A Formal Framework for Predicting Distributed System Performance under Faults

Ziwei Zhou[1], Si Liu[2], Zhou Zhou[1], Peixin Wang[1], and Min Zhang[1]

[1] East China Normal University, China
[2] Texas A&M University, USA

Abstract. Today's distributed systems operate in complex environments that inevitably involve faults and even adversarial behaviors. Predicting their performance under such environments directly from formal designs remains a long-standing challenge. We present the first formal framework that systematically enables performance prediction of distributed systems across diverse faulty scenarios. Our framework features a fault injector together with a wide range of faults, reusable as a library, and model compositions that integrate the system and the fault injector into a unified model suitable for statistical analysis of performance properties such as throughput and latency. We formalize the framework in Maude and implement it as an automated tool, PerF. Applied to representative distributed systems, PerF accurately predicts system performance under varying fault settings, with estimations from formal designs consistent with evaluations on real deployments.

1 Introduction

Distributed systems, such as cloud databases and blockchains, form the backbone of today's digital infrastructure. Formal methods have proven highly effective in verifying the correctness of their designs [35, 49]. *Performance, however, is equally critical.* The need to predict a system's performance (e.g., throughput and latency) in realistic distributed environments, ideally before its implementation and deployment, has long been recognized by both academia [10, 15] and industry [21, 25, 51]. Such environments inevitably involve faults, including network partitions, message delays, and even Byzantine behaviors [45, 56]. In particular, Amazon Web Services has been seeking, for over a decade, a feasible way to model distributed systems and predict their performance degradation [16, 51].

State-of-the-Art. Despite attempts to address this long-standing challenge, it remains largely unresolved. First, most quantitative formal analyses of distributed systems assume a fault-free environment [15, 36, 38, 40–43, 54, 57]. Yet ignoring faults can introduce a significant gap between model-based predictions and observed system performance. For instance, two distributed transaction protocols that appear competitive under a recent fault-free analysis [41] diverge sharply once message loss is introduced (see Section 6). Second, approaches that incorporate faults are often *ad hoc* [8, 9, 26, 39, 58], manually intertwining fault behaviors with system semantics. This hinders reuse, e.g., applying the same fault to a different protocol requires rebuilding the entire model from scratch. Third, all these approaches consider only a few isolated fault types, whereas systems in practice often experience a wide range of faults that may also coexist, e.g., network partitions coinciding with malicious attacks. Therefore, existing analyses capture only a narrow fragment of the overall system performance space.

All these limitations call for a systematic formal framework that (i) provides *comprehensive fault coverage* across diverse faults encountered in practice; (ii) ensures *modularity* by decoupling individual fault modeling from system modeling to support composition, reuse, and extensibility; and (iii) enables *automatic* generation of fault-injected system models and quantitative analysis of performance properties, e.g., using statistical verifiers.

Our Solution. We present the *first* formal framework that realizes these goals. At its core lies *a library of faults* covering a wide range, including both benign faults (e.g., message loss and network partitions) and Byzantine faults (e.g., tampering and equivocation). We model each fault as a message-passing *actor system* [4], a design choice motivated by our two insights. First, virtually all distributed systems can be specified as actor systems, where nodes, such as clients and servers, communicate via asynchronous message exchanges. Modeling faults as actors allows their effects on a system to be naturally captured through message-triggered interactions between the fault injector and system actors. Second, most practically relevant faults in distributed systems can be commonly viewed as manipulations of messages between actors. For instance, a node crash corresponds to dropping all messages sent to a specific actor; a network partition can be seen as discarding all messages between two actor sets, while preserving communication within each set. This unified view not only simplifies the modeling of individual faults but also allows different fault behaviors to be seamlessly integrated into the framework for performance analysis.

How do faults, then, interact with the distributed system under analysis? We realize this interaction through *model composition*. Specifically, our framework takes as input a system model and a fault injector incorporating one or more faults, and constructs an integrated model that preserves the original system semantics while enriching it with relevant fault behaviors. To account for realistic environments in which multiple faults may arise simultaneously (e.g., a network partition alongside equivocation), we introduce *fault-behavior priority levels* for fine-grained control over their interactions. This avoids semantic inconsistencies such as delivering an equivocated message across partitions. Moreover, we design the model composition to also preserve the *absence of nondeterminism* property [5], thereby supporting *statistical model checking* analysis, a formal approach that offers stronger guarantees for quantitative results than pure simulation and scales to large distributed systems [6, 55]. To facilitate exploring performance space, our framework also allows users to configure which faults to inject and when to inject them and provides a monitor that records events of interest at runtime, based on which users can readily define performance properties. All these features are integrated into our automated tool PerF.

Our approach is formalism-independent. In this work, we instantiate it in the Maude specification language and tool [22] that has been successfully applied to a wide range of distributed systems [41, 47]. Specifically, both the distributed system under analysis and the faults are formalized as *probabilistic rewrite theories* [5]. Since performance properties, such as throughput and latency, intrinsically involve time, probabilities are primarily used to model message delays, which are sampled from certain probability distributions. The model compositions are then realized as compositions of rewrite theories.

Contributions. Overall, this work makes the following contributions:

– At the conceptual level, we address the long-standing challenge of predicting the performance of distributed systems under realistic faulty environments directly from their formal designs.
– At the technical level, we develop an actor-based formal framework in Maude, featuring a reusable library of practically relevant faults. We design a model composition that integrates the system and the fault injector managing these fault behaviors into a unified model amenable to quantitative analysis.
– At the practical level, we implement our framework as an automated tool, called PerF, that supports configurable fault injection and end-to-end performance prediction through statistical model checking.

We apply PerF to six representative distributed systems of different kinds. Experimental results demonstrate that it can accurately predict system performance under various types and combinations of faults, with model-based estimations aligning closely with empirical evaluations on real deployments.

This work also complements long-established, *post hoc* fault-injection efforts on distributed system deployments by offering *proactive* insight into system dynamics under faults, already at the design stage. See Section 7 for a discussion.

2 Preliminaries

Maude. As an executable formal specification language and analysis tool, Maude [22] has been successfully applied to a broad spectrum of distributed systems. A Maude module specifies a *rewrite theory* [48] $\mathcal{R} = (\Sigma, E, L, R)$, where:

– Σ is an algebraic *signature*, i.e., a set of *sorts*, *subsorts*, and *function symbols*;
– (Σ, E) is an *order-sorted equational logic theory* [28] specifying the system's data types, where E is a set of (possibly conditional) equations and axioms;
– L is a set of rule *labels*; and
– R is a collection of *labeled conditional rewrite rules* $[l] : t \longrightarrow t'$ `if` *cond*, with t, t' Σ-terms and $l \in L$. These rules specify the system's local transitions.

We summarize the fragment of Maude syntax used in this paper and refer the reader to [22] for details. Operators are declared by `op` f `:` $s_1 \ldots s_n$ `->` s and may have user-defined syntax, where '`_`' denotes argument positions, as in `_+_`. (Unconditional and conditional) equations and rewrite rules are introduced with the keywords `eq` and `ceq`, and `rl` and `crl`, respectively. Modules can be imported as submodules, e.g., by using the keyword `inc`. Comments start with '`***`'.

Actors. We follow Agha's message-passing *actor* paradigm [4] to specify systems. In Maude, a declaration `class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n declares an actor (or object) class C with attributes att_1 to att_n of sorts s_1 to s_n. An instance of class C is a term `<` $o : C$ `|` $att_1 : val_1, \ldots, att_n : val_n$ `>`, where o (of sort `Oid`) is the object's identifier, and val_1 to val_n are the values of the attributes. Messages exchanged between objects are terms of sort `Msg`. A system state is modeled as a term of sort `Config` and is structured as a *multiset* of objects and messages, formed via the (juxtaposition) multiset union operator `__`.

The dynamic behavior of a system is axiomatized by specifying each of its transition patterns through a rewrite rule. For instance, the rule labeled `[reply]`

```
rl [reply] :  (read(KEY) from O' to O)    < O : Server | database: DB >
          =>  < O : Server | >    (<KEY, DB[KEY]> from O to O') .
```

defines a family of transitions in which a message `read(KEY) from O' to O` sent by the client object `O'` is consumed by the server object `O`. The server then looks up its database and replies with the corresponding key-value pair via the message `<KEY, DB[KEY]> from O to O'`. Note that attributes whose values do not change, such as `database`, can be omitted in the right-hand side of a rule. Appendix A provides a Maude specification of the Two-Phase Commit protocol.

Probabilistic Rewrite Theories. Extending the original rewrite theories, *probabilistic rewrite theories* [5] can specify a broader class of systems with probabilistic behavior. In our framework, probabilities arise from message delays sampled from probability distributions *and* from probabilistic choices during fault injection, e.g., the likelihood of dropping a message. Probabilistic behavior can be modeled with rules of the form

$$[l] : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \text{ if } cond(\overrightarrow{x}) \;\; \textit{with probability} \;\; \overrightarrow{y} := \pi(\overrightarrow{x})$$

where the term t' has new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in t. The probabilistic nature of the rule stems from the probability distribution $\pi(\overrightarrow{x})$, which depends on the matching instance of $\overrightarrow{x}$, and governs the probabilistic choice of the instance of $\overrightarrow{y}$ in the term $t'(\overrightarrow{x}, \overrightarrow{y})$ according to $\pi(\overrightarrow{x})$.

Statistical Model Checking (SMC). This is a formal method for analyzing probabilistic systems against temporal logic properties, which scales to large distributed systems [6, 55]. SMC verifies whether a property, expressed in a stochastic temporal logic like QuaTEx [5], holds with a user-specified statistical confidence by performing Monte Carlo simulations of the system model. The expected value of the property is iteratively evaluated with respect to two parameters α (confidence level) and δ (error margin) until a value $\bar{v}$ is obtained such that with $(1-\alpha)$ statistical confidence, the expected value lies in the interval $[\bar{v} - \frac{\delta}{2}, \bar{v} + \frac{\delta}{2}]$.

A Maude model is suitable for SMC analysis only if it satisfies the *absence of nondeterminism* (AND) property [5]. This property requires that, for any given system state, the subsequent transition is uniquely determined, without any nondeterministic branching. We establish that the model composition, as well as the model transformation (Section 5), satisfies this guarantee. Our framework then integrates the PVeStA tool [7] for end-to-end statistical model checking of system performance properties expressed in QuaTEx formulas.

