A Domain Specific Language for Testing Consensus Implementations

Cezara Dragoi
INRIA, ENS

Constantin Enea
LIX, CNRS, Ecole Polytechnique

Srinidhi Nagendra
Universite Paris Cite, CNRS, IRIF, CMI

Mandayam Srivas
Chennai Mathematical Institute (CMI)

Abstract

Large-scale, fault-tolerant, distributed systems are the backbone for many critical software services. Since they must execute correctly in a possibly adversarial environment with arbitrary communication delays and failures, the underlying algorithms are intricate. In particular, achieving consistency and data retention relies on intricate consensus (state machine replication) protocols. Ensuring the reliability of implementations of such protocols remains a significant challenge because of the enormous number of exceptional conditions that may arise in production. We propose a methodology and a tool called Netrix for testing such implementations that aims to exploit programmer's knowledge to improve coverage, enables robust bug reproduction, and can be used in regression testing across different versions of an implementation. As evaluation, we apply our tool to a popular proof of stake blockchain protocol, Tendermint, which relies on a Byzantine consensus algorithm, a benign consensus algorithm, Raft, and BFT-Smart. We were able to identify deviations of the Tendermint implementation from the protocol specification and verify corrections on an updated implementation. Additionally, we were able to reproduce previously known bugs in Raft.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Distributed Systems, Domain Specific Languages, Testing, Blockchains

Category Extended Abstract for Lightning Talk

1 Introduction

Large-scale, fault-tolerant, distributed systems are the backbone of many critical software services. Since they must execute correctly and efficiently in the presence of concurrent and asynchronous message exchanges as well as failures, the underlying algorithms are intricate. In particular, *consensus protocols* are used to guarantee data retention and consistency which are important ingredients of storage systems such as Cassandra and Redis, or Blockchain systems based on proof-of-stake models. Improving the reliability of production implementations of such protocols remains a significant challenge, precisely because of the enormous number of exceptional conditions that may arise in production. In particular, such protocols have to tolerate faults that can be (1) benign such as replica crashes, network partitions, message drops, replays, or (2) Byzantine, where replicas can behave arbitrarily, coordinate or lie. Given a distributed protocol implementation, the number of possible executions is large even if the number of inputs (client requests) is small. This is an instance of the infamous state explosion problem. The number of interleavings grows exponentially in the number of replicas and the number of failures.

In this work we explore the space of tradeoffs between programmer effort in crafting good tests on one hand and the ability of automated tools to explore many behaviours on the other.

At one end, a skilled developer with a lot of knowledge of the system can write detailed tests that encode low-level executions of the system which control all the non-determinism by hand. However, this is very expensive in developer time. At the other end there are automated exploration tools which are protocol agnostic and search the execution space. These tools are particularly attractive for developers because they require minimal effort but are also unreliable because they usually get lost in a large execution space and explore many uninteresting executions. In this work we explore the space between these two approaches, and we propose a framework where the programmer gives lighthouses to automated search tools to guide their exploration.

Prior work on testing distributed protocol implementations focuses extensively on *exhaustive enumeration* or *random sampling* of executions for a fixed set of client requests, e.g., [3, 1, 23, 27, 19, 28, 24, 20, 16, 10]. Since even a small number of client requests leads to an enormous number of possible executions, the execution space is further bounded using protocol-agnostic parameters such as the number of steps in an execution, the time frequency of network partitions [1], the number of periods in which a replica is isolated from the rest [14], the number of events for which all possible orderings must be covered (bug depth) [23] etc. Shrinking the search space using such bounds decreases exploration time in the case of exhaustive enumeration and increases the effectiveness of random sampling.

In general, while random sampling techniques scale better to production implementations when compared to exhaustive enumeration, the techniques are typically applied only on abstract models. Among the prominent random sampling techniques, Jepsen [1] samples executions where a number of benign faults are introduced randomly with a certain frequency. Jepsen was shown to be quite effective in discovering bugs in production systems, but it provides no theoretical guarantees. On the other hand, depth-bounded random testing [6, 23] provides theoretical guarantees under an empirical hypothesis that many bugs in distributed systems are exposed by considering the sequencing of a small number of events, i.e., the bug depth, independent of the ordering of the rest of the events. For instance, the probability that the PCTCP algorithm [23] catches a bug of depth d is roughly, at least $1/n^d$ where n is a bound on the size of the executions (number of exchanged messages). While this probability reduces exponentially only with respect to the bug depth d, it can still become exceedingly small when executions are production scale, e.g., in the order of thousands of events, and d has a reasonable value, e.g., 10. As we show in our experimental evaluation (Section 5), such values of d are to be expected when searching for “interesting” executions that are not necessarily bugs but deviations from an abstract model for instance, or (bounded) liveness related bugs.

