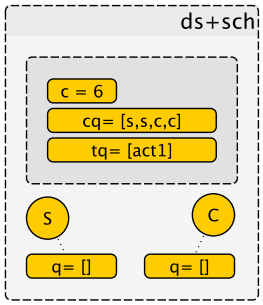
(a) Initial State



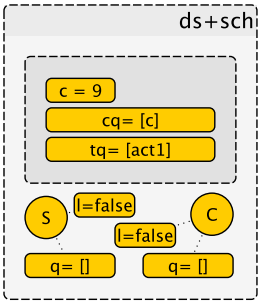
(b) Executed line 2



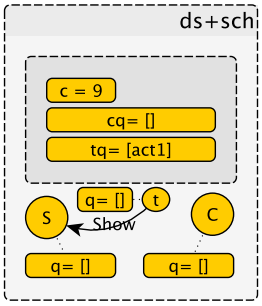
(c) Executed lines 3,4



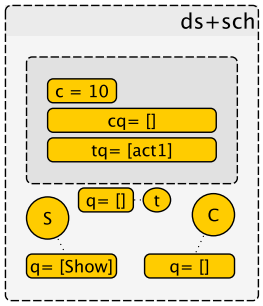
(d) Executed lines 5-8



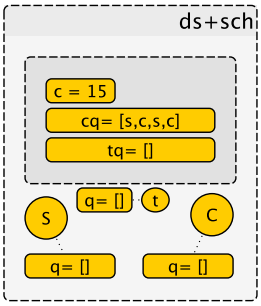
(e) Executed lines 9-11



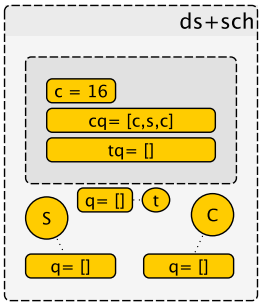
(f) Executing line 12



(g) Executed line 12



(h) Executed 14-19



(i) Executed 20

(j) Executed 21 (k) Executing 24-(1/4) (l) Executing 24-(2/4)

(m) Executing 24-(3/4) (n) Executing 24-(4/4) (o) Executed line 28

(p) Executed 29,30 (q) Executed 31 (r) Executed 33

Fig. 1: Example run invoking majority of operational semantics rules. Subfigures refer to line numbers in Listing 1.2

5 Operational Semantics

Despite DS2's simplicity, creating a concise semantics for it proved to be a challenge. Our initial operational semantics spanned dozens of pages [9], which led us to invent several abstract state predicates. The result is a concise semantics expressed in eight simple rules, and complemented with another eight fault-rules § 6, that still retain the full expressiveness that real-world developers demand. For example, it provides both asynchronous communication and synchronization primitives. At the same time, it forms the minimal rule set needed to understand and reason about communication and synchronization patterns (§5.2). This conciseness gives us confidence that our operational semantics and model will benefit designers who seek to build formal method tools and those seeking to model and understand new and existing distributed protocols.

We achieved this parsimony by defining a set of structures representing the state of a distributed system, a set of predicates to de-clutter the rules from mathematical details, and a set of conventions to make our presentation intuitive.

5.1 Structures

Now, we introduce the core structures of our semantics. We use the convention $(,,,p,q,,r,\ldots)$ to indicate three don't care arguments followed by p and q, then one don't care followed by r, and the tail sequence is a don't care.

Distributed System is a tuple $\langle \mathscr{A}, \mathscr{T}, \Sigma \rangle$ with the set of agents in a distributed system $\mathscr{A}$, the set of all *temporary* agents $\mathscr{T}$ (initially empty; each such agent handles resolving a single future, and then disappears), and a scheduler Σ. We will first elaborate the primitives in our model, and then return to the components of the distributed system.

Message is a tuple $\langle s, p_0, \ldots, p_n \rangle$ where s is the sender agent, and $p_0, \ldots, p_n$ is the payload of the message. The set of messages is symbolized by $\mathscr{M}$.

Statement is a wrapper around the code to execute. It is a tuple $\langle \gamma, code, k, p^* \rangle$ where γ is the action containing this statement, $code$ is the actual code to execute, k indicates the kind of a statement e.g. **Send** for send statement, and p^* are the parameters of the statement. A statement holds a reference to its containing action γ to provide access to the message m that invoked the action and agent a whose local state could be modified/accessed by the statement. In addition, in case a statement was preempted, it is known where it should be put back (to the front of the to-execute queue of the action γ); a walk-through of the rules in §5.4 will clarify this.

Action is essentially the sequence of statements to execute. More specifically, it is a tuple $\langle m, a, stmts, \zeta_x \rangle$ where a is the agent containing this action (whose local state may get modified/accessed by this action's statements), m is the message that invoked this action, $stmts$ is a sequence of statements or the template. ζ_x is a reference to the *to-execute queue* of statements. The set of all actions is Γ.

Timed Action is an action associated with two values of time. It is the tuple $\langle t1, t2, \gamma \rangle$ where γ is scheduled to execute every $t1$ time with tolerance of $t2$

$$\text{SCHEDULE}\ \frac{\begin{array}{c}ChoseSchedule(\Sigma,\alpha) \wedge \gamma_\alpha\ as\ \langle \mu,\alpha,stmts,\zeta_x\rangle = \mathscr{R}(\mu) \wedge \\ stmts_\gamma \neq \epsilon \wedge \zeta_{x,\gamma} = stmts_\gamma\end{array}}{\langle \mathscr{A}(\alpha(\mu.q,,\neg b,\mathscr{R},,,)),\mathscr{T},\Sigma(,\zeta_t,)\rangle \rightarrow \langle \mathscr{A}(\alpha(q,,\neg b,\mathscr{R},,,)),\mathscr{T},\Sigma(,\zeta_t.\gamma_\alpha,)\rangle}$$