3 Framework Overview

Our formal framework composes a distributed system model with a fault injector that includes a library of fault behaviors. This yields an integrated model in which the system interacts with the faults through message passing.

Fault Library. Seven representative faults are currently included in the fault library: *message loss*, *duplication*, *delay*, *tampering*, *equivocation*, *node crash*, and *network partition*. These cover both benign and Byzantine fault behaviors commonly observed in realistic distributed environments and widely exercised in deployment-level fault-injection analyses in industrial practice [3, 12, 33, 50].

Each fault is modeled as an actor, referred to as a *fault handler* (see below and Section 4.3). This design choice is motivated by our two insights. First, modeling faults as actors allows their effects to be captured through message-triggered interactions with the system, whose nodes themselves communicate via asynchronous message passing. Second, all these faults can be viewed as manipulations of messages exchanged between actors. For example, message loss, duplication, delay, tampering, and equivocation operate on individual messages: loss drops a message between two nodes; duplication creates a copy; delay injects an abnormal latency (due to network congestion or adversarial interference); tampering modifies the message content; equivocation sends different message copies to different recipients. Other faults operate on *sets* of messages. Crashing a node can be represented as dropping all messages destined for it; a network partition corresponds to dropping all messages between two disjoint node sets, while preserving communication within each set.

This unified view not only simplifies the modeling of individual faults but also provides a common “interface” for extending the library with new faults.

Fault Injection. Injecting faults is carried out through the cooperation of three components within the fault injector: the *scheduler*, the *fault controller*, and the *fault handlers*. All interactions among these components occur internally and are thus transparent to the system. In other words, the system continues to evolve according to its original semantics; the injector merely intercepts and manipulates in-transit messages without altering the system's transition rules.

The overall workflow of our framework is as follows (also illustrated in Fig. 1):

① **Message Interception and Scheduling.** Fault injection begins when the scheduler intercepts an outgoing message *msg* from a sender node in the system, e.g., the server's reply in the example of Section 2. It then augments the message into the form `[`$gt + md$ `,` msg `,` l`]`, where $gt + md$ denotes the message's expected arrival time, with gt the current global time maintained by the scheduler and md a message delay sampled from a predefined probability distribution (e.g., lognormal [13]), and l is the label of the rewrite rule that produced the message.[3]

[3] More precisely, a preprocessing step annotates each outgoing message with the corresponding rule label prior to interception; see Section 5.

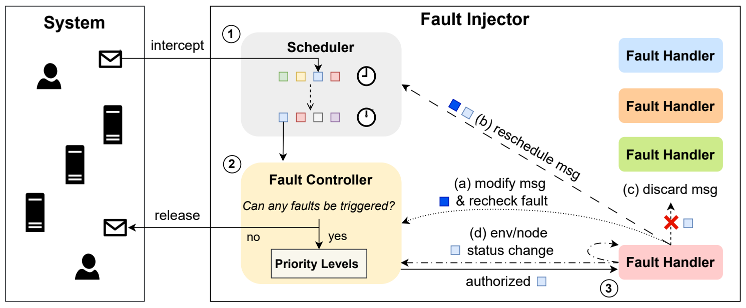

Fig. 1: The workflow of our fault-injection framework.

The scheduler maintains an ordered list of all in-transit messages, sorted by their expected arrival times. When a message reaches the head of the list, the global clock is advanced to its expected arrival time. Rather than delivering the message directly to the destination node, the scheduler hands it over to the fault controller for potential fault injection. See Section 4.1 for details.

② **Fault Control.** Upon receiving the message, the fault controller determines if any fault conditions are triggered based on its metadata (e.g., its destination, content, and source rule label). If a condition holds, the controller authorizes the corresponding fault behavior *fbhv* and passes the message to the appropriate handler via `auth([`$gt + md$`,`msg`,`l`], `$fbhv$`)`. For instance, under a network partition, *fbhv* may be `part-drop` that discards the message sent across partitions.

If multiple conditions hold simultaneously, the controller selects which fault behavior to authorize first according to a predefined *fault priority* (Section 4.2). Note that, in our framework, fault control is designed to be *fine-grained*: a single fault type may encompass multiple distinct fault behaviors. For example, network partition also includes behaviors such as `part-msg`, where partitions are triggered by a specific message that marks an "interesting" moment in the system execution (e.g., right before a leader is selected in consensus protocols).

Otherwise, the fault controller releases the message directly into the system configuration. Note that the delivered message may differ from the original one if it has been modified by a fault handler, as explained below.

③ **Fault Handling.** Once a fault behavior is authorized, the corresponding handler applies its designated operation to the message. This results in one of the following four outcomes (see Section 4.3 for details):

(a) The handler modifies the message and sends it back to the fault controller for rechecking, which may authorize another fault behavior to be injected. This arises in faults such as message tampering and equivocation.
(b) The handler produces one or more messages that must be rescheduled. For example, a delay fault adds an additional abnormal delay to the message.
(c) The handler discards the message, which may result from a message-loss fault, a network partition, or delivery to a crashed node.
(d) The message remains unchanged and is rechecked for other faults, while the environment (e.g., a network partition or recovery) or the node's status (e.g., a crash or reboot) is altered.

4 Formalizing Fault Injection

This section formalizes the three components within the fault injector, i.e., the scheduler, the fault controller, and the fault handlers, together with their interactions. The complete Maude specification is available at [1].

4.1 Scheduler

The scheduler maintains a message queue `msgQueue`, sorted by the messages' expected arrival times, and a global clock, `clock`, which is advanced based on these times. The Maude module `SCHEDULER` below presents its declaration as an actor class, along with its key operation (corresponding to Step ① in Fig. 1):[4]

```
1  mod SCHEDULER is
2    ...  *** omitted variable declarations and module importations
3
4    class Scheduler | clock: Float , msgQueue: List{Msg} .
5
6    *** intercept and schedule an outgoing msg
7    eq [MSG , L]  < sch : Scheduler | clock: GT , msgQueue: MS >
8     = < sch : Scheduler | msgQueue: insert(MS , [GT + md , MSG , L]) > .
9    *** tick global clock
10   eq tick(< sch : Scheduler | clock: GT , msgQueue: [GT',MSG,L] ; MS >  OBJS)
11    = < sch : Scheduler | clock: GT', msgQueue: MS >  OBJS  [GT',MSG,L] .
12 endm
```

The first equation (lines 7–8) inserts an outgoing message `MSG` into the scheduler's queue with its expected arrival time `GT + md`, where `md` is sampled at runtime from a certain probability distribution. Note that the source rule label `L` has been captured by a preprocessing step (Section 5). The second equation (lines 10–11) advances the global clock to the corresponding time `GT'` when the message is forwarded to the fault controller for potential manipulation.

4.2 Fault Controller

Fault control is complex as our framework incorporates a wide range of faults, some of which encompass multiple behaviors. This complexity is further amplified when several behaviors become applicable simultaneously, making it hard for the controller to determine when and how they should be injected effectively.

Prioritizing Fault Behaviors. We introduce a set of fault-behavior priority levels to enable fine-grained control over these behaviors. Based on the precondition required to trigger each behavior (see below), we organize all fault behaviors into three levels, where Level i has higher priority than Level j for $i < j$; therefore, behaviors at Level i are always triggered before those at Level j at runtime. If multiple fault behaviors at the same level become simultaneously applicable, the controller uniformly selects one to trigger, with the remaining behaviors considered in subsequent iterations. Table 1 summarizes these behaviors, along with their descriptions and assigned priority levels.

[4] For brevity, we omit variable declarations while following the Maude convention that variables are written in capital letters.

Table 1: Priority levels of fault behaviors. Level 1: time-triggered behaviors; Level 2: message-modification behaviors; Level 3: message-induced behaviors.

Behavior	Description	Level 1	Level 2	Level 3
`msg-loss`	drop an in-transit message		●	
`tampering`	modify the message content		●	
`equivocation`	send different message copies to different receivers		●	
`part-time`	network is partitioned when the global time reaches a specific point	●		
`part-msg`	partition the network right before delivering a specific message			●
`part-drop`	drop a message across partitions		●	
`recover-time`	partition is recovered when the global time reaches a specific point	●		
`recover-msg`	recover the partition right before delivering a specific message			●
`crash-time`	a node crashes when the global time reaches a specific point	●		
`crash-msg`	crash a node right before delivering a specific message			●
`crash-drop`	drop a message destined for the crashed node		●	
`reboot-time`	a node is rebooted when the global time reaches a specific point	●		
`reboot-msg`	reboot a crashed node right before delivering a specific message			●
`duplication`	create extra copies of the message			●
`abnormal-delay`	add an extra delay to the message			●

Level 1: Time-Triggered Behaviors. These behaviors are triggered purely by time.[5] As the global clock may be advanced by messages whose expected arrival times go beyond the scheduled times of these behaviors, all pending ones at this level must be applied before processing any behaviors of other levels. For example, consider a network partition (`part-time`) scheduled to occur at time t_1, but the global time has already been advanced to t_2 by a cross-partition message m, where $t_2 > t_1$. If the partition is not triggered first, m would be delivered to its destination, which is semantically incorrect, as no message should be delivered across partitions once the network partition has taken effect.

Level 2: Message-Modification Behaviors. These behaviors operate on a message while it is still mutable, i.e., modifiable or removable, assuming that all Level 1 behaviors have already been resolved. They directly alter the message's content (the behaviors `tampering` and `equivocation`) or existence (the behaviors `msg-loss`, `part-drop`, and `crash-drop`).

Level 3: Message-Induced Behaviors. These behaviors require the message to be stable, i.e., its content and destination have been finalized, and they act based on this finalized message without modifying it. Such behaviors may create new message copies (`duplication`), add an extra delay (`abnormal-delay`), or trigger message-induced changes to the system or environment (e.g., `part-msg` for partitioning the network and `reboot-msg` for rebooting a crashed node).