We present Netrix, a domain specific language with a networking infrastructure for programmer guided exploration of large-scale distributed systems. Netrix enables the programmer to write high-level scenario based tests as opposed to low-level packed manipulation with existing tools. The DSL allows the programmer to specify only the required event constraints, thereby allowing existing randomized techniques such as PCTCP to explore the residual non determinism. Thus, Netrix magnifies the coverage of behaviors specified by the programmer. In addition, our unit tests improve reproducibility, as opposed to ad-hoc methods for controlling the schedule, e.g., by introducing network partitions for a fixed period or sleep cycles. Netrix’s unit tests are aimed at regression testing different versions of an implementation and therefore, the unit tests help developers confidently develop and deploy new features without fear of breaking existing behavior.

Netrix runs a central server collecting messages and other events from replicas. It relies on a shallow instrumentation of the implementation under test that rewrites the communication primitives; Netrix provides several libraries to aid the instrumentation process. A unit test consists of a set of *filters* that apply to each event and message sent by a replica (which passes through the Netrix server), and a *state machine* that monitors the execution for some safety or bounded liveness property. A filter specifies some action to apply on a message, e.g., store and delay its reception, modify before forwarding to destination, no action at all, etc. The DSL simplifies the writing of filters using specific if-then constructs which are akin to entries in a Match-Action table from network devices. The "if" condition can relate to the type of the message or the number of messages already delivered to a destination, and the "then" part defines a set of messages to forward to a certain destination. The latter allows for message corruption, duplication, or arbitrary delays (defining that set of messages to be empty). Applying a filter may also lead to an update of Netrix's state, e.g., storing a message for later delivery.

A unit test imposes restrictions on the order in which *some* messages are delivered (if at all) while the order in which the rest of the messages are delivered is handled using the PCTCP algorithm. For those messages that are left by the unit test to be handled by PCTCP, the probability of hitting any specific ordering required to manifest a bug (or "interesting" behavior) is characterized by the probabilistic guarantees of PCTCP. If that specific ordering concerns few messages, then the probability of hitting it will be high. This allows a programmer to control reproducibility by adding more or less filters to the unit test. Moreover, introducing (benign or Byzantine) faults in a programmatic manner makes it possible to relate them to the logic of the protocol, e.g., dropping $f+1$ messages of some type when the protocol tolerates less than f process crashes, which simplifies debugging. This also allows a developer to decompose the testing effort into a number of disjoint scenarios that can be explored in parallel with no redundancy.

Netrix is implemented in the `go` programming language. Replicas communicate events and messages over RPC. As a consequence, developers can test implementations written in any programming language by changing the communication primitives of the implementation. We evaluated Netrix on several production implementations of consensus protocols. Consensus protocols work in the context of a number of servers called replicas that receive requests from clients, and they enable the replicas to agree on a common order in which to execute the client requests. During the execution of the protocol, replicas maintain a state and change it when they send/receive messages. The protocol defines rules to modify this state based on the type, order, and number of messages that arrive at a replica. Replicas can fail, and protocols ensure consensus in the presence of such failures.

We instrumented an open-source implementation of the Raft [21] protocol, the Tendermint [5] protocol and BFT-Smart [4, 25]. Using Netrix, we were able to (1) write unit tests to explore behaviors that are not commonly observed in production environments, including known bugs (2) demonstrate behaviors where the implementation deviates from the protocol specification, and (3) run tests on multiple versions of the implementation thereby checking corrections for the observed deviations. The tests for Tendermint allowed developers to gain confidence in the correctness of the implementation and write unit tests for planned changes. Overall, we make the following contributions,

- We introduce a DSL to write unit tests for testing distributed protocol implementations. These unit tests adopt a scenario based generic unit testing approach as opposed to current ad hoc mechanisms.
- We provide a testing tool Netrix to run the unit tests using PCTCP and describe the formal semantics of a distributed protocol execution under Netrix.
- We evaluate our tool and DSL on open source implementations of **Raft**, **Tendermint**

and **BFT-Smart**. We also report our findings for Tendermint where we identified bugs and helped the developer team to check fixes to the bugs.

1.0.0.1 Outline

We provide an overview of Netrix unit tests in Section 2. Section 3 provides the theoretical semantics of an execution under Netrix, while Section 4 explains the syntax and semantics of our DSL. We present our findings and common testing patterns while testing Tendermint, Raft and BFTSmart in Section 5. In Section 6, we present other related work before concluding in Section 7.

2 Our approach

A distributed system that Netrix tests consist of replicas that send/receive messages according to the protocol. A sequence of events characterizes an execution of the protocol in a replica. An event can correspond to sending, receiving a message or other internal steps such as starting or expiring a timer. Figure 1 illustrates the architecture of Netrix applied to a 4-node distributed system. All replicas run the same implementation augmented with a communication adapter that sends all events and messages to Netrix and receives messages from Netrix.