$$\text{CONSUME}\ \frac{ChoseConsume(\Sigma,\alpha)}{\begin{array}{c}\langle \mathscr{A}(\alpha(,,\neg b,,,,)),\mathscr{T},\Sigma(,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,,,s.\zeta_x),\ldots),\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha(,,\neg b,,,,)),\mathscr{T},\Sigma(,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,,,\zeta_x),\ldots),\zeta_c.s)\rangle\end{array}}$$

$$\text{SEND}\ \frac{\begin{array}{c}IsSnd(s) \wedge \langle \alpha_s,\mu,\alpha_d\rangle = Involved(s) \wedge \neg IsRSnd(\mathscr{T},s) \wedge (p = p') \wedge \\ ChoseExOne(\Sigma)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\{\alpha_s(,,\neg b,,,,p),\alpha_d(q,\neg l,,,,,p')\}),\mathscr{T},\Sigma(c,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\{\alpha_s(,,\neg b,,,,p),\alpha_d(q.\mu,\neg l,,,,,p')\}),\mathscr{T},\Sigma(c+1,,\zeta_c)\rangle\end{array}}$$

$$\text{ASK}\ \frac{\begin{array}{c}IsAsk(s) \wedge \langle \alpha_s,\mu,\alpha_d,vn\rangle = Involved(s) \wedge (p = p') \wedge ChoseExOne(\Sigma) \wedge \\ f = fresh(Future) \wedge \alpha_t = fresh(Agent) \wedge where = fresh(Ids)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\{\alpha_s(,,\neg b,,,,p),\alpha_d(q,\neg l,,,,,p')\}),\mathscr{T},\Sigma(c,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha_s(,,,,,\mathscr{L}\cup(vn,f),p),\alpha_d(q.\mu(\alpha_t,),\neg l,,,,,p')),\mathscr{T}\cup \\ \alpha_t(,\neg l,\neg b,,,\mathscr{L}\cup\{(vn,f),(where,\alpha_s)\},p'),\Sigma(c+1,,\zeta_c)\rangle\end{array}}$$

$$\text{R-SEND}\ \frac{\begin{array}{c}IsSnd(s) \wedge \langle \alpha_s,\mu,\alpha_t\rangle = Involved(s) \wedge IsRSnd(\mathscr{T},s) \wedge \\ (p = p') \wedge ChoseExOne(\Sigma) \wedge RF = fresh(Message)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\alpha_s(,,\neg b,,,,p),\alpha_d(RF^*.q,\neg l,,,,\mathscr{L}\cup(vn,f),p')),\mathscr{T}\cup \\ \alpha_t(,\neg l,,,,\mathscr{L}\cup\{(vn,f),(where,\alpha_d)\},p'),\Sigma(c,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha_s(,,\neg b,,,,p),\alpha_d(RF^*.RF(\alpha_s,f(true,\mu[p_0])).q,),\neg l,,,,\mathscr{L}\cup(vn,f),p'), \\ \mathscr{T}\setminus\{\alpha_t(,,,,,\mathscr{L}\cup\{(vn,f(true,\mu[p_0])),(where,\alpha_d)\},p')\},\Sigma(c+1,,\zeta_c)\rangle\end{array}}$$

$$\text{R-GET}\ \frac{\begin{array}{c}(IsGet(s) \vee IsTGet(s)) \wedge \\ (\langle \alpha,vn,vn2\rangle = Involved(s) \vee \langle \alpha,vn,vn2,to\rangle = Involved(s)) \wedge ChoseExOne(\Sigma)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(vn,f(true,val)),)),\mathscr{T},\Sigma(c,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup\{(vn,f(true,val)),(vn2,val_f)\},)),\mathscr{T},\Sigma(c+1,,\zeta_c)\rangle\end{array}}$$

$$\text{BLK-GET}\ \frac{\begin{array}{c}IsGet(s) \wedge \langle \alpha,vn,vn2\rangle = Involved(s) \wedge \\ ChoseExOne(\Sigma) \wedge ss = PreEmpted(\Sigma,\alpha)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(vn,f(false,)),)),\mathscr{T},\Sigma(c,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,\alpha,,\zeta_x),\ldots),s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha(,,b,,,\mathscr{L}\cup(vn,f(false,)),)),\mathscr{T},\Sigma(c+1,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,\alpha,,s.ss.\zeta_x),\ldots),\zeta_c\setminus ss)\rangle\end{array}}$$

$$\text{T-GET}\ \frac{\begin{array}{c}IsTGet(s) \wedge \langle \alpha,vn,vn2,to\rangle = Involved(s) \wedge \\ ChoseExOne(\Sigma) \wedge ss = PreEmpted(\Sigma,\alpha) \wedge to \leq 0\end{array}}{\begin{array}{c}\langle \mathscr{A}(\alpha(,,,,,\mathscr{L}\cup(vn,f(false,)),)),\mathscr{T},\Sigma(c,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(vn,f(false,)),)),\mathscr{T},\Sigma(c+1,,\zeta_c.ss)\rangle\end{array}}$$