Formalizing Fault Control. The module `CONTROLLER` specifies the fault controller, as shown below. It imports all predefined fault handlers (Section 4.3) relevant to the user-specified faults (line 2). Here, we use message loss and network partition to illustrate how the controller operates (referring to Step ②).

```
1 mod CONTROLLER is
2   inc MSG-LOSS + PARTITION .  *** imported according to the user config
```

[5] Our framework allows users to specify the time points at which these behaviors are triggered; see Appendix A.

```
3
4   class Controller | fbhvs : List{Bhv}, priority : Map{Bhv, Nat} .
5
6   ceq [GT, MSG, L]  < ml : MsgLoss | >  < pt : Partition | >
7       < ctrl : Controller | fbhvs: BS, priority: P >
8     = < ml : MsgLoss | >  < pt : Partition | >  < ctrl : Controller | >
9       (if isSatisfied(BS,[GT,MSG,L], < ml : MsgLoss | > < pt : Partition | >)
10         then auth([GT, MSG, L], B)  *** fault behavior authorized
11         else MSG fi)  *** msg released into the system config
12    if B := topBhv(BS,P,[GT,MSG,L],< ml : MsgLoss | > < pt : Partition | >) .
13
14  *** check whether message loss can be triggered
15  eq isSatisfied(msg-loss, [GT, CONTENT from O to O', L],
16       < ml : MsgLoss | lossReceivers: OS, lossRules: LS, lossRate: R > HDLS)
17   = O' in OS and L in LS and rand < R .
18
19  *** check whether network partition can be triggered by time
20  eq isSatisfied(part-time, [GT, MSG, L],
21        < pt : Partition | status: S, occurTime: T > HDLS)
22   = GT >= T and S == healthy .
23  ...
24 endm
```

The controller is modeled as an actor class with two attributes (line 4): `fbhvs`, which specifies the list of all fault behaviors to be injected, and `priority`, which maps each fault behavior to a natural number indicating its priority level. Both attributes are initialized according to the user configuration; see Appendix A for an example with the message-loss and network-partition fault behaviors.

The controller's operation is defined by a conditional equation (lines 6–12), where the controller evaluates all fault behaviors in `fbhvs` via the predicate `isSatisfied` to determine which behaviors are eligible for triggering (line 9). For example, the `msg-loss` behavior is eligible when the receiver `O'` belongs to the designated set of receivers `OS` whose incoming messages may be dropped, the source rule label matches (by the predicate `L in LS`), which identifies a specific execution point relevant to that receiver, and a probabilistic draw, denoted by the random number `rand`, falls below the user-specified loss rate `R` (lines 15–17). The `part-time` behavior becomes eligible when the global time exceeds the configured trigger time of the network partition (i.e., `GT >= T`) and the current status of the network is `healthy` (lines 20–22).

When multiple behaviors are eligible, the controller selects the highest-priority one based on the predefined levels `P` via the function `topBhv` (line 12). For instance, `part-time` is chosen over `msg-loss` when both apply. The controller then forwards the message, along with the selected behavior to the corresponding handler (line 10), or releases it into the system configuration otherwise (line 11).

4.3 Fault Handlers

A fault handler is the component for executing the designated fault behavior once it has been authorized by the controller. All handlers are modeled as actors, each with its operation defined in its own module. Depending on the specific behavior, a handler may produce one of the four outcomes, as described in Step ③.

The module `PARTITION` below exemplifies how the fault handler manages three network-partition behaviors of different levels. For brevity, only the attributes of the `Partition` class relevant to these behaviors are shown.

```
 1 mod PARTITION is
 2   class Partition | status : NetworkStatus, allNodes : Set{Oid},
 3                     parts : Partitions, occurTime : Float, ...
 4
 5   *** Level 1: a network partition is triggered by a specific time
 6   eq auth([GT, MSG, L], part-time)
 7      < pt : Partition | status: healthy, parts: [OS1 | OS2], allNodes: OS >
 8    = < pt : Partition | status: partitioned,
 9            parts: (if OS1 == empty then randomPart(OS) else [OS1 | OS2] fi) >
10      [GT, MSG, L] .
11
12   *** Level 2: a message is dropped across partitions
13   eq auth([GT,MSG,L],part-drop) < pt : Partition | > = < pt : Partition | > .
14
15   *** Level 3: a network partition is triggered by a specific message
16   eq auth([GT, MSG, L], part-msg)
17      < pt : Partition | status: healthy, occurTime: T,
18                         allNodes: OS, parts: [OS1 | OS2] >
19    = < pt : Partition | status: partitioned, occurTime: GT,
20            parts: (if OS1 == empty then randomPart(OS) else [OS1 | OS2] fi) >
21      [GT, MSG, L] .
22   ...
23 endm
```

The network is partitioned either at a specific time point (lines 6–10) or upon the arrival of a designated message (lines 17–22). A partition is created by switching the network status from `healthy` to `partitioned`. The partition sets are either preconfigured, e.g., `[OS1 | OS2]` denotes two disjoint node sets `OS1` and `OS2` that are instantiated in the initial state, or generated probabilistically via the function `randomPart`. Once the partition is applied, the message is returned to the controller for further fault processing, as discussed earlier. This corresponds to the outcome (d) in Step ③.

During a network partition, any message sent across the two partitions is dropped. This is signaled by `part-drop` and realized by the equation at line 13, corresponding to the outcome (c) in Step ③.

4.4 Fault Injector

The fault injector is formed by importing the two modules `SCHEDULER` and `CONTROLLER` introduced earlier, with the latter also importing all user-specified fault handlers such as `MSG-LOSS` and `PARTITION`.

```
mod FAULT-INJECTOR is    inc SCHEDULER + CONTROLLER .    endm
```

As we will see in the next section, our framework automates the fault injection for quantitative performance analysis, including statistical model checking (SMC). To ensure that the composed system model with the fault injector is suitable for SMC, we prove that it satisfies the *absence of nondeterminism* (AND) property (Section 2). The proof is given in Appendix B.

Theorem 1. *The composed system model with fault injection guarantees AND.*

5 Automating Fault-Injection Analysis

We have implemented our fault-injection framework in Maude and incorporated several additional components into an automated tool, PerF (available at [1]),

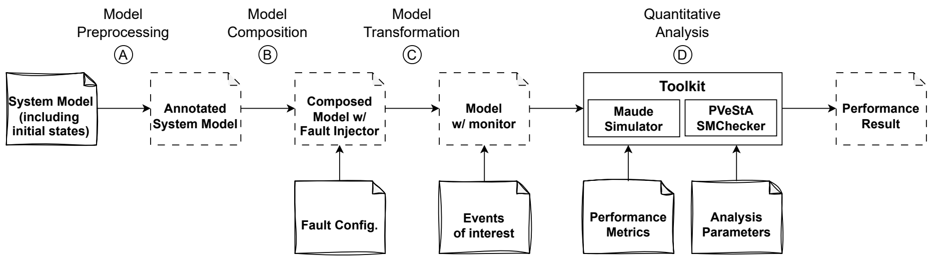

Fig. 2: The pipeline of PerF. Sketched files are provided by users, while dashed files are generated automatically by the tool. The composed model with the fault injector in Step Ⓑ corresponds to Fig. 1.

to enable end-to-end performance prediction. The overall tool pipeline comprises four steps, as shown in Fig. 2:

Ⓐ **Model Preprocessing.** The tool annotates the user-provided system model, typically nondeterministic and untimed, by attaching to each outgoing message *msg* the label l of the rule that generates it, yielding `[`msg`,`l`]`.

Ⓑ **Model Composition.** This step corresponds to the discussion in Section 4, where the annotated system model is composed with the fault injector that consists of the scheduler, the fault controller, and the relevant fault handlers. Before injecting any faults, the scheduler augments each message with a normal delay that is sampled at runtime from a user-specified probability distribution. Our tool provides a collection of network latency distributions, including lognormal, Weibull, and exponential, that reflect the characteristics of real data centers [13]. It also offers an interface through which users can specify both the types of faults to inject (e.g., message loss) and their injection parameters (e.g., a 10% loss rate for acknowledgments from followers to the leader).

Ⓒ **Model Transformation.** The composed model is then transformed into one equipped with a monitor that timestamps events during execution, e.g., the times at which a transaction is issued and committed. We realize this monitoring mechanism by extending the approach introduced in [41] to support logging multiple events of interest, allowing users to readily define and compute performance metrics such as throughput and abort rate over these events.

Ⓓ **Quantitative Analysis.** The toolkit performs quantitative analysis using either Maude's built-in simulator or the PVeStA statistical model checker [7]. In both modes, simulations are produced by executing the transformed model from the initial state. For statistical model checking, users additionally provide performance properties expressed as QuaTEx formulas, along with experimental parameters (e.g., the statistical confidence level). The analysis output is either the raw simulation result or the expected value of a given QuaTEx formula.

We illustrate these steps using the Two-Phase Commit protocol, which is deferred to Appendix A due to space limitations. Note that, in addition to Theorem 1, we prove that the model transformation in Step Ⓒ also preserves AND, making the resulting model suitable for SMC analysis; see Appendix B.

Table 2: Summary of case studies. †: Maude specifications obtained from the public repositories; ‡: implementations developed in this work.

System	Kind	Model LoC	Impl. LoC	Injected Fault	Deployment Platform
2PC with CTP	atomic commitment	314	354‡	message loss	CloudLab Utah (d6515)
Raft	consensus	536	1400	node crash	CloudLab Utah (xl170)
PowerDNS	name resolution	2200†	4887	delayed messages	DNS testbed [39]
Cassandra Quorum	consistency	465	1402‡	network partition	CloudLab Clemson
RAMP & OPW	concurrency control	2523†	7977	message loss	Tencent Cloud
Fast-HotStuff	Byzantine consensus	304	4247	equiv. & network part.	Emulab

6 Case Studies

We apply PerF to six distributed systems of different kinds: (1) the widely used atomic commitment protocol 2PC, which integrates the Cooperative Termination Protocol (CTP) [14] to mitigate its blocking issues during failures; (2) the consensus protocol Raft [52], which achieves fault-tolerant log replication; (3) the authoritative DNS server PowerDNS [53], which is widely deployed in production networks to ensure reliable name resolution; (4) the quorum-based consistency protocol of Apache Cassandra [2], which balances data consistency and availability via read/write quorums; (5) the RAMP distributed transaction protocol [11], along with its optimization, which has been layered atop Facebook's TAO [20]; and (6) the Byzantine fault-tolerant consensus protocol Fast-HotStuff [31], which improves the classical HotStuff design for low-latency agreement.