Netrix determines the order in which messages are received by replicas using two mechanisms. The default mechanism is the PCTCP algorithm for sampling executions of a distributed system (in principle, this can be replaced by any other random testing algorithm). PCTCP algorithm observes the set of messages sent during an execution, stores them into a set of (causally-ordered) chains, and delivers messages from these chains according to a random strategy (we give more details later in this section). Additionally, Netrix receives as input a unit test written by a developer. The unit test processes the events as a stream and determines the messages to be delivered at each step. A unit test can, for instance, block delivery of a message based on event A and deliver the message upon observing a later event B. These blocked messages are not seen by the PCTCP algorithm (they are not added to PCTCP's chains). Additionally, a unit test can also introduce fictitious messages which simulate Byzantine behaviors (e.g., where replicas sent messages that are not prescribed by the protocol). A unit test consists of a set of filters to control message delivery and an assertion defined by a state machine. We introduce a domain-specific language that is embedded in `go` to specify filters and assertions. Filters allow developers to introduce specific faults such as dropping messages, reordering, or replaying messages. If no filters are specified, the delivery of messages is controlled entirely by PCTCP.

To explain the structure of filters and assertions, we use the Practical Byzantine Fault Tolerance (PBFT)[7] protocol as a running example for the rest of the paper. PBFT solves consensus in the presence of Byzantine faults. For a distributed system of n replicas, PBFT can tolerate $f < \frac{n}{3}$ byzantine replicas. Replicas go through a sequence of rounds of communication called views. In each view v, one replica is chosen to be the leader, and the remaining replicas are identified as backups.

Figure 3, illustrates the "normal" execution of the protocol (no faults and instantaneous message delivery). In the *PrePrepare* phase, the leader p accepts a request α from the client and broadcasts a `PrePrepare`(α, v, i) message to all replicas. By doing so, the leader proposes to add the request α at index i to the log in view v. In the *Prepare* phase, if a replica accepts the `PrePrepare` message, it broadcasts a `Prepare`(α, v, i) message. Once a replica receives $2f + 1$ `Prepare` messages matching the `PrePrepare` message, it proceeds to

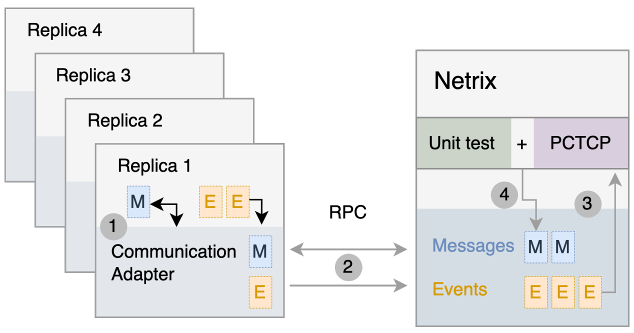

Figure 1 Netrix architecture
(1) The replicas submit events, send/receive messages to a communication adapter. (2) The adapter talks to Netrix via RPC. (3) The events and messages drive unit tests. (4) Unit tests decide which messages to deliver

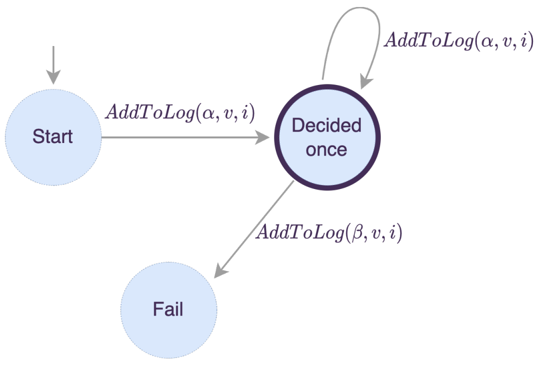

Figure 2 Event driven state machine for assertions.

the *Commit* phase and broadcasts a `Commit`(α, v, i) message. Finally, upon receiving $2f + 1$ `Commit` messages that match the `Prepare` message, each replica decides to add the request to the log. The client sends the request α to all replicas and upon receiving the request, the replicas start a timer. When the timer expires, if the replica has not added α to the log a view change protocol is initiated by broadcasting a `ViewChange`$(v + 1)$ message for the next view. When the leader of the view $v + 1$ receives `ViewChange` messages from $2f$ other replicas, it broadcasts `NewView`$(v + 1)$ messages to all replicas.