Fig. 2: DS2 Operational Semantics

amount of time, usually used to model *heart beat*. The tolerance time $t2$ is there to add delay-tolerance for message delivery, since it cannot be predicted.
Future is a synchronization construct, which is a tuple $\langle r, val \rangle$ where r is a boolean indicating whether the future is resolved (defaults to $false$) and val is the value resolving this future (defaults to $\perp$, or no value). A future object is returned by the **Ask** statement type, as will be presented shortly.
Agent is a communicating autonomous process. It is a tuple $\langle q, l, b, \mathscr{R}, \tau, \mathscr{L}, p \rangle$ where q is its receive queue, the *reactions* of the agent $\mathscr{R} : \mathscr{M} \rightarrow \Gamma$, timed actions set $\tau = \{\langle t1, t2, \gamma \rangle\}$, $\mathscr{L}$ is the local state of an agent $\mathscr{L} : Ids \rightarrow Vals$. Blocked b and locked l are flags whose initial values are `false` and `true`, respectively. A blocked agent cannot execute statements, consume, or schedule tasks, an agent is blocked if a future it tries to access is not resolved. Locked l is used by the scheduler to tell if the process is receiving messages or not, e.g. its server socket is not open, to model a disconnect. In order to model *network partitioning*, however, the p is the partition id to which this agent belongs, initially the same for ALL agents. Many or all agents can share the same value, if they are in the same network partition. Otherwise, agents in different partitions *must* have different partition id.
Scheduler encodes the runtime (an algorithm) of the distributed system. It is a tuple $\langle c, \zeta_t, \zeta_c \rangle$ where ζ_t is the task queue (a queue of actions), ζ_c is the consume queue (queue of statements), and c is a logical clock (lamport clock [20]) that is incremented by one each time a statement is executed. The role of a scheduler is to explore a sub-set of the allowed behaviors by the operational semantics in order to reveal a conclusion about the system under analysis.

5.2 Communication & Synchronization

Our model uses a minimal set of four communication and synchronization primitives. Such a restriction does not limit the expressivity of the language, while at the same time facilitating understanding. These primitives are now explained.
Send is a fire-and-forget type of statement. Its signature is $send(\alpha_{src}, \mu, \alpha_{dst})$.
Ask is a fire and return a handle (future) statement. Its signature is $ask(\alpha_{src}, \mu, \alpha_{dst}, vn)$. Its details are similar to that of send, except for vn which is a *variable name* where the handle (future) f is stored in source agent α_{src} local state. When an ask statement is fired, the message μ is sent to the destination agent, by a *temporary agent* on behalf of the source agent α_{src}. Later in time, there may/not be a reply (possibly from another agent than α_{dst}) to the sender of the message that resolves the future when received by the temporary agent. The temporary agent would update the source agent α_{src} upon receiving that reply. **Get** is the statement used to *block on* a future object. Blocking means not executing any more statements till that future object is resolved, then the agent is unblocked to schedule, consume, and/or execute tasks/statements. There are *two* variants of get statements. The blocking-get signature is $get(\alpha, vn, vn2)$ and the timed-get signature is $get(\alpha, vn, vn2, to)$. The agent calling it is α, the variable holding the future is vn, the variable that holds resolved value is $vn2$, and the time out limit is to.

5.3 Predicates and Conventions

We first need to state our conventions. **References and Types** are inferred directly from alphabets $\alpha, \mu, \gamma, s, f, vn$ and $where$ as follows: an agent with $\alpha \in \mathscr{A}$, message with $\mu \in \mathscr{M}$, action $\gamma \in \Gamma$, statement $s \in Statement$, and future with $f \in \mathscr{F}$. We also refer to variable names with $\{vn, where\} \in Ids$, appended with a number if more than one. The same thing goes to other types. **Subscripts** are used in two ways. First, in parameters to indicate the function of the parameter e.g. α_s for source agent. Second, we use it to indicate where the entity belongs e.g. $\mathscr{L}_\alpha$ for local state of agent α. **Specific task in a scheduler** $\gamma_{\Sigma,\alpha}$ means a front-most action in a scheduler's task queue ζ_t that was scheduled by agent α. We also use the same **structure (tuple) as a predicate** with commas indicating place inside the tuple representing them as opposed to dots that are used to state the flexibility of location inside a tuple/sequence. `Involved(`s`)` returns a variable length tuple according to the kind of current statement. The tuple represents the arguments *involved* in this statement.

$$Involved(s) = \begin{cases} \langle \alpha_{src}, \mu, \alpha_{dst} \rangle & \text{if } s[k] = SEND \\ \langle \alpha_{src}, \mu, \alpha_{dst}, vn \rangle & \text{if } s[k] = ASK \\ \langle \alpha, vn, vn2 \rangle & \text{if } s[k] = GET \\ \langle \alpha, vn, vn2, to \rangle & \text{if } s[k] = TGET \\ undefined & otherwise \end{cases}$$

The meanings of symbols inside each tuple returned are as explained in §5.2.

5.4 Rules Walkthrough

Now, we explain the rules in Figure 2 one by one, and illustrate the effects of these rules with the aid of the subfigures of Figure 1 (when applicable). Let us start by explaining one rule, namely `SCHEDULE`.

Schedule. The `SCHEDULE` rule states that to schedule a task from an agent α, the agent needs to be in the unblocked ($\neg b$) state. Further, the agent should have an entry in its reactions map $\mathscr{R}$ that maps the message received, μ, residing at the head of its queue q (indicated by $\mu.q$), to an action. Moreover, the action field formal parameters m and a must be set to μ and α, respectively. Further, the action's execute queue ζ_x must refer to $stmts$ (indicated by $\zeta_{x,\gamma} = stmts$), where $stmts$ is nonempty. In addition, the `ChoseSchedule(`Σ, α`)` predicate must be true. The predicate means the scheduler *chose to schedule* a task from agent α. We use the "as" notation, as in Ocaml, to serve as an alias; for instance, $\gamma_a \, as \, \langle \mu, \alpha, stmts, \zeta_x \rangle$ uses γ_a as an alias for $\langle \mu, \alpha, stmts, \zeta_x \rangle$. The system transitions to a state where the message μ is removed from the front of agent α's queue, and the action γ_α is appended to the scheduler's task queue (ζ_t). An example of this rule before it triggered twice is shown in Figure 1b and after it triggered in Figure 1c.