Table 2 summarizes these case studies, including the system types, modeling and implementation efforts, and the injected faults considered in our analysis.

6.1 Experimental Setup and Rationale

Setup. We predict system performance from the formal models under different network latency distributions, including lognormal and exponential. We employ a cluster of Emulab machines [59] to parallelize the PVeStA analysis, each with a 2.4 GHz quad-core Xeon processor and 12 GB of RAM. The statistical confidence level α is set to 95%, and the error margin δ to 0.01. All three steps prior to SMC are performed on a single machine, each completing instantly.

We deploy the actual systems across multiple platforms (see Table 2), including CloudLab (and its different clusters) and Tencent Cloud, each exhibiting distinct characteristics of real-world distributed environments. For each system, we implement dedicated fault injectors that correspond to our model-based analysis.

Evaluation Rationale. Our evaluation rationale follows prior work [41, 44]: we consider a model-based prediction as accurate if it closely mirrors the observed behaviors in the deployment evaluation in terms of overall curve trends, up to a certain scaling factor. This criterion is motivated by the fact that model-based predictions seldom align numerically with empirical results.[6] In a probabilistic

[6] Even the same implementation evaluated on different platforms rarely yield numerically identical results due to factors like distinct network latency distributions [41].

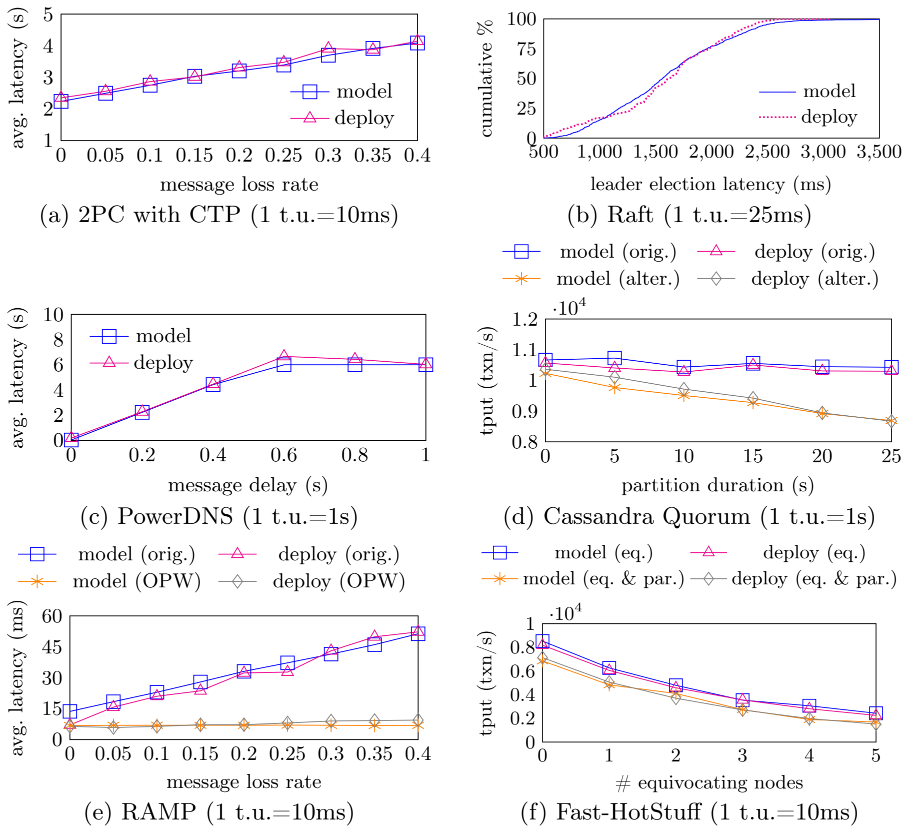

Fig. 3: Model-based performance predictions vs. deployment-level evaluations across six distributed systems and their variants. The scaling factor 1 t.u.= T ms means that one time unit in the model corresponds to T ms in the deployment.

model, quantitative values are inherently tied to an abstract unit of time. This unit only approximates the real timing observed in deployed systems, and the corresponding scaling factor remains unknown at the design stage.

6.2 Experimental Results

Summary. Our experimental results are shown in Figure 3. Overall, across all six case studies, we demonstrate the predictive power of our framework by showing that the model-based performance estimations, specifically throughput and latency, align closely with evaluations of the deployed systems under various types of faults. These include message loss (Figure 3(a) and (e)), node crashes (Figure 3(b)), delayed messages (Figure 3(c)), network partitions (Figure 3(d)), and equivocation combined with network partitions (Figure 3(f)).

In particular, our analysis reveals that although some systems were previously deemed competitive with their variants, substantial performance degradation can occur in either the original design or its optimization once faults are introduced. This highlights the strength of PerF: it enables the exploration of a much broader spectrum of system behaviors already at the design stage. We discuss one such case, namely RAMP and its optimization, along with a mixed-fault analysis of Fast-HotStuff; the remaining ones are deferred to Appendix C.

RAMP. The Read Atomic Multi-Partition (RAMP) system [11] delivers performant transactions and the read atomicity isolation through efficient concurrency control. In addition to the original design, its developers introduced a faster variant with one-phase writes (OPW) that relaxes certain consistency guarantees.

A recent study [41] found both designs competitive, with the original design incurring only marginally higher latency. However, this conclusion holds only under fault-free conditions. Using PerF, we reveal that as message loss increases, the original design exhibits significantly higher latency than the optimization OPW. Deployment-level results corroborate our model-based predictions, as shown in Figure 3(e). Since message loss is inevitable in practical deployments and often intensifies under high network contention, this finding can help practitioners make more informed choices between alternative designs.

Fast-HotStuff. Byzantine fault-tolerant consensus plays a pivotal role in modern blockchain systems. The recently proposed Fast-HotStuff consensus protocol [31] offers improved performance and greater robustness against forking attacks compared to the state of the art.

We use PerF to predict how Fast-HotStuff behaves under equivocation attacks [12, 30], where adversaries propose conflicting blocks that cause ambiguity among participants, leading to forks and delayed consensus progress. As shown in Figure 3(f), increasing the number of equivocating nodes causes throughput to decline. Moreover, we assess Fast-HotStuff's performance when equivocation and (benign yet prevalent) network partitions coexist. The results indicate a moderate decrease in throughput, an expected yet reassuring outcome, as the degradation remains limited. All our model-based predictions align closely with deployment-level evaluations conducted on a real cluster.

Notably, this mixed-fault experiment extends beyond the evaluations performed in the original study [31], providing a more comprehensive understanding of Fast-HotStuff's behavior under more realistic scenarios. Additionally, this case study demonstrates the modularity of our framework, which enables rapid exploration of a system's performance space under complex fault combinations.

7 Related Work

Formal Quantitative Analysis. We have already discussed in Section 1 the limitations of existing approaches to predicting distributed system performance from formal designs. They either ignore faults entirely, couple them directly with system semantics, or examine only a few isolated fault types. All these works are based on Maude [22] or TLA+ [35], which is largely because both formalisms provide high expressiveness for modeling large-scale distributed systems. On the other hand, a number of automata-based quantitative frameworks, such as Uppaal-SMC [17], PRISM [34], and Modest [29], support statistical or probabilistic model checking over timed or stochastic automata. Nevertheless, applying these frameworks to our setting remains highly challenging, as certain system characteristics are difficult to capture within automata-based models, e.g., the dynamic creation of actors or messages that can lead to an unbounded population of such components. Yet this is common in both normal (e.g., the joining of new nodes) and faulty scenarios (e.g., message duplication).

Performal [61] advances the formal verification of distributed system performance by providing worst-case latency bounds; yet such bounds are generally unattainable, e.g., in the presence of Byzantine behaviors. Network calculus [24] provides a mathematical foundation for reasoning about networked system performance and has been applied to establish latency guarantees [32, 37]. These approaches address an orthogonal problem; we aim instead to capture the concrete system dynamics under faults, covering a wide range of quantitative metrics beyond latency, e.g., those expressible in QuaTEx (see examples in [5, 41]).

Posterior Fault Injection. Recent years have witnessed a growing number of deployment-level fault injectors for assessing the reliability and resilience of distributed systems. Academic advances include CoFI [18] for exposing network-partition failures, CrashFuzz [27] for uncovering crash-recovery bugs, Chronos [19] for detecting timeout bugs, Phoenix [46] for revealing blockchain resilience issues, and ByzzFuzz [60] for triggering Byzantine consensus bugs. Fault-injection techniques have also been widely adopted in production, exemplified by Jepsen [33], Netflix's Chaos Monkey [50], AWS's FIS [3], and Facebook's Twins [12].

Compared with these approaches, our framework enables early and proactive performance exploration during the design stage, rather than being reactive and applicable only after system implementation. Moreover, our work focuses on system performance, quantifying how systems behave under diverse faults, whereas most existing approaches concentrate on correctness and reliability issues and on how to trigger them. It is also worth noting that, although various benchmarks exist across different domains for evaluating system performance, e.g., YCSB [23] for cloud database systems, they do not incorporate faults and therefore can only explore a limited portion of the overall performance space.

8 Conclusion

We have presented a formal framework and tool for predicting distributed system performance under fault conditions. Built on Maude, our framework introduces semantics-preserving model compositions that seamlessly integrate system and fault models into a unified representation suitable for statistical analysis. Case studies demonstrate that the framework accurately captures the performance impact of diverse and co-existing faults, aligning with empirical observations.

This work paves the way for rapid and comprehensive exploration of the system performance space at an early design stage. When paired with deployment-level fault injection, it offers a complementary means to better understand and improve the performance robustness of future distributed systems.