The correctness of PBFT relies on the property that no two replicas decide to add different requests to their log for a given index i in any given view. Developers can write state machines to specify this property. Consider a system of 4 replicas that receives two client requests α and β. We should not observe a replica p adding α at index i and replica q adding β at the same index. Figure 2 illustrates the state machine for this property. The transitions between states are labelled by a condition on an event. Starting from the initial state, the state machine takes a step every time we process an event received from a replica. When a transition condition is satisfied for the current event, we transition to the next state. In this case, `AddToLog`$(\alpha$`,v,i)` is a condition that is satisfied on the event indicating a replicas decision to add a request ”α” to the log at index ”`i`” in view ”`v`”. The predicate can be a function on the messages that Netrix observes, or the implementation explicitly emits the event (part of the instrumentation effort). The first time we observe such an event for a client request ”α” we transition to an accepting ”Decided Once” state. If we observe an

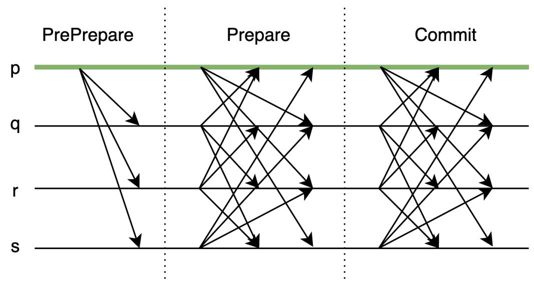

Figure 3 PBFT normal case execution with p as leader.

▶ **Example 2.1.** Drop `Prepare` messages of view v to one replica p

```
If(
  IsMessageType(Prepare)
    .And(MessageTo(p))
).Then(DropMessage())
```

event indicating a decision for a different client request "β" then we transition to a fail state.

Unit tests that contain no filters capture executions with arbitrary communication delays and faults (network partitions can be simulated with “infinite” delays), but no Byzantine faults. All messages that are delivered were sent by replicas following the protocol and the order in which messages are delivered is under the control of the underlying random testing algorithm, PCTCP in our case. Filters can be used to introduce specific delays or faults whose probability of being exposed by PCTCP is very low, or fictitious messages that simulate Byzantine faults. For instance, we can drop `Prepare` messages to one replica p and check if the property stated above still holds. Example 2.1 describes the corresponding filter. Filters form a general structure of, `If(condition).Then(actions)`. In this case, we define the conditions `IsMessageType` and `MessageTo` and compose them with logical operators. `IsMessageType` is true when the event is of type message-send, and the message is of the specified type. `DropMessage` is an action that does not deliver the and consequently, will not be seen by PCTCP. When the condition is true, the corresponding action is executed. Here, we drop `Prepare` messages to p.

Consequently, p will request for view change as it does not add the request r to its log. We expect the leader of the new view to not initiate a view change as only one out of the four replicas send a view change request. Consider an extension to the filter from Example 2.1, we drop `Prepare` messages to three replicas p, q and r. Example 2.2 describes the corresponding filter. The three replicas p, q and r initiate a view change as they did not add the request to

▶ **Example 2.2.** Drop `Prepare` messages of view v to replicas p, q and r

```
If(IsMessageType(Prepare).And(
  MessageTo(p).
  Or(MessageTo(q)).
  Or(MessageTo(r)))
).Then(DropMessage())
```

their respective logs. The leader of the new view will now have received $2f + 1$ (3 out of 4) view change requests to initiate a view change.

For Example 2.2, we strengthen the *safety* property specified by the state machine in Figure 2 by checking if additionally, the replicas initiated a view change. This is an instance of a (bounded) liveness property which requires that something “good” should eventually happen (it is bounded liveness because it is checked on executions that finish before a timeout expires – see Section 4). More precisely, from the initial state, upon observing 3 `ViewChange(1)` messages from distinct replicas, we transition to "ViewChangeExpected" state. Subsequently, when we observe a `NewView(1)` message from the leader of view 1, we transition to an accepting state.

A random exploration algorithm like PCTCP is very unlikely to sample executions where, like in Example 2.2, message are dropped or delivered towards the end. The low probability of delivering these messages at the end is due to the large number of messages in an execution (in the order of hundreds). Filters provide developers with a mechanism to introduce specific faults, delays and Byzantine behavior. Intuitively, this is analogous to mechanized proofs requiring additional input from the user, e.g., inductive invariants.

In the examples above, the filters introduce message drop faults. Developers can also introduce Byzantine faults using the filters. For example, we can change the contents of `Prepare` message of one replica p to *nil*. Example 2.3 defines the corresponding filter. Developers can define custom actions like `ChangePrepareToNil` that are specific to the protocol to introduce byzantine behavior.

▶ **Example 2.3.** Change `Prepare` messages of p to *nil*

```
If(IsMessageType(Prepare).
    And(MessageFrom(q))
).Then(ChangePrepareToNil())
```

In addition to introducing Byzantine behavior, the DSL allows reordering messages. Example 2.4 describes three filters. (1) The filter drops all `Prepare` messages of view 0. (2) `PrePrepare` message of view 0 to a process r are stored in a set. (3) Messages stored in the set are delivered when a `PrePrepare` message of view 1 is observed. Under this scenario, we test for the generic safety property that no two processes decide different values (in the same view). Filter 1 ensures that processes move to the next view and filters 2 and 3 ensure that process r receives two different `PrePrepare` messages.