Consume. The `CONSUME` rule fires when the scheduler (Σ) has a task in its task queue (ζ_t). In addition, that task is scheduled by a currently non-blocked ($\neg b$) agent α. Further, the predicate `ChoseConsume(`Σ, α`)` must return true, which means that the scheduler *chose to consume* from a front-most task (symbolized by

$\gamma_{\Sigma,\alpha}$) that was scheduled by agent α. Then, the system transitions by (1) popping the statement that is at the front of the task's execute queue ($s.\zeta_x$) (2) appending that statement to the back of the scheduler's consume queue ($\zeta_c.s$). An example of this rule before triggering four times is shown in Figure 1c and after it triggered in Figure 1d.

Send. This rule (`SEND`) states that if the current statement s at the front of the scheduler consume queue ζ_c is a send statement (`IsSnd(`s`)`), and that statement parameters returned by `Involved(s)` are $\langle \alpha_s, \mu, \alpha_d \rangle$, and the statement is not a resolving send ($\neg IsRSend(\mathscr{T}, s)$), i.e. the destination α_d isn't a member of the temporary agents. In addition, neither the source agent α_s is blocked ($\neg b$) nor the destination agent α_d is locked nor they reside in different network partitions ($p = p'$). Then, if the scheduler *chose to execute* a statement (`ChoseExOne(`Σ`)`), the transition happens. That is, the message μ is appended to agent α_d queue ($q.\mu$). In addition, the statement is removed from the front of the scheduler's consume queue (ζ_c). An example of this rule when it triggers is shown in Figure 1f and after it completes is shown in Figure 1g.

Ask. `ASK` rule states that if the current statement to execute is an ask, predicate `IsAsk(`s`)`, and like in `SEND` rule, the source agent is in unblocked state ($\neg b$) and the destination agent α_d is in unlocked state ($\neg l$), and they reside in the same network partition ($p = p'$). Then, if the scheduler *chose to execute* a statement, a fresh temporary agent α_t (i.e. α_t =`fresh(Agent)`) is created along with a fresh future $f \in \mathscr{F}$ (i.e. f =`fresh(Future)`). Then, the transition happens: (1) the temporary agent α_t is added to the temporary agents $\mathscr{T}$, inheriting the same network partition as the asker/source agent (p'), (2) the future f is added to both the temporary agent and the source agent α_s local states under the key vn, i.e. $\mathscr{L} \cup (vn, f)$ (3) the temporary agent local state updated with the $(where, \alpha_s)$, to keep track *where to* forward the RF (Resolve Future) message in case the future was resolved. (4) the sender field of the message updated to be the temporary, $\mu(\alpha_t,)$ (5) the ask message enqueued at the destination agent α_d's receive queue, $q.\mu$. An example operation of this rule is visualized in multiple frames of Figure 1, namely in frames 1e, 1f, and 1g. Important to notice that the updates to local states of both temporary and source agent stated here are *not* shown in the figure due to space constraints.

Resolving Send. `R-SEND` rule states the same guards as a regular `SEND` rule except that the destination is a temporary agent, i.e. predicate $IsRSnd(\mathscr{T}, s)$. As such, additional work need be done over a normal send by α_t. So, if the scheduler *chose to execute* the statement, the transition happens: (1) The temporary agent updates its future resolved status to `true` and that future's value from the first payload of the message μ, and encapsulates it in a fresh RF message setting its sender to the original sender α_s, i.e. $RF(\alpha_s, f(true, \mu[p_0]))$ (2) The RF message is then *inserted* into the destination agent α_d queue with the resolved future, before all non-Resolve-Future messages but after all other RF messages, as shown by $RF^*.RF(\alpha_s, f(true, \mu[p_0])).q$ (3) The temporary removes itself from the temporary agent's set, $\mathscr{T} \setminus \{\alpha_t(,,,,,\mathscr{L} \cup \{(vn, f(true, \mu[p_0])), (where, \alpha_d)\}, p')\}$. Up to this point, the future is considered resolved, however it is up to the scheduler im-

plemented to decide when to update α_d local state with the resolved future, as can be told from the post state of the destination agent local state, $\mathscr{L} \cup (vn, f)$. An example of a resolving send executing is shown in multiple frames in Figure 1, namely frames: 1k, 1l, 1m and 1n.

Resolved Get/Timed-Get. `R-GET` rule states that if the current statement is either a blocking-get (i.e. `IsGet(`s`)`) or a timed-get (i.e. `IsTGet(`s`)`), and the future they try to retrieve is already resolved,$f(true, val)$. In addition, that future is stored in α's local state under entry vn. Further, the scheduler *chose to execute* a statement. Then, that future's value is retrieved and stored in another entry in the local state of the same agent, i.e. $\mathscr{L} \cup \{(vn, f(true, val)), (vn2, val_f)$. Figure 1p shows the state before this rule triggered, and Figure 1q shows the effect after it is triggered by agent c.

Blocking Get. `BLK-GET` rule states the same guards stated by `R-GET` except that: (1) the future in this case is *not* resolved (2) the current statement is a blocking-get ($IsGet$) (3) and the future is unresolved $f(false,)$. If the scheduler *chose to execute* a statement, a transition happens: All statements indicated by ss, that are returned by the $PreEmpted(\Sigma)$ in the same order they were consumed, are *appended* to the current statement and the resulting sequence of statements *prepended* to the front-most task execute queue, i.e. $s.ss.\zeta_x$. `PreEmpted(`Σ`)` determines all those statements, that were consumed from the same agent α tasks and returns them. Then, these statements in ss are removed from the consume queue, as in $\zeta_c \setminus ss$. Lastly, the agent blocking status is updated from unblocked $\neg b$ to blocked b. An Example of this rule prior it triggers is shown in Figure 1i and after it triggers in Figure 1j.