References

1. Artifact for "A Formal Framework for Predicting Distributed System Performance under Faults". `https://github.com/ZooWagon/PerF` (2025)
2. Apache Cassandra: Open Source NoSQL Database (Nov 2025), `https://cassandra.apache.org/`
3. AWS Fault Injection Service (Nov 2025), `https://aws.amazon.com/fis/`
4. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
5. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. **153**(2) (2006)
6. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1) (2018)
7. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: CALCO'11. LNCS, vol. 6859, pp. 386–392. Springer (2011)
8. AlTurki, M.A., Kanovich, M.I., Kirigin, T.B., Nigam, V., Scedrov, A., Talcott, C.L.: Statistical model checking of distance fraud attacks on the Hancke-Kuhn family of protocols. In: CPS-SPC@CCS 2018. pp. 60–71. ACM (2018)
9. Alturki, M.A., Rosu, G.: Statistical model checking of RANDAO's resilience to pre-computed reveal strategies. In: Formal Methods. FM 2019 International Workshops. LNCS, vol. 12232, pp. 337–349. Springer (2019)
10. Alur, R., Henzinger, T.A., Vardi, M.Y.: Theory in practice for system design and verification. ACM SIGLOG News **2**(1) (2015)
11. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with RAMP transactions. ACM Trans. Database Syst. **41**(3), 15:1–15:45 (2016)
12. Bano, S., Sonnino, A., Chursin, A., Perelman, D., Li, Z., Ching, A., Malkhi, D.: Twins: BFT Systems Made Robust. In: OPODIS '21. Leibniz International Proceedings in Informatics (LIPIcs), vol. 217, pp. 7:1–7:29 (2022)
13. Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: IMC'10. pp. 267–280. ACM (2010)
14. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
15. Bobba, R., Grov, J., Gupta, I., Liu, S., Meseguer, J., Ölveczky, P.C., Skeirik, S.: Survivability: design, formal modeling, and validation of cloud storage systems using Maude. Assured cloud computing pp. 10–48 (2018)
16. Brooker, M.: Fifteen years of formal methods at AWS. TLA+ Conference (2024), `https://conf.tlapl.us/2024/MarcBrooker-FifteenYearsOfTLAPlus.pdf`
17. Bulychev, P.E., David, A., Larsen, K.G., Mikucionis, M., Poulsen, D.B., Legay, A., Wang, Z.: UPPAAL-SMC: statistical model checking for priced timed automata. In: QAPL'12. EPTCS, vol. 85, pp. 1–16 (2012)
18. Chen, H., Dou, W., Wang, D., Qin, F.: CoFI: consistency-guided fault injection for cloud systems. In: ASE '20. pp. 536–547 (2020)
19. Chen, Y., Ma, F., Zhou, Y., Gu, M., Liao, Q., Jiang, Y.: Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay. In: IEEE Symposium on Security and Privacy (SP). IEEE Computer Society (2024)
20. Cheng, A., Shi, X., Pan, L., Simpson, A., Wheaton, N., Lawande, S., Bronson, N., Bailis, P., Crooks, N., Stoica, I.: RAMP-TAO: layering atomic transactions on facebook's online TAO data store. Proc. VLDB Endow. **14**(12), 3014–3027 (2021)

21. Chuat, L., Legner, M., Basin, D.A., Hausheer, D., Hitz, S., Müller, P., Perrig, A.: The Complete Guide to SCION - From Design Principles to Formal Verification. Information Security and Cryptography, Springer (2022)
22. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
23. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC 2010. pp. 143–154. ACM (2010)
24. Cruz, R.L.: A calculus for network delay, part II: network analysis. IEEE Trans. Inf. Theory **37**(1), 132–141 (1991)
25. Davis, A.J.J.: Are we serious about using TLA+ for statistical properties? TLA+ Community Event (2025), `https://conf.tlapl.us/2025-etaps/davis-slides.pdf`
26. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: FASE 2012. LNCS, vol. 7212, pp. 78–93. Springer (2012)
27. Gao, Y., Dou, W., Wang, D., Feng, W., Wei, J., Zhong, H., Huang, T.: Coverage guided fault injection for cloud systems. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 2211–2223 (2023)
28. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. **105**(2), 217–273 (1992)
29. Hartmanns, A., Hermanns, H.: The Modest toolset: An integrated environment for quantitative modelling and verification. In: Proc. TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer (2014)
30. Jaffe, A., Moscibroda, T., Sen, S.: On the price of equivocation in byzantine agreement. In: PODC '12. p. 309–318. ACM (2012)
31. Jalalzai, M.M., Niu, J., Feng, C., Gai, F.: Fast-HotStuff: A fast and robust BFT protocol for blockchains. IEEE Transactions on Dependable and Secure Computing pp. 1–17 (2023)
32. Jang, K., Sherry, J., Ballani, H., Moncaster, T.: Silo: Predictable message latency in the cloud. In: SIGCOMM '15. p. 435–448. ACM (2015)
33. Jepsen: A framework for distributed systems verification, with fault injection (Nov 2025), `http://jepsen.io/`
34. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV'11. LNCS, vol. 6806, pp. 585–591. Springer (2011)
35. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
36. Liang, L., Liu, S.: Exploring design alternatives for replicated RAMP transactions using Maude. In: TASE '21. pp. 111–118. IEEE (2021)
37. Liebeherr, J., Burchard, A., Ciucu, F.: Delay bounds in communication networks with heavy-tailed and self-similar traffic. IEEE Trans. Inf. Theory **58**(2), 1010–1024 (2012)
38. Liu, S.: All in one: Design, verification, and implementation of SNOW-optimal read atomic transactions. ACM Trans. Softw. Eng. Methodol. **31**(3) (2022)
39. Liu, S., Duan, H., Heimes, L., Bearzi, M., Vieli, J., Basin, D., Perrig, A.: A formal framework for end-to-end DNS resolution. In: SIGCOMM '23. p. 932–949. ACM (2023)
40. Liu, S., Ganhotra, J., Rahman, M.R., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. Leibniz Trans. Embed. Syst. **4**(1), 03:1–03:26 (2017)

41. Liu, S., Meseguer, J., Ölveczky, P.C., Zhang, M., Basin, D.A.: Bridging the semantic gap between qualitative and quantitative models of distributed systems. Proc. ACM Program. Lang. **6**(OOPSLA2), 315–344 (2022)
42. Liu, S., Ölveczky, P.C., Wang, Q., Gupta, I., Meseguer, J.: Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. Formal Aspects Comput. **31**(5), 503–540 (2019)
43. Liu, S., Ölveczky, P.C., Wang, Q., Meseguer, J.: Formal modeling and analysis of the Walter transactional data store. In: WRLA 2018. LNCS, vol. 11152, pp. 136–152. Springer (2018)
44. Liu, S., Sandur, A., Meseguer, J., Ölveczky, P.C., Wang, Q.: Generating correct-by-construction distributed implementations from formal Maude designs. In: NFM'20. LNCS, vol. 12229. Springer (2020)
45. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
46. Ma, F., Chen, Y., Zhou, Y., Sun, J., Su, Z., Jiang, Y., Sun, J., Li, H.: Phoenix: Detect and locate resilience issues in blockchain via context-sensitive chaos. In: CCS '23. p. 1182–1196. ACM (2023)
47. Maude: Applications. `https://maude.cs.illinois.edu/wiki/Applications` (Nov 2025)
48. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
49. Meseguer, J.: Capturing system designs with formal executable specifications. In: FASE 2025. LNCS, vol. 15693, pp. 1–32. Springer (2025)
50. Netflix: Chaos Monkey (Nov 2025), `https://netflix.github.io/chaosmonkey/`
51. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Communications of the ACM (2015)
52. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: USENIX ATC'14. p. 305–320. USENIX Association (2014)
53. PowerDNS: PowerDNS: a faster, safer, and more secure internet (Nov 2025), `https://www.powerdns.com/`
54. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: QMaude: Quantitative specification and verification in rewriting logic. In: FM 2023. LNCS, vol. 14000, pp. 240–259. Springer (2023)
55. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: CAV '05. LNCS, vol. 3576. Springer (2005)
56. Tanenbaum, A.S., van Steen, M.: Distributed systems - principles and paradigms, 2nd Edition. Pearson Education (2007)
57. Vanlightly, J., Kuppe, M.: Obtaining statistical properties via TLC simulation. TLA+ Conference (2022), `https://conf.tlapl.us/2022/JackMarkusTLA+Statistics.pdf`
58. Weghorn, T., Liu, S., Sprenger, C., Perrig, A., Basin, D.A.: N-Tube: Formally verified secure bandwidth reservation in path-aware Internet architectures. In: CSF'22. pp. 147–162. IEEE (2022)
59. White, B., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: OSDI'02. USENIX Association (2002)
60. Winter, L.N., Buse, F., de Graaf, D., von Gleissenthall, K., Ozkan, B.K.: Randomized testing of Byzantine fault tolerant algorithms. Proc. ACM Program. Lang. **7**(OOPSLA1), 757–788 (2023)
61. Zhang, T.N., Sharma, U., Kapritsos, M.: Performal: Formal verification of latency properties for distributed systems. Proc. ACM Program. Lang. **7**(PLDI) (2023)

A The PerF Tool

A.1 Running Example: The Two-Phase Commit Protocol

Two-Phase Commit (2PC) is a distributed atomic commitment protocol that ensures all participants in a transaction either commit or abort. It proceeds in two phases. In the PREPARE phase, the coordinator sends a prepare request to all participants (or cohorts) and waits for their votes. Each cohort checks whether it can commit, e.g., by verifying local constraints, and replies with either *yes* or *no*. In the COMMIT phase, if all cohorts vote *yes*, the coordinator sends a commit message; otherwise, it propagates an abort. Cohorts then log the final decision.

Note that this is a slightly simplified version of 2PC, in which a cohort is not required to acknowledge to the coordinator that it has received the decision. We use this simplified version here to focus on illustrating PerF. In our case study, however, we consider the full integration of 2PC with the Cooperative Termination Protocol (CTP) [14] to resolve the inherent blocking issue of 2PC.