▶ **Example 2.4.** Reorder the `PrePrepare` message to process r to the next view.

```
1. If(IsMessageType(Prepare).And(MessageView(0))).
      Then(DropMessage())
2. If(IsMessageType(PrePrepare).And(MessageTo(r)).And(MessageView(0))).
      Then(StoreInSet("reordered"))
3. If(IsMessageType(PrePrepare).And(MessageView(1))).
      Then(DeliverAllFromSet("reordered"))
```

As mentioned above, PCTCP is very unlikely to explore executions that correspond to Filter 1 (dropping messages), but it is effective in exploring more restricted re-orderings like the one represented by Filters 2 and 3. For a brief recall, for some integer d which is given as input, PCTCP generates uniformly at random an execution from a family of *d-hitting* executions F of size $|F| = w^2 n^{d-1}$ where n is an upper bound on the number of steps in an execution and w is the width of the partial order of messages (causally ordered) observed by the algorithm. This poset of messages is maintained as a set of (at most w^2) chains which are sequences of messages whose order is consistent with the causality order. The chains are associated with randomly chosen priorities and they are scheduled according to these priorities. The algorithm ensures to hit each ordered sequence of d messages with probability of at least $\frac{1}{w^2 n^{d-1}}$.

Returning to Example 2.4, the set of executions where `Prepare` messages of view 0 are delivered at the end, as a proxy for dropping, is contained in a d-hitting family for a large d in practice and hence probability of PCTCP hitting is very small. This is observed in practice as well: running PCTCP without these filters for 200 times produces no execution where `Prepare` messages of view 0 are delivered at the end. On the other hand, Filters 2 and 3 define a message re-ordering with a smaller scope: `PrePrepare` messages of view 0 to a process r need to be delivered after *any* message from view 1 and PCTCP is able to explore this reordering. This provides a tradeoff to the developers writing unit tests where more expressive tests (in terms of number of filters) provide better probabilistic guarantees. On the other hand, broader exploration of the execution space can be achieved with smaller tests. Furthermore, in Section 5, we associate filters with a distance score based on the number of messages between the actual position in a synchronous execution and the expected position after re-ordering. We show experimentally that filters that exceed a certain distance threshold are necessary to constrain the execution space to observe the desired property. The distance measure can help developers in making the right tradeoff decisions.

We present the syntax and the semantics of the filters and of the state machine in Section 4 (along with the instrumentation effort). The next section formalizes the semantics of a protocol with Netrix.

3 System model

We formalize the semantics of a Netrix unit test for a protocol as a product between a transition system modeling the executions of the protocol with a set $\mathcal{R}$ of replicas ($|\mathcal{R}| = n$) and a transition system modeling a monitor that controls the delivery of messages to the replicas and asserts some property. The latter is an abstract representation of Netrix unit tests whose syntax is presented in Section 4. We characterize the capability of the monitor to restrict the protocol behavior by showing that it can be programmed to reproduce precisely any set of executions that differ only in the order of concurrent computation steps (not related by the standard happens-before relation).

The set $\mathcal{R}$ of replicas exchange messages during the execution of the protocol. A message m is a tuple $(\mathcal{R} \times \mathcal{R} \times \mathcal{V}_m \times \mathcal{T}_m)$, where $\mathcal{V}_m$ denotes the set of possible message values and $\mathcal{T}_m$ denotes the set of possible message types. Any set $M \subseteq \mathcal{M}$ of messages can be partitioned based on the replica the message is intended to $M[r] = M \cap (\mathcal{R} \times \{r\} \times \mathcal{V}_m \times \mathcal{T}_m)$ where $r \in \mathcal{R}$.

An execution of a protocol at a replica can be characterized by a sequence of events. Events can correspond to sending or receiving a message, or an internal computational step. For example, in PBFT an internal step can be committing a value or changing the view. An event e is a tuple $(\mathcal{R} \times \mathcal{T}_e \times \mathcal{V}_e)$, where $\mathcal{T}_e = \{send, receive, internal\}$ is the set of event types and $\mathcal{V}_e \supseteq \mathcal{M}$ is the set of possible event values. We say $\mathcal{E}$ is the set of all events.