Timed Get. `T-GET` rule states the same guards as in `BLK-GET` rule except it does not block indefinitely. It only blocks temporarily until it times out ($to \leq 0$), delaying execution of all those statements till the agent unblocks. That is the time for them to have been consumed, i.e. appended to the consume queue of the scheduler, $\zeta_c.ss$ (skipping the statement s). Agent α's state changes to unblocking.

6 Faults & Exploration Semantics

We present the semantics for faulty behaviors that can, in our framework, be introduced during testing. These rules can be thought of as facilitating high-level fault injection. Structuring high-level fault injection rules in this manner ensure that users can cover faulty scenarios systematically.

In § 6.1 we will walk through the rules shown in Figure 3 and explain in detail how they work. The next section, § 6.2, we will show how they can be used to simulate faults in the context of previous sections example, and other complementary examples when needed.

6.1 Rules Walk-Through

In order for a scheduler to determine the existence of a bug in the model, it needs some facilities. These facilities, as we mentioned before, include the presence of

$$\text{MSG-DROP}\ \frac{ChoseDrop(\Sigma,\alpha,\mu_m)}{\langle \mathscr{A}(\alpha(\ldots \mu_m \ldots,,,,,,,)),\mathscr{T},\Sigma\rangle \rightarrow \langle \mathscr{A}(\alpha(\ldots,,,,,,,)),\mathscr{T},\Sigma\rangle}$$

$$\text{IMPL-DROP}\ \frac{\begin{array}{c}[((IsSnd(s)\vee IsRSnd(s))\wedge\langle\alpha_s,\mu,\alpha_d\rangle = Invovled(s))\vee \\ (IsAsk(s)\wedge\langle\alpha_s,\mu,\alpha_d,\rangle = Involved(s))]\wedge(p\neq p')\wedge ChoseExOne(\Sigma)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\alpha_s(,,\neg b,,,,p),\alpha_d(q,,\neg b,,,,p')),\mathscr{T},\Sigma(,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha_s(,,\neg b,,,,p),\alpha_d(q,,\neg b,,,,p')),\mathscr{T},\Sigma(,,\zeta_c)\rangle\end{array}}$$

$$\text{MSG-REORD}\ \frac{ChoseReOrder(\Sigma,\alpha,O)\wedge O\in\{P2P,\neg P2P\}\wedge QPermuted(q,O,q')}{\langle \mathscr{A}(\alpha(q,,,,,,)),\mathscr{T},\Sigma\rangle \rightarrow \langle \mathscr{A}(\alpha(q',,,,,,)),\mathscr{T},\Sigma\rangle}$$

$$\text{TASK-INTRLV}\ \frac{ChoseInterleave(\Sigma,O)\wedge O\in\{PO,\neg PO\}\wedge TPermuted(\zeta_t,O,\zeta_t')}{\langle \mathscr{A},\mathscr{T},\Sigma(,\zeta_t,\zeta_c)\rangle \rightarrow \langle \mathscr{A},\mathscr{T},\Sigma(,\zeta_t',\zeta_c)\rangle}$$

$$\text{MSG-DUPL}\ \frac{ChoseDupl(\Sigma,\alpha,\mu_m)}{\langle \mathscr{A}(\alpha(\ldots \mu_m \ldots,,,,,,,)),\mathscr{T},\Sigma\rangle \rightarrow \langle \mathscr{A}(\alpha(\ldots \mu_m \ldots \mu_m \ldots,,,,,,,)),\mathscr{T},\Sigma\rangle}$$

$$\text{NET-PART}\ \frac{ChosePartition(\Sigma,\{\alpha_{a0},\ldots,\alpha_{am}\})\wedge p' = fresh(Ids)}{\begin{array}{c}\langle \mathscr{A}(\alpha_{a0}(,,,,,,p),\ldots,\alpha_{am}(,,,,,,p),\alpha_{b0}(,,,,,,p),\ldots,\alpha_{bn}(,,,,,,p)), \\ \mathscr{T},\Sigma\rangle \rightarrow \\ \langle \mathscr{A}(\alpha_{a0}(,,,,,,p'),\ldots,\alpha_{am}(,,,,,,p'),\alpha_{b0}(,,,,,,p),\ldots,\alpha_{bn}(,,,,,,p)), \\ \mathscr{T},\Sigma\rangle\end{array}}$$

$$\text{NET-UNPART}\ \frac{\begin{array}{c}ChoseUnPartition(\Sigma,\{\alpha_{a0},\ldots,\alpha_{am}\},\{\alpha_{b0},\ldots,\alpha_{bn}\})\wedge \\ (p'\neq p)\wedge p'' = fresh(Ids)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\alpha_{a0}(,,,,,,p'),\ldots,\alpha_{am}(,,,,,,p'),\alpha_{b0}(,,,,,,p),\ldots, \\ \alpha_{bn}(,,,,,,p)),\mathscr{T},\Sigma\rangle \rightarrow \\ \langle \mathscr{A}(\alpha_{a0}(,,,,,,p''),\ldots,\alpha_{am}(,,,,,,p''),\alpha_{b0}(,,,,,,p''),\ldots, \\ \alpha_{bn}(,,,,,,p'')),\mathscr{T},\Sigma\rangle\end{array}}$$

$$\text{PROC-CRSH}\ \frac{ChoseCrash(\Sigma,\alpha)}{\langle \mathscr{A}(\ldots,\alpha,\ldots),\mathscr{T},\Sigma\rangle \rightarrow \langle \mathscr{A}(\ldots),\mathscr{T},\Sigma\rangle}$$

Fig. 3: DS2 Faults Semantics

a locking mechanism to simulate network partitioning, fault injections to simulate, for example, message duplication, and many others such as network message delivery/dropping mechanisms like the incoming queue. The reader is highly encouraged to refer frequently to Figure 3 in order to understand rules while reading this walk through.