The Maude module `2PC` below specifies the protocol. For brevity, we omit some auxiliary function definitions.

```
mod 2PC is ...
  vars O O' : Oid .  var OS : Oids .  var P : Proposal .  vars V V' : Bool .
  vars PS PS' : Proposals . vars RS VS : Map{Proposal,Bool} .

  class Coord  | proposals : Proposals,   cohorts : Oids,
                 waiting   : Oids,        results : Map{Proposal,Bool} .
  class Cohort | votes : Map{Proposal,Bool} .

  op propagate_from_to_: Proposal Oid Oids -> Msgs .
  eq propagate P from O to (O' OS) = (P from O to O')
                                     (propagate P from O to OS) .
  eq propagate P from O to empty = null .

 crl [start] :  (start to O)
                < O : Coord | proposals: P ; PS, waiting: empty,
                              cohorts: OS, results: RS >
            =>  < O : Coord | waiting: OS, results: insert(P,true,RS) >
                (propagate P from O to OS) if not exist(RS,P) .

  rl [vote] : (P from O' to O)  < O : Cohort | votes: (VS, P |-> V) >
            =>  < O : Cohort | >  (vote(P,V) from O to O') .

  rl [collect] : (vote(P,V') from O' to O)
                  < O : Coord | waiting: (O' OS), results: (RS, P |-> V) >
      =>  < O : Coord | waiting: OS, results: (RS, P |-> (V and V')) > .

  rl [decision] : < O : Coord | proposals: P ; PS, waiting: empty,
                                cohorts: OS, results: (RS, P |-> V) >
            =>  < O : Coord | proposals: PS >  (start to O)
                (propagate decision(P,V) from O to OS) .

  rl [log] : (decision(P,V) from O' to O)
              < O : Cohort | votes: (VS, P |-> V') >
            =>  < O : Cohort | votes: (VS, P |-> V) > .
endm
```

The coordinator initiates the first proposal `P` from the pool `proposals` by propagating it to the cohorts `OS` (rule `[start]`). Each cohort responds with its locally stored, pre-determined vote for the requested proposal, retrieved via a key–value lookup (rule `[vote]`). Upon receiving a vote, the coordinator updates the aggregated decisions (through logical conjunction of the newly received and previously collected votes), removes the responder from its waiting list, and continues awaiting the remaining responses (rule `[collect]`). Once all votes have been collected (i.e., the waiting list becomes `empty`), the coordinator propagates the final decision and proceeds to issue the next proposal in the pool by removing the current one (rule `[decision]`). Finally, the cohorts record the decision (rule `[log]`). Note that, for example, the term `propagate P from O to OS` (line 18) reduces to a set of messages sent from the coordinator to the cohorts, as defined by the equations (lines 9–12). Note also that each voting process is triggered by a self message `start` (lines 14 and 29).

The following Maude module `INIT-2PC` specifies an initial state, where one coordinator and two cohorts (each initialized with pre-defined votes) participate in two proposals. The module `2PC` described earlier is imported at line 2.

```
1  mod INIT-2PC is
2    inc 2PC .
3
4    ops c ch1 ch2 -> Oid .  ops p1 p2 : -> Proposal .
5
6    op initState : -> Config .
7    eq initState = (start to c)
8                   < c : Coord | proposals: p1 ; p2, cohorts: ch1 ch2,
9                                 waiting: empty, results: empty >
10                  < ch1 : Cohort | votes: p1 |-> true, p2 |-> false >
11                  < ch2 : Cohort | votes: p1 |-> true, p2 |-> true > .
12 endm
```

A.2 How PerF Works

This section illustrates PerF's workflow, depicted in Fig. 2, using 2PC as a running example. It consists of four steps: *model preprocessing*, *model composition,model transformation*, and *quantitative analysis*.

Model Preprocessing. In this step, PerF annotates each outgoing message in the user-provided system model with its *source rule label*. To give users greater flexibility when modeling distributed systems, following prior work [41], PerF also equips each object-triggered rule with an `eagerEnable` equation to ensure its immediate execution once enabled.

Attaching Source Rule Labels. Whenever a rewrite rule generates messages, PerF rewrites that rule by wrapping the message-generation fragment with a labeling function. This ensures that every outgoing message is annotated with the label of the rule that created it, enabling the controller to determine whether specific fault behaviors should be activated.

The following Maude specification illustrates how the rule `[start]` in the module `2PC` is preprocessed:

```
crl [start.p] : (start to O)     *** rule label changed
                < O : Coord | proposals: P ; PS, waiting: empty,
                              cohorts: OS, results: RS >
            =>  < O : Coord | waiting: OS, results: insert(P,true,RS) >
                attachLabel('start.p, (propagate P from O to OS))
                if not exist(RS,P) .
```

```

  *** auxiliary function that attaches the rule label
  op attachLabel : Qid Msgs -> LabelMsgs .
  eq attachLabel(L,MSG MSGS) = [MSG,L] attachLabel(L,MSGS) .
  eq attachLabel(L,null) = null .
```

Eagerness of Object-Triggered Rules. A user-provided system model may include rules that are not triggered by messages; we refer to them as *object-triggered rules*. Such rules must be applied immediately once their enabling conditions become true; otherwise, nondeterministic choices could arise in firing rules. To enforce this semantics, the preprocessing step generates an `eagerEnable` equation for every object-triggered rule. This equation evaluates to `true` whenever the left-hand side of that rule matches the current configuration.

The following example illustrates this construction using the rule `[decision]`:

```
  rl [decision.p] : < O : Coord | proposals: P ; PS, waiting: empty,
                                 cohorts: OS, results: (RS, P |-> V) >
        =>  < O : Coord | proposals: PS >
        attachLabel('decision.p, (start to O) (propagate decision(P,V) from O to OS)) .

  *** eager enabling for object-triggered rule
  eq eagerEnabled(< O : Coord | proposals: P ; PS, waiting: empty,
                              cohorts: OS, results: (RS, P |-> V) > OBJS) = true .
```

In addition, the scheduler's "tick" equation is strengthened with a guard ensuring that no object-triggered rule is eagerly enabled, as shown below. This prevents the scheduler from releasing messages while an object-triggered rule is ready to fire, which is essential for preserving AND (i.e., Theorem 1).

```
  *** guarded tick: apply only when no object-triggered rule is eagerly
     enabled
 ceq tick(< sch : Scheduler | clock: GT, msgQueue: [GT',MSG,L] ; MS >  OBJS)
   = < sch : Scheduler | clock: GT', msgQueue: MS >  OBJS  [GT',MSG,L] .
     if not eagerEnable(OBJS) .
```

Model Composition. In this step, PerF composes the user-provided system model with the fault injector. Users specify which fault behaviors to inject, together with their configuration parameters such as target objects, source rule labels, probabilities, and timing, through a dedicated interface module.

The following module illustrates how users can configure the faults *message loss* and *network partition*.

```
mod 2PC-FAULT-CONFIG is
  inc FAULT-INJECTOR + INIT-2PC .

  *** injected fault behaviors
  eq injectedFaultBehaviors = msgloss ; part-time ;
                              part-drop ; recover-time .

  *** configuring message loss
  eq msgLossRate = 0.3 .
  eq msgLossRules = 'decision .
  eq msgLossReceivers = ch2 .

  *** configuring network partition
  eq partitionOccurTime = 5.0 .
  eq partitionDuration = 20.0 .
  eq partitionAllNodes = (c ch1 ch2) .
  eq partitions = [c ch1 | ch2] .
endm
```

Note that we choose to drop messages at the `[decision]` rule (line 10), i.e., the coordinator's decision message to a cohort. This prevents blocking in this simplified version of 2PC and keeps the example focused on illustrating PerF.

The module `INIT-2PC-FAULT` then imports this interface module, together with the module `2PC-FAULT`, which composes the system module `2PC` with the fault-injector module `FAULT-INJECTOR` (Section 4.4).[7]

```
mod INIT-2PC-FAULT is
  inc 2PC-FAULT-CONFIG + 2PC-FAULT .

  op initStateFault : -> Config .
  eq initStateFault = initState  *** defined in the module INIT-2PC
        < sch : Scheduler | clock: 0.0, msgQueue: nil >
        < ctrl : Controller | fbhvs: injectedFaultBehaviors,
                priority: (msgloss |-> 2, part-time |-> 1,
                           part-drop |-> 2, recover-time |-> 1) >
        < ml : MsgLoss | lossRate: msgLossRate, lossRules: msgLossRules,
                         lossReceivers: msgLossReceivers >
        < pt : Partition | status: healthy, allNodes: partitionAllNodes,
                           parts: partitions,
                           occurTime: partitionOccurTime,
                           duration: partitionDuration > .
endm
```

Model Transformation. This step equips the composed system model with a monitor object `< m : Monitor | events:` ES_1`@`T_1 `; ... ;` ES_n`@`T_n `>`, which records the set of user-specified events (e.g., ES_1) specified by the user together with their corresponding global timestamps (e.g., T_1) at runtime. We realize this monitoring mechanism by extending prior work [41] to support recording multiple events occurring at the same timestamp.

```
  sorts Event Events TimedEvent TimedEvents .
  subsort Event < Events .    subsort TimedEvent < TimedEvents .

  op empty : -> Events [ctor] .
  op __ : Events Events -> Events [ctor assoc comm id: empty] .
  op empty : -> TimedEvents [ctor] .
  op _;_ : TimedEvents TimedEvents -> TimedEvents [ctor assoc id: empty] .
  op _@_ : Events Float -> TimedEvent [ctor]

  class Monitor | events: TimedEvents .

  *** user interface
  sorts Pair EventMap .    subsort Pair < EventMap .

  op [_,_] : Qid Events -> Pair [ctor] .
  op none : -> EventMap [ctor] .
  op _;;_ : EventMap EventMap -> EventMap [ctor comm assoc id: none] .
  op eventMap : -> EventMap .
```

PerF instruments the rules in the composed system model based on the user-specified events and their associated rule labels. All other rules remain unchanged. The following module illustrates how users specify the events of interest.

```
mod EVENT-2PC is
  inc EVENTS + 2PC .

  ops propose finish : Proposal -> Event .
```

[7] The module `FAULT-INJECTOR` imports the modules `SCHEDULER` and `CONTROLLER`; the latter further imports all relevant fault handlers, such as `MSG-LOSS` and `PARTITION`.

```
  eq eventMap = ['start, propose(P)] ;; ['decision, finish(P)] [nonexec] .
endm
```

The corresponding transformed rule then logs the `propose(P)` event at time `GT`, marking the point at which `P` starts. Note that the monitor and scheduler objects are added automatically by PerF, and the monitor is updated on the right-hand side of the rule to record this timestamped event.

```
  crl [start.p.m] : (start to O)
        < O : Coord | proposals: P ; PS, waiting: empty,
                      cohorts: OS, results: RS >
        < m : Monitor | events: TES > < sch : Scheduler | clock: GT >
     => < O : Coord | waiting: OS, results: insert(P,true,RS) >
              attachLabel('start.p, (propagate P from O to OS))
        < m : Monitor | events: TES ; (propose(P) @ GT) >  < sch : Scheduler | >
     if not exist(RS,P) .
```

Finally, the transformation adds the monitor object `< m : Monitor | events: empty >` to the initial state.