3.1 Protocol transition system

In a protocol, we represent each replica as a transition system $(\Sigma_p, s_p^0, \delta_p, F_p)$ where

- Σ_p is the set of replica states as defined by the protocol with the initial state $s_p^0 \in \Sigma_p$
- $\delta_p : \Sigma_p \times (\mathcal{M} \cup \{\bot\}) \rightharpoonup \Sigma_p \times \mathcal{E}$ is the (partial) transition function of the replica. Each transition emits an event. Internal steps are represented with transitions $\delta_p(s, \bot)$ while receiving a message m is represented with a transition $\delta_p(s, m)$. We assume that internal steps and message receive steps cannot be enabled in the same state, i.e. from any state $s \in \Sigma_p$ if $\delta_p(s, \bot)$ is defined, then $\delta_p(s, m)$ for any $m \in \mathcal{M}$ is not defined.
- $F_p \subseteq \Sigma_p$. For any state $s \in F_p$, δ_p is not defined from s. Final states allows us to restrict the length of an execution in each replica.

For example, in PBFT, the set of states is a valuation of the local variables (index, view, etc) with the initial state as both index and view being 0. A replica in PBFT can transition to a leader state and emit the corresponding event.

A protocol $\mathcal{P}$ is a product of $|\mathcal{R}| = n$ replica transitions systems. The configuration of the protocol is denoted by $C = (E, \mathit{pool}, \mathit{states}, \mathit{messages})$ where,

- $E = (e_0, e_1, \cdots)$ is a sequence of events $e_i \in \mathcal{E}$. This serves as a queue of events where replicas push new events to be consumed by the monitor (as it will be clear when defining the monitor transition system below).
- $\mathit{pool} \subseteq \mathcal{M}$ is the set of messages in transit between different replicas
- states maps each replica r to a state in Σ_p
- $\mathit{messages}$ maps each replica r to a sequence of messages $(m_0, m_1, \cdots)$ with $m_i \in \mathcal{M}[r]$. This sequence is used as an inbound queue that replicas use to process incoming messages

$$\frac{\delta_p(\mathit{states}[r], \bot) = (s', e) \qquad e = (r, \mathit{internal}, v)}{(E, \mathit{pool}, \mathit{states}, \mathit{messages}) \xrightarrow{e} (E, \mathit{pool}, \mathit{states}[r \to s'], \mathit{messages})} \ \textsc{Internal}$$

$$\frac{\delta_p(\mathit{states}[r], \bot) = (s', e) \qquad e = (r, \mathit{send}, m)}{(E, \mathit{pool}, \mathit{states}, \mathit{messages}) \xrightarrow{e} (E.e, \mathit{pool} \cup \{m\}, \mathit{states}[r \to s'], \mathit{messages})} \ \textsc{Send}$$

$$\frac{\mathit{messages}[r] = \sigma \qquad m \in \mathit{pool}[r]}{(E, \mathit{pool}, \mathit{states}, \mathit{messages}) \xrightarrow{\mathit{network}} (E, \mathit{pool} \setminus \{m\}, \mathit{states}, \mathit{messages}[r \to \sigma.m])} \ \textsc{Network}$$

$$\frac{\mathit{messages}[r] = m.\sigma \qquad \delta_p(\mathit{states}[r], m) = (s', e) \qquad e = (r, \mathit{receive}, m)}{(E, \mathit{pool}, \mathit{states}, \mathit{messages}) \xrightarrow{e} (E.e, \mathit{pool}, \mathit{states}[r \to s'], \mathit{messages}[r \to \sigma])} \ \textsc{Receive}$$

$$\frac{M \subseteq \mathcal{M}}{(E, \mathit{pool}, \mathit{states}, \mathit{messages}) \xrightarrow{\mathit{adversary}} (E, M, \mathit{states}, \mathit{messages})} \ \textsc{Adversary}$$

Figure 4 Transition rules of a protocol. For a function $f : A \to B$, we use $f[a \to b]$ to denote a function $f' : A \to B$ where $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \neq a$.

Figure 4 refers to the transition rules between two configurations of the protocol. The rule Internal allows a replica to transition its state and emit *internal* events. In the Send rule, a replica emits a *send* event for a message m and the message is added to *pool*. The Network rule adds messages from the pool to a replica's inbound message queue $\mathit{messages}[r]$. Rule

Receive allows replicas to consume a message from their inbound queues by emitting a *receive* event. Additionally, the Adversary rule models adversarial (Byzantine) behavior where the message pool is transformed arbitrarily.

We define an execution ρ as a sequence of transitions between configurations: $\rho = C_0 \xrightarrow{l_0} C_1 \xrightarrow{l_1} \cdots \xrightarrow{l_{k-1}} C_k$. The initial configuration is $C_0 = ((), \phi, states_0, messages_0)$ with $\forall r \in \mathcal{R}$, $states_0[r] = s_p^0$ and $messages_0[r] = ()$. An execution ρ is *complete* if all replicas have reached final states. ($\forall r, states_k[r] \in F_p$).