Message Dropping. `MSG-DROP` rule states that if the scheduler (Σ) chose to drop a message (μ_m) from an agent ($\alpha(\ldots \mu_m \ldots,,,,,,,)$), shown as $ChoseDrop(\Sigma,\alpha,\mu_m)$, then that message is discarded from that agent's queue ($\alpha(\ldots,,,,,,,)$). This rule is of extreme importance, and the reason is that message dropping is the root cause of the majority of problems in distributed systems. As a matter of fact, the majority of faults can be reduced to message dropping or are a product of dealing with dropped messages. Our *non-primitive* network partitioning rules are built around *implicit message dropping* by communication rules (`SEND`, `ASK`, and

R-SEND). When these three rules detect a different partition indicator/identifier between the source and destination agents, they drop the message to be sent; that is what rule IMPL-DROP in Figure 3 does and is explained next.

Implicit Message Dropping. The rule IMPL-DROP states that on executing any statement ($ChoseExOne(s)$) that is either a send or a resolving send (($IsSnd(s) \vee IsRSnd(s)$), or an ask ($IsAsk(s)$) whose involved communicating agents (a_s and a_d), shown as $\langle a_s, \mu, a_d \rangle = Involved(s)$ for send/resolving-send and as $\langle a_s, \mu, a_d, \rangle = Involved(s)$ for ask, reside on different network partitions ($p \neq p'$) then the message (μ) is implicitly dropped. Hence, the post state is the exact equivalent to the prior state of the distributed system.

Message Reordering. MSG-REORD rule states that if: (1) The scheduler (Σ) chose to reorder ($ChoseReOrder(\Sigma, \alpha, O)$) an agent's ($\alpha$) incoming queue according to one of the policies specified by $O \in \{P2P, \neg P2P\}$ (P2P policy means that the new reordering of messages must keep the relative order between point-to-point communication, i.e. messages sent from the same sender and to the same receiver should maintain their relative order), and (2) the effect of permuting these messages in the old version of the queue (q) according to the policy specified above, $QPermuted(q, O, q')$, resulted in a policy-respecting queue of messages (q'), then the system transitions and the new reordering of messages takes effect.

Task Interleaving. Task interleaving rule TASK-INTERLV states that if a scheduler (Σ) chose to interleave its task queue (ζ_t), stated as ($ChoseInterleave(\Sigma, O)$), according to one of the polices $O \in \{PO, \neg PO\}$, then the resulting queue (ζ_t') is a permutation of the old task queue that conforms to the policy specified($TPermuted(\zeta_t, O, \zeta_t')$). The policy PO means that the interleaving of tasks should respect "program order". That is, the order of tasks scheduled by the same agent should stay the same relative to each other.

Message Duplication. The message duplication rule (MSG-DUPL) is a fault injection mechanism. This rule states that if a scheduler (Σ) chose to duplicate a message (μ_m) that is in an agent's queue ($\alpha(\ldots \mu_m \ldots, , , , , ,)$) – all of that is stated as $ChoseDupl(\Sigma, \alpha, \mu_m)$ – then it inserts a copy of it anywhere in the agent's queue ($\alpha(\ldots \mu_m \ldots \mu_m \ldots, , , , , , ,)$) in a way that is specific to the target task of that scheduler. This is why a certain position of where the duplicated message is to be inserted into an agent's queue is not specied. Other kinds of message duplication can be due to either crashes of processes followed by retries e.g. Re-transmission of such messages till an ack is received, or due to some other *behavior induced* re-transmission[3]. Such behaviors are not controlled by our model, since they are specified by target systems' developer(s), and do impose causality constraints for message duplication. However, they are explorable by schedulers even if said schedulers do not use fault injection mechanisms. Fault injection using message duplication is a way to give that last nudge (when appropriate) to the system to hit a bug, and hence is prioritized last compared to other rules that simulate faultiness.

Network Partitioning. There are two rules that manipulate network partitioning (NET-PART and NET-UNPART), and another that makes use of network parti-

[3] e.g. a heart beat sending messages every certain time period

tioning to implicitly drop messages (IMPL-DROP explained before). The network partition rule NET-PART states that if the scheduler (Σ) decided to partition a set of agents ($\{\alpha_{a0}, \ldots, \alpha_{am}\}$) away from a distributed system – that is to place them in a separate partition of their own – then the partition identifier (p) in these agents is changed to a fresh partition identifier (p'), indicated by $p' = fresh(Ids)$. The network unpartitioning rule NET-UNPART, on the other hand, (1) takes two sets of agents having different partition numbers per set, (2) creates a new partition id for the resultant combined partition, and (3) updates the partition id on all of the agents from both sets/partitions to be (p''), i.e. the new partition id. After which, all agents reside in the same partition (p'') and can communicate with each other. An implementation of the model may choose any function that guarantees the deterministic resultant partition id for combining the two. The model does not impose any restriction to do that or not.

Process Crashing. The process crashing rule PROC-CRSH states that if a scheduler (Σ) chose to crash ($ChoseCrash(\Sigma, \alpha)$) an agent ($\alpha$), then that agent simply ceases to exist in the distributed system ($\mathscr{A}(\ldots)$), along with its stored state (queue and local state[4]).

In the next section, we will discuss how these fault rules make sense in a realistic system exploration by discussing the previous example and introducing high level complementary ones when needed.

6.2 How Faults-Rules Help

From Figure 1 we will try to focus on deadlock detection to illustrate the importance of the rules shown in Figure 3. This helps in illustrating the benefits of the faults we can model to address realistic situations. In the case that the deadlock detection example does not apply to illustrate a rule, we will introduce a complementary example to help.