Quantitative Analysis. PerF supports two modes of quantitative analysis: pure simulation with Maude's built-in simulator and statistical model checking (SMC) with PVeStA [7]. In what follows, we focus on the latter.

To evaluate performance properties such as average latency, the user provides a QuaTEx expression of the desired metric:

```
AvgLatency() = { s.rval(0) } ;
eval E[ # AvgLatency() ] ;
```

where `rval(0)` invokes the Maude function `val(0,C)` that corresponds to the function `avglatency`. This function returns the average proposal latency by dividing the total accumulated latency by the number of completed proposals (lines 10–11). The relevant definitions are provided in the `ANALYSIS-2PC` module below. Note that performance properties in general and average latency in particular are defined over the events recorded in the monitor (line 8)

```
1  mod ANALYSIS-2PC is
2    ...
3
4    vars T T' : Float .  var P : Proposal .  vars E E' : Events .
5    vars TES TES2 TES3 : TimedEvents .
6
7    op val : Nat Config -> Float .
8    eq val(0, < m : Monitor | events: TES > C) = avglatency(TES) .
9
10   op avglatency : -> TimedEvents -> Float .
11   eq avglatency(TES) = totalLatency(TES) / numberOfProposal(TES) .
12
13   op totalLatency : TimedEvents -> Float .
14   eq totalLatency(TES ; (E propose(P) @ T) ; TES2 ;
15                   (E' finish(P) @ T') ; TES3)
16    = T' - T + totalLatency(TES ; TES2 ; TES3) .
17   eq totalLatency(TES) = 0.0 [owise] .
18
19   op numberOfProposal : TimedEvents -> Float .
20   eq numberOfProposal(TES ; (E propose(P) @ T) ; TES2 ;
21                       (E' finish(P) @ T') ; TES3)
22    = 1.0 + numberOfProposal(TES ; TES2 ; TES3) .
23   eq numberOfProposal(TES) = 0.0 [owise] .
24 endm
```

Once these steps are completed, the user only needs to specify the standard SMC parameters (e.g., the confidence level and error margin) to carry out the analysis, as we will see next.

A.3 Running the Tool

We provide a command-line interface that automates PerF's entire workflow.

Command Pattern. PerF is invoked via the following general command pattern:

```
sh run.sh [--pvesta serverlist analysis-model quatex confidence threshold]
    protocol init fault [event]
```

The arguments are interpreted as follows:

- `protocol` and `init` are the user-provided system model and its initial-state module, e.g., `2PC` and `INIT-2PC` in our running example.
- `fault` specifies the fault-injection interface module, e.g., `2PC-FAULT-CONFIG`.
- `event` (optional) specifies the events to be monitored during the model-transformation step, e.g., `EVENT-2PC`.

The optional `pvesta` flag enables analysis via PVeStA. In this mode:

- `serverlist` lists the servers used to run simulations,
- `analysis-model` denotes the analysis module (e.g., `ANALYSIS-2PC`),
- `quatex` denotes the QuaTEx formula file,
- `confidence` sets the confidence level (e.g., 0.01), and
- `threshold` sets the error margin (e.g., 0.01).

Preprocessing and Composition Only. The following invocation runs only the *model preprocessing* and *model composition* steps:

```
sh run.sh 2pc.maude init-2pc.maude 2pc-fault-config.maude
```

This command produces the composed model in which the user-provided system model and the fault injector are combined with the configured faults. No monitor is added, and no quantitative analysis is performed.

End-to-End Quantitative Analysis. The following command executes the entire workflow, including model preprocessing, model composition, model transformation, and finally quantitative analysis with PVeStA:

```
sh run.sh --pvesta -l serverlist -m analysis-2pc.maude -f latency.quatex -a
    0.01 -d 0.01 2pc.maude init-2pc.maude 2pc-fault-config.maude event-2pc.
    maude
```

This invocation produces the fully instrumented model, including the monitor, and automatically calls PVeStA to evaluate the QuaTEx formula written in `latency.quatex` using the provided analysis parameters.

Analysis Result. Figure 4 shows a screenshot of PerF running under the full command. The model composition and model transformation complete instantly. Moreover, 600 simulations finish in approximately 2.27 seconds and yield an expected average latency of 1.13 (time units), using 85.578 MB of memory.

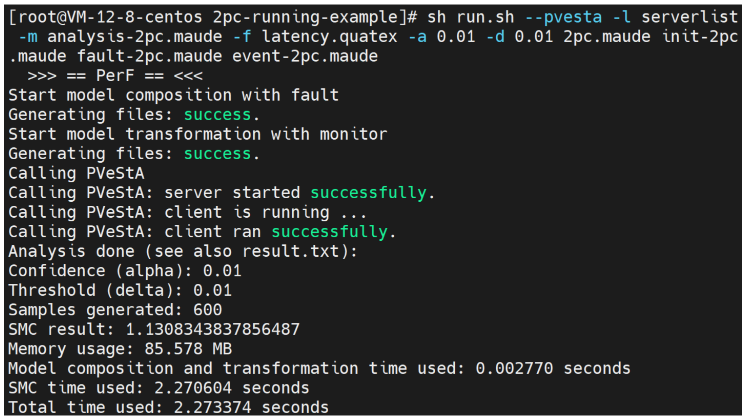
```
[root@VM-12-8-centos 2pc-running-example]# sh run.sh --pvesta -l serverlist
 -m analysis-2pc.maude -f latency.quatex -a 0.01 -d 0.01 2pc.maude init-2pc
.maude fault-2pc.maude event-2pc.maude
  >>> == PerF == <<<
Start model composition with fault
Generating files: success.
Start model transformation with monitor
Generating files: success.
Calling PVeStA
Calling PVeStA: server started successfully.
Calling PVeStA: client is running ...
Calling PVeStA: client ran successfully.
Analysis done (see also result.txt):
Confidence (alpha): 0.01
Threshold (delta): 0.01
Samples generated: 600
SMC result: 1.1308343837856487
Memory usage: 85.578 MB
Model composition and transformation time used: 0.002770 seconds
SMC time used: 2.270604 seconds
Total time used: 2.273374 seconds
```

Fig. 4: An example run of PerF with the SMC result.

B Proofs

This section presents proofs of the theorems that underpin PerF's statistical model checking analysis. Specifically, we prove that both the model composition (with fault injection) and the model transformation (with monitoring) satisfy the *absence of nondeterminism* (AND) property.

Definition 1 (AND [41]). *With probability 1, for any reachable state there is at most one rewrite rule applicable, with a unique matching substitution.*

B.1 Proof of Theorem 1

Theorem 1 (Section 4.4) establishes that the system model composed with the fault injector guarantees the AND property. The proof proceeds by induction on the number of rewrite steps from the initial state.

Assumptions. The user-provided model is untimed, nondeterministic with two kinds of rules: a rule that consumes a message, called a *message-triggered rule*, and a rule that fires without consuming a message, called an *object-triggered rule*. Following prior work [41], we make the following assumptions:

(1) If a message is delivered to an object, it processes the message by applying a *unique* message-triggered rule with a *unique* substitution.
(2) If an object-triggered rule is enabled, the object performs its local transition by applying a *unique* object-triggered rule with a *unique* substitution.
(3) The initial state contains exactly one object that is enabled either by a message-triggered rule or an object-triggered rule.

Assumptions (1) and (2) ensure local determinism: whenever an object is enabled, either by a message or by itself, there is exactly one applicable rule with a unique substitution. Note that this does not rule out nondeterministic choice between the local transitions currently possible for an object. Assumption (3) imposes the same form of local determinism on the initial state by requiring that exactly one object is enabled at the start.

Proof. We begin by categorizing the elements that may appear in a top-level system configuration:

- *qobjs*: quiescent objects that are not currently enabled by any *object-triggered rule*;
- *eobj*: an enabled object for which an *object-triggered rule* is applicable;
- *imsg*: an incoming message that has been released by the fault injector and is ready to be consumed by an object (e.g., a term like `MSG` that appears in the configuration outside the objects);
- *omsg*: outgoing message(s) produced by an object and already preprocessed, waiting to be intercepted by the scheduler (a term of the form `[MSG,L]`, which is immediately captured by the scheduler's insertion equation);
- *fmsg*: internal messages within the fault injector, such as `[GT,MSG,L]` and `auth([GT,MSG,L],B)`.

Based on these elements, we classify top-level system configurations into the following five mutually exclusive types:

(1) *qobjs eobj*: an object is enabled by an object-triggered rule (i.e., an object-triggered rule in the user-provided system model is applicable).
(2) *qobjs imsg*: there exists an incoming message ready for consumption.
(3) *qobjs*: all objects are quiescent, with no available messages and no enabled objects.
(4) *qobjs omsg*, *qobjs omsg eobj*, or *qobjs omsg fmsg*: one or more outgoing messages exist;
(5) *qobjs fmsg*: an internal message within the fault injector is present (e.g., for message inspection or authorization of fault behaviors).

Note that Types (4) and (5) do not represent persistent rewriting states; instead, they are transient configurations resolved by equations, not by the rewrite rules whose nondeterministic applicability is under consideration. The argument below makes this distinction explicit.

We are now ready to begin the proof. Let $n \geq 0$. For any configuration C reachable from the composed initial state by n rewrite steps, we need to show that, with probability 1, C has one of the forms (1)–(5) listed above, *and* at most one rewrite rule, with a unique matching substitution, is applicable to C. We prove by induction on the number of rewrite steps from the initial state.

Base Case ($n = 0$). The composed initial state is obtained by adding the fault-injection objects, i.e., the scheduler, the controller, and the handlers, to the original initial state `initState`. By Assumption (3), `initState` contains exactly one object, enabled either by an object-triggered rule or by a message-triggered rule. Therefore, the composed initial state is of Type (1) or Type (2).

Moreover, by Assumptions (1) and (2), exactly one rewrite rule with a unique matching substitution is enabled in this configuration.