The *history* of an execution ρ is the tuple $H_\rho = (E_\rho, <_\rho)$ where E_ρ is the set of events in ρ ordered by the standard (partial) happens-before order $<_\rho$. Formally, $E_\rho = \{e \mid \exists l_i \in \rho, l_i = e\}$. We will use $e \in E_\rho$ and $e \in H_\rho$ interchangeably to denote an event exists in a history. For two events $e_1, e_2 \in E_\rho$, we say $e_1 <_\rho e_2$ if (1) e_1, e_2 are emitted by the same replica, and e_1 occurred before e_2 in ρ, (2) e_1 is a send event and e_2 is the matching receive event, i.e., $e_1 = (m.from, send, m)$ and $e_2 = (m.to, receive, m)$, and (2) (transitive closure) there exists e_3 such that $e_1 <_\rho e_3$ and $e_3 <_\rho e_2$.

Also, we define $M_\rho = \{m \mid (m.to, receive, m) \in E_\rho\}$ as the set of messages delivered to the replicas in the execution.

3.2 Monitor transition system

Netrix includes a central monitor that receives all communication from the replicas and is able to control the delay of delivered messages or send new messages by itself as a way to model Byzantine faults. The monitor is also used to assert some property. The central monitor is driven by the events emitted by the replicas. At each step, the monitor can decide to block a certain message from being delivered or to deliver some set of messages based on the current event and context (monitor state). We define the monitor transition system as a tuple $\mu = (\Sigma_\mu, s_\mu^0, \delta_\mu, F_\mu)$ where

- Σ_μ is the set of possible monitor states with s_μ^0 as the initial state
- $\delta_\mu : \Sigma_\mu \times \mathcal{E} \rightharpoonup \Sigma_\mu \times 2^{\mathcal{M}} \times \mathbb{B}$ is the transition function which accepts the current state and event, and transitions to a new state along with a set of messages to be delivered and a Boolean value in $\mathbb{B} = \{\bot, \top\}$ which in the case of a send event signals whether the message is blocked (if $\top$) or available for delivery (if $\bot$). If the message is blocked in order to be delivered later, then it can be stored in the state of the monitor until delivery. δ_μ encapsulates the semantics of executing the filters and state machine for a given unit test.
- $F^M \subseteq \Sigma_\mu$ is the set of accepting states (used to signal some property being satisfied)

A configuration of the monitor is defined by $C = (E, blocked, messages, s)$ where

- E is a sequence of events as described in the protocol transition system. Here it serves the purpose of an event queue for the monitor to consume.
- *blocked* is the set of blocked messages
- *messages* maps each replica to a sequence of messages as in the protocol transition system
- $s \in \Sigma_\mu$ is a monitor state

Figure 5 describes the Monitor transition rule. The rule invokes the transition function δ_μ with the head of the event queue E and the current monitor state as input and returns the new state, a set of messages to deliver, and a possibly updated set of blocked messages. The delivered messages are added to the respective replica's inbound message queues and the monitor state is updated.

$$\frac{E = e.E' \qquad \delta_\mu(s,e) = (s', M, b) \qquad \forall r.\ \mathit{messages}'(r) = \mathit{messages}(r).M[r] \\ \mathit{blocked}' = \mathit{blocked} \cup (b?\{m\} : \emptyset),\ \text{if } e = (\_, \mathit{send}, m), \text{ and } \mathit{blocked}' = \mathit{blocked}, \text{ otherwise}}{(E, \mathit{blocked}, \mathit{messages}, s) \xrightarrow{\mathit{monitor}} (E', \mathit{blocked}', \mathit{messages}', s')}$$

Figure 5 Monitor transition rule ($\mathit{cond}?v_1 : v_2$ is interpreted to v_1 if cond is true and v_2, otherwise).

$$\frac{(E, \mathit{pool}, \mathit{states}, \mathit{messages}) \xrightarrow{e \in \mathcal{E}} (E', \mathit{pool}', \mathit{states}', \mathit{messages}')}{(E, \mathit{pool}, \mathit{states}, \mathit{messages}, \mathit{blocked}, s) \longrightarrow (E', \mathit{pool}', \mathit{states}', \mathit{messages}', \mathit{blocked}, s)}\ \textsc{Protocol}$$

$$\frac{(E, \mathit{blocked}, \mathit{messages}, s) \xrightarrow{\mathit{monitor}} (E', \mathit{blocked}', \mathit{messages}', s')}{(E, \mathit{pool}, \mathit{states}, \mathit{messages}, \mathit{blocked}, s) \longrightarrow (E', \mathit{pool}, \mathit{states}, \mathit{messages}', \mathit{blocked}', s')}\ \textsc{Monitor}$$

$$\frac{\mathit{messages}[r] = \sigma \qquad m \in \mathit{pool}[r] \qquad m \notin \mathit{blocked}}{\begin{array}{c}(E, \mathit{pool}, \mathit{states}, \mathit{messages}, \mathit{blocked}, s) \\ \longrightarrow (E, \mathit{pool} \setminus \{m\}, \mathit{states}, \mathit{messages}[r \to \sigma.m], \mathit{blocked}, s)\end{array}}\ \textsc{Network-NB}$$

Figure 6 Transition rules of product system

3.3 Product transition system

The asynchronous product of the two transition systems, that of the protocol $\mathcal{P}$ and the monitor μ defines the set of executions admitted by a unit test (that will be explored by Netrix using PCTCP). The protocol transition system takes steps which publish events that are consumed by steps of the monitor. In turn, the monitor controls the messages that are consumed by the replicas in the protocol transition system. A configuration of the product transition system is $C = (E, \mathit{pool}, \mathit{states}, \mathit{messages}, \mathit{blocked}, s)$:

- E, *pool*, *states* and *messages* are defined as in the protocol transition system.
- *blocked* is the set of blocked messages controlled by the monitor and s is the monitor state.

Figure 6 defines the transition rules for the product transition system. Steps can be either Internal, Send, Receive from the protocol transition system (transitions labeled by events $e \in \mathcal{E}$), a step in the monitor transition system Monitor, or a variation of the Network rule (Network-NB) which adds a *non-blocked* message to a replica's inbound message queue. Note that a monitor step can deliver a set of messages (add them to replica inbound queues) which have been either sent by replicas in the past (these messages were blocked by the monitor and stored in its state) or fictitious messages (corresponding to Byzantine behavior). Therefore, a monitor step simulates a (possibly-empty) sequence of Network and Adversary steps from the protocol transition system.

A run of the product system $\rho = C_0 \xrightarrow{l_0} C_1 \xrightarrow{l_1} \cdots \xrightarrow{l_{k-1}} C_k$ is a sequence of transitions as above. A run is accepting if the monitor's state in the last configuration is a final state, i.e., $C_k^M.s \in F^M$. The set of accepting states of the monitor models the success or failure of a unit test. Netrix uses PCTCP in order to explore the non-determinism introduced by Network-NB transitions.

3.4 On the Expressivity of the Monitor

We give a characterization of the monitor's capability to restrict the protocol behavior. We show that as an extreme case, the monitor can restrict the protocol to produce a *single* history for a complete execution, i.e., all the complete executions in the product transition system have the same history. We characterize the capabilities of the monitor in terms of histories because the monitor cannot control the order between concurrent events, which are incomparable w.r.t. happens-before. The order in which such events are pushed to the event queue by the replicas is not under the control of the monitor. In theory, the number of executions that have the same history can still be exponential. However, in practice, for consensus protocols in particular, large numbers of them will be indistinguishable in the sense that every replica will go through exactly the same states (modulo stuttering). For instance, the order between concurrent receive events on different replicas does not affect any replica local state. Since assertions are expressed on local replica states, restrictions in terms of produced histories are effective.

We state our result as a relation between the histories produced in a product transition system and the history of a given complete protocol execution ρ. Histories of possibly incomplete executions of the product transition system are not necessarily equal to the history of ρ but only a *prefix*. The prefix relation $\preceq$ between two histories $H_1 = (E_1, <_1)$ and $H_2 = (E_2, <_2)$ is defined as usual, i.e., $H_1 \preceq H_2$ if (1) Downward closure: $E_1 \subseteq E_2$ and for every event $e \in E_1$ and $e' \in E_2$, $e' <_2 e \Rightarrow e' \in E_1 \wedge e' <_1 e$, and (2) Preserving happens before: For two events $e, e' \in E_1$, $e <_1 e' \Leftrightarrow e <_2 e'$.

▶ **Theorem 1.** *For any complete run ρ in the protocol $\mathcal{P}$, there exists a monitor μ such that, for all executions ρ' in the product transition system of $\mathcal{P}$ and μ, $H_{\rho'} \preceq H_\rho$*

In this proof we construct a monitor to reproduce *exactly one* history. In practice however, a developer writes unit tests to reproduce one of a set of histories. For example, in a unit test of PBFT, a developer is interested in executions where `Commit` messages are delivered to a replica before `Prepare` messages, without restricting the order between messages of the same type. Our DSL allows a developer to program such constraints into the monitor's transition function δ_μ and hence observe the exact expected behavior. We defer a more elaborate discussion to Section 4. In the next section we describe the effort needed to instrument an existing protocol implementation with the monitor i.e. Netrix.

4 Netrix unit tests

To use Netrix for testing, the implementation needs to be modified such that Netrix receives all messages and events from the participating replicas. Netrix exposes an API to receives messages, events, and other necessary information from the replicas via RPC. We provide client libraries, currently in `go` and `java`, that developers can use to change their implementation's communication primitives. Figure 