Message Dropping. A message is not considered delivered till it is handled by the agent, e.g. by stashing it or by processing (potentially producing more messages, and other possibly irreversible side effects). This leaves sufficient room for the model to address realistic message dropping scenarios to model different faults. One example is that we can model packet/message loss by dropping that message from the destination agent's queue. We could have dropped the resolve future message (RF) from the client's queue to simulate the packet/message loss. The client, then, can deadlock and the scheduler will be able to detect that. These examples (along with upcoming network partitioning examples) show the importance of the ability to simulate message dropping in a formal model for distributed systems.

Message Reordering. An intuitive example of message reordering rule benefit is easily shown in multiple examples. One could think of a bank account getting mutated by two clients at the same time, $c1$ and $c2$. These clients issue their orders independently, while the bank account keeps track of what balance remains

[4]For this base model being discussed in this work, all it has as a state is a queue and a local-state. Implementations, of course, can have more than that.

to be withdrawn. Let us assume that the initial balance is zero and it is updated in real time, i.e. no human supervision and/or review is involved in updating the balance. Client $c1$ deposits some money, but meanwhile $c2$ is withdrawing from the balance. If $c1$'s message/update reaches before $c2$ request is received and fulfilled, $c2$ is happy. If, however, $c2$'s request happens before $c1$'s request, $c2$ is not happy. Client $c2$ keeps on re-issuing the same request, but withdrawal is always prioritized (or reaches the server faster) over $c1$'s. There, we can see that message reordering simulating delayed messages (relative order of messages arriving at the server) taking bad effect on the outcome of $c2$. The same scenario happens with updating any shared resource that supports any non-commutative pair of operations, except it may go much deeper than two interactions between just three agents. All of these scenarios are enabled by the message reordering rule.

Message Duplication. Consider the same example for the bank account discussed above. Since $c2$ is duplicating the request often, there could be the possibility of repeating a withdrawal. It could be the case that one request took completely different route to the server, and got delayed beyond anticipation. Due to that, $c2$ times out and normally re-issues the same request. However, the old message gets delivered along with the new one and both get processed. Of course, this duplication is very common in distributed systems, and normally processes should be coded to handle such scenarios. It is helpful to have such exploration ability in the model to check for unwanted behavior due to message duplication and that logic handling such duplication handles the situation correctly.

Tasks Interleaving. In the case that message reordering is short of showing a bug, can there be a tweaked task interleaving to target that bug and show its existence? It turns out that, in our bank account example, if the server dispatches its tasks in a program-order (PO) and point-2-point ($P2P$) respecting manner, there still could be a way to interleave the tasks of two different agents in a way that they cause problems, e.g. overdraft. The worst case scenario would be the server dispatches all tasks coming from $c2$ then the account balance is doomed, if there is no overdraft protection. If at least one $c2$ task was dispatched before $c1$ deposits money, a bit more optimistic, it still can cause overdraft. A scheduler relying on this rule can explore said scenarios, even if message reordering fell short of revealing a bug.

Network Partitioning. One example is that we can model a network partition between the client and the server shown in the example, dropping all messages going either way implicitly by any communication primitive described previously. Another example is to simulate a network partition before that RF message (in example shown in Figure 1) is sent to the client and it will be auto-dropped based on the partitioning semantics and the `IMPL-DROP` rule semantics. The same network partition, between the client and server, can actually occur before the client sends its request. That would lead to the request to be lost (i.e. not delivered to the server for handling), and then the way the client is coded assumes it was delivered. That, in turn, will lead the client to *block* over a promised future whose request isn't even delivered, leading to another deadlock scenario. A more involved

network partitioning scheme that may be simulated by network partitioning (implicitly message dropping between partitions) is to keep two partitioned sets of agents to evolve their state and stay divergent to simulate some consistency violation scenarios.

Process Stop/Crash. Going back to our deadlock detection example, from Figure 1, it is easy to crash the server after/before the client request is issued (but before the server processes it) to cause a deadlock at the client. That server may come back online, after it lost the message, or it can stay down forever, leaving the client blocking on an unresolved future. Under the assumption of having some kind of a persistent state private to each agent, e.g. by checkpointing in-memory data to the disk, there would be an interesting set of examples to investigate the state consistency between processes that occasionally crash then reboot and weather they may recover and converge from state view divergence. Our implementation of the model takes advantage of the assumption that there is always private disk (persistent) store available to each agent, in order to be able to simulate that. Actually, in any distributed system, when there is an agent that is to process requests on which other agents are waiting or that causes more updates to propagate through the distributed system, crashing certain processes is a good idea to cause data consistency problems across the system (divergent state view) and/or other concurrency problems (e.g. deadlocks). After all, a lot of distributed systems problems/faults are due to message dropping, processes demise and/or reboots.

7 Bug Discovery in Snapshot of Runtime

By forcing us to think clearly, the operational semantics helped expose a bug in the original implementation of snapshotting support. For example, before we had an operational semantics, we struggled to correctly capture snapshots of the runtime state (which includes both the distributed system and the scheduler attached to it). For the most part, a system and its scheduler must be "deep copied," to create an isomorphic state; however, not everything can be copied verbatim. When copying a distributed system, the steps must be carried out in two phases: (1) a copy phase (creating objects but leaving references untouched) for all entities in it, followed by (2) a link phase (*re-wiring* references to entities from new snapshot) for all of them. However, in our original implementation, this was not the case. For example, an action's ζ_x statements kept on referring and affecting the original distributed system's agents even after the linking phase. More over, ζ_x statements were not the same statements from the snapshot's `stmts` template. So, when the link operation updated the agent field `a`, which in turn updates all of the template statements, it did not reflect in those inside of ζ_x. The operational semantics exposed the bug in the original snapshotting implementation that left actions in the snapshot *still attached* to the parent distributed system instead of their snapshotted counterparts. The operational semantics were essential in correcting snapshotting, which will be the cornerstone of DS2's model checking and testing functionality.