Induction Step. Assume that the AND property holds for all configurations reachable in n rewrite steps. Let C be a configuration reachable in n steps, and consider a single rewrite step $C \longrightarrow C'$. By the induction hypothesis, C must have one of the five forms listed above. We now examine each form of C and establish that (i) in all cases exactly one rewrite rule is applicable to C, and (ii) the application of this rule yields a configuration C' that again belongs to one of the five forms.

Type (1): qobjs eobj. An object in the system model is enabled for exactly one object-triggered rewrite rule (with a unique matching substitution). By Assumption (2) and by the mechanism of eagerly applying object-triggered rules (see Appendix A), this rule is applied immediately, without releasing any incoming message to the actor. The application of the rule results in one of the following outcomes, all determined by the semantics of the system model:

- an outgoing message is produced (Type (4)), and the object may or may not remain enabled;
- no outgoing message is produced, but the object *eobj* remains enabled (Type (1)); or
- the object *eobj* becomes quiescent and no message is produced (Type (3)).

In all cases, exactly one rewrite rule is applied in this step, and the configuration C' again belongs to one of the five types.

Type (2): qobjs imsg. A message has been released into the configuration and is ready to be consumed by its destination object. By Assumption (1), the destination object has at most one enabled rewrite rule (with a unique matching substitution). Applying this rule yields one of the following outcomes:

- the message is consumed and one or more outgoing messages are generated (Type (4)), with the destination object either remaining enabled or becoming quiescent;
- the message is consumed and the destination object becomes enabled (Type (1)); or
- the message is consumed without producing any outgoing message, and the destination object remains quiescent after performing only internal updates (Type (3)).

Type (3): qobjs. No object is enabled, and no incoming message is pending for any object.

- If the scheduler holds no in-transit messages, then the system is quiescent and no rewrite rule is applicable from this point onward. The configuration is terminal (i.e., no more rewrite steps), and the AND property trivially holds.

– If the scheduler does hold in-transit scheduled messages, the scheduler proceeds by applying the "tick" equation, which advances the global clock and emits an internal message `[T,MSG,L]` within the fault injector, i.e., an *fmsg*. Since the scheduler's clock tick is realized by an equation, it does not count as a rule choice for AND. The configuration is transformed into Type (5), after which the controller is invoked. Hence, no nondeterministic rewrite rule is introduced at this stage.

Type (4): qobjs omsg, qobjs omsg eobj, or qobjs omsg fmsg. An outgoing message has been produced by a rewrite rule in the system model and appears as an *omsg*. In our framework, such outgoing messages are immediately processed by equations that attach the source rule label and insert the message into the scheduler's queue. This is deterministic.

Consequently, a configuration of Type (4) is transient and never serves as a rewriting target: it is immediately reduced by equations to a configuration of Type (3), or to Type (1) if an *eobj* is also present, or to Type (5) if an *fmsg* is generated. Therefore, Type (4) does not introduce any source of nondeterminism.

Type (5): qobjs fmsg. This category encompasses the internal inspection and authorization messages used within the fault injector. All transitions involving such messages are realized by equations associated with the controller and handlers. Therefore, configurations of Type (5) do not introduce nondeterministic rule choices. We now analyze the two subcases of *fmsg* in detail.

– *`[GT,MSG,L]` is delivered to the controller.* The controller inspects the message to determine whether any fault condition holds. Each predicate evaluation depends only on the handler objects' attributes and on the current random sample; hence, every predicate outcome is uniquely determined. The controller then issues at most one authorization (by selecting the behavior of the highest priority) or releases the message back to the configuration. Both the issuance of an authorization and the release are performed by equations. Therefore, the controller deterministically produces one subsequent fault-injector configuration, either a Type (5) configuration carrying an authorization or a Type (2) configuration after a release.
– *`auth([GT,MSG,L],B)` is delivered to a handler.* An authorization always targets a specific handler object that deterministically executes the associated behavior. Handler behaviors fall into one of the following four categories:
  - *Modify and recheck the message:* the handler produces a modified internal message `[GT,MSG',L]` and returns it to the controller (yielding a Type (5) configuration with a new *fmsg*);
  - *Reschedule the message:* the handler produces an outgoing message that is inserted into the scheduler's queue (Type (4));
  - *Drop the message:* the handler removes the message, resulting in a Type (3) configuration;
  - *Environment update:* the handler updates the network or node status and then returns the message to the controller (Type (5)).

Each authorization matches exactly one handler equation and is consumed by that equation. Hence, the handler's operation is uniquely determined, and no nondeterminism arises.

We have shown that each configuration C has at most one applicable rewrite rule and that its successor C' after one rewrite step again falls into one of the five forms. Hence, AND is preserved inductively, and the composed model with fault injection satisfies this property. □

B.2 AND Preserved by Monitoring

We now prove that the model transformation, which equips the composed model with a monitoring mechanism, preserves the AND property.

Theorem 2. *The composed model equipped with the monitor guarantees AND.*

Proof. Let $\mathcal{R}_C$ be the rewrite theory obtained after model composition. By Theorem 1, $\mathcal{R}_C$ satisfies the AND property. Let $\mathcal{R}_{C+M}$ denote the rewrite theory obtained after applying the model transformation. This transformation introduces two changes: (i) it adds a monitor object to the configuration, and (ii) it transforms only those rules that have user-specified events.

The transformation augments the initial configuration with a monitor object (see Appendix A), which is updated exclusively by the transformed rules corresponding to user-specified events. For each rule `[l]` associated with a user-specified event, the transformation produces a corresponding rule `[l.m]`. This rule preserves the original rule's semantics, and its only modification is an additional update to the monitor object's attributes, which records the events. Rules without associated events remain unchanged. Hence, no new rewrite rules are introduced, and the monitor does not participate in any rule-enabling conditions.

Let C be any reachable configuration of $\mathcal{R}_{C+M}$, and let $h(C)$ be its projection obtained by removing the monitor object. Then $h(C)$ is a reachable configuration of $\mathcal{R}_C$. Every rule in $\mathcal{R}_{C+M}$ is either an unchanged rule from $\mathcal{R}_C$, or a transformed rule whose applicability depends only on the non-monitor portion of the configuration. Hence, a rule of $\mathcal{R}_{C+M}$ is enabled in C iff its underlying rule in $\mathcal{R}_C$ is enabled in $h(C)$. Accordingly, the enabled-rule set of C coincides with that of $h(C)$. As $\mathcal{R}_C$ satisfies AND, at most one rule is enabled in $h(C)$, and it has a unique matching substitution. The same therefore holds in C: at most one (transformed or unchanged) rule is enabled, and it matches uniquely. The additional monitor updates do not influence rule applicability or matching elsewhere in the configuration, thereby introducing no nondeterminism.

We conclude that enabledness and unique matching are preserved by the model transformation, and therefore $\mathcal{R}_{C+M}$ also satisfies the AND property. □

C Other Case Studies

In this section, we discuss the remaining four case studies: (i) Two-Phase Commit (2PC) integrated with the Cooperative Termination Protocol (CTP) [14], (ii) the

Raft consensus protocol [52], (iii) the authoritative DNS server PowerDNS [53], and (iv) Cassandra's quorum-based consistency protocol [2].

2PC with CTP. 2PC is an atomic commitment protocol commonly used in distributed database systems. It consists of the PREPARE phase and the COMMIT phase, together ensuring the atomicity of transactions across multiple servers. To address its inherent blocking issues, CTP is often integrated with 2PC in practical implementations. For example, when a server S has performed PREPARE for transaction T but times out waiting for a COMMIT (due to, e.g., message loss), S can check T's status on its sibling servers.[8] In case a sibling S' has received COMMIT for T, then S can also commit T.

Figure 3(a) shows the average latency of 2PC with varying message loss rates. When combined with CTP, we observe a linear increase in the latency of 2PC as the number of undelivered COMMIT messages rises. The SMC estimation aligns closely with the CloudLab (Utah) evaluation.

Raft. The past decade has seen widespread adoption of the Raft consensus algorithm across distributed systems requiring fault-tolerant consensus. It is a leader-based algorithm in which the leader manages log replication. If the leader becomes disconnected or crashes, a new leader is elected through voting and then resolves any inconsistencies by forcing followers' logs to match its own.

We measure the time Raft takes to detect and replace a crashed leader in a cluster of five servers, as shown in Figure 3(b). The CloudLab (Utah) deployment results closely follow the overall cumulative trend predicted by PerF: both exhibit a similar rise in leader election latencies, although the deployment curve is slightly shifted to the right due to real-world network and system overheads.

PowerDNS. The Domain Name System (DNS) is a hierarchical distributed system that translates human-readable domain names into IP addresses, enabling efficient Internet navigation. Authoritative DNS servers, such as PowerDNS (widely used in Europe), are essential components of this infrastructure, playing a critical role in name resolution.

A recently identified attack reported in [39] is a slow DoS attack in which an attacker keeps a query "alive" inside a resolver, consuming resources needed for legitimate traffic. Using PerF, we reproduce this attack in PowerDNS by automatically injecting delays to the responses of attacker-controlled nameservers. The SMC estimation is plotted in Figure 3(c) alongside results from a DNS testbed [39]. Overall, the total query duration increases as additional delay is introduced by the malicious nameserver up to 0.6s, and the behavior of actual resolvers closely matches our model. In particular, introducing a delay of 0.6s can yield a cumulative end-to-end delay of roughly 6s for resolving a single query.

Cassandra's Quorum-based Consistency. Apache Cassandra is a distributed key–value store that employs a quorum-based mechanism to balance data consistency and availability. A quorum specifies the number of data replicas that

[8] A server's siblings are those servers associated with the same 2PC instance. For example, when a coordinator starts committing a transaction, it propagates PREPARE messages to three cohorts S_1, S_2, and S_3, then S_1 and S_3 are S_2's siblings.

must acknowledge a read or write operation for it to be considered successful. An alternative design [15] was shown to achieve performance comparable to the original mechanism, while returning more consistent data in certain scenarios. However, the analysis was performed under fault-free network conditions.

As shown in Figure 3(d), under network partitions the original design maintains more stable throughput, even when recovery is slower. The CloudLab (Clemson) deployment evaluations further confirm the behaviors observed in our model-based analysis. Since network partitions are unavoidable in geo-distributed settings, these results provide developers with deeper insight into the performance–consistency trade-offs between the two mechanisms.