After correcting this mistake, DS2 can now easily and reliably capture, fork, and restore distributed system states. Listing 1.3 shows how succinctly one can now snapshot a whole distributed system along with its scheduler state and then restore from it.

```
val state = saveState(sch)
restoreState(sch,state)
```

Listing 1.3: Capturing and restoring to runtime state

A key detail here is that the scheduler `sch` *need not be the same scheduler that saved the state*. This flexibility enables different schedulers, for example, those running truly concurrently to explore different paths in the distributed system's state-space; and/or cooperation between different analyzing schedulers, i.e. switching on the fly between different kinds of analyses to be performed by different schedulers. An additional importance of this snapshot feature is enabling backtracking/stateful algorithms.

8 Related Work

Several distributed-system related projects have emphasized the use of formal semantics. SimgGrid [7]focuses on the simulation and model checking of distributed systems. InVerdi [29], the Coq system is used to develop formal operational semantics for network and node-failure models to synthesize distributed systems from specifications. Our work focuses on targeted correctness of distributed systems, as opposed to performance simulation as in SimGrid. Our goal is to allow designers to model a variety of distributed systems in the DS2 language; the formal definition of a modeling language is not targeted in Verdi.

MoDist [30] is a transparent operating system-agnostic model checker for unmodified distributed systems. Our work is extensible to address specific needs of distributed systems using custom schedulers. MaceMC is an execution based model checker for distributed systems specific to the Mace language. It benefits from coarse-grained interleaving exploration and random walk [17].

IronFleet [13] is a layered approach to TLA-style [22] state-machine refinement and Hoare-logic [15] based verification to synthesize provably correct distributed systems, developed by Microsoft Research.

Thanks to its utilization of operational semantics information for control flow statements and state space reduction policies, SAMC [23] was able to show huge gains, both in performance and precision, over regular Dynamic Partial Order Reduction (DPOR) based approaches.

Pony [2] is an actor based programming language that relies on avoiding blocking (i.e. lockless) to produce high performance distributed systems. Future support for fault-tolerance is expected to be based on Erlang's actors supervisory hierarchies [8].

Distributed Closures [10] rely on fixing distributed data (i.e. not allowing mutation) that are called "silos" and sending closures (function shipping) that are called "spores" instead of sending messages. These functions construct a lazy

graph of computations over these constant distributed data that are finally evaluated/computed when needed to be materialized. This provides strong type safety, a functional approach to programming distributed systems, and some fault tolerance.

An approach for sequential programming (actors) in distributed systems [19] proposes a paradigm shift in programming distributed systems, by suggesting programming language level support to address said systems. Future direction of this work may support fault tolerance.

9 Conclusion

The current implementation status of our DS2 core is in its final stages of testing, and tightly follows the formal operational semantics explained in this work. We are working on developing an extended version of linearizability checking scheduler for distributed systems inspired by Line-up [6] and guided by the operational semantics explained in this work. Our immediate targets are developing a front-end parsing the benchmarks we developed in Akka: Paxos [21], Chord [26], and Zab [18] for our linearizability checker, semantics-aware distributed systems automated testing, analysis, and synthesis. We are developing many use-cases [4],[9] to drive our work forward along these lines.

Acknowledgments: We acknowledge NSF grant CCF 7298529, and help by Heath French, Jarkko Savela, Zepeng Zhao, and Anushree Singh. This work is, also, supported in part by NSF Award CCF 1346756.

References

1. Akka, February 2016. http://akka.io/.
2. Pony - High Performance Actor Programming, November 2016. http://www.ponylang.org/.
3. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
4. Mohammed Al-Mahfoudh, Ganesh Gopalakrishnan, and Ryan Stutsman. Toward rigorous design of domain-specific distributed systems. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, FormaliSE '16, pages 42–48, New York, NY, USA, 2016. ACM.
5. Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, October 2014.
6. Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. *SIGPLAN Not.*, 45(6):330–340, June 2010.
7. H. Casanova. Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437, 2001.
8. Sylvan Clebsch. Pony: Co-designing a Type System and a Runtime. SPLASH 2016, New York, NY, USA, November 2016. ACM. Appeeared in SPLASH-I, Presentation.
9. DS2 official website, 2016. `http://formalverification.cs.utah.edu/ds2`, Retrieved Jan 31, 2016.

10. Philipp Haller and Heather Miller. Distributed programming via safe closure passing. In Simon Gay and Jade Alglave, editors, *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015.*, volume 203 of *EPTCS*, pages 99–107, 2015.
11. Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proceedings of the 7th Joint Conference on Modular Programming Languages*, JMLC'06, pages 4–22, Berlin, Heidelberg, 2006. Springer-Verlag.
12. Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
13. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17, 2015.
14. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
16. Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2000.
17. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, October 2009.
18. F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 245–256, June 2011.
19. Ivan Kuraj and Daniel Jackson. Exploring the role of sequential computation in distributed systems: Motivating a programming paradigm shift. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 145–164, New York, NY, USA, 2016. ACM.
20. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
21. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.
22. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
23. Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 399–414, Berkeley, CA, USA, 2014. USENIX Association.
24. Caitie McCaffrey. The verification of a distributed system. *ACM Queue*, 13(9):60:150–60:160, December 2015.
25. Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

26. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.
27. Strategy design pattern, 2017. `https://sourcemaking.com/design_patterns/strategy`, Retrieved Jan 1, 2017.
28. Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32Nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 219–234, Berlin, Heidelberg, 2012. Springer-Verlag.
29. James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. *ACM SIGPLAN Notices*, 50(6):357–368, June 2015.
30. Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *NSDI*, pages 213–228. USENIX Association, 2009.
31. Pamela Zave. How to make chord correct (using a stable base). *Computing Research Repository (CoRR)*, abs/1502.06461, 2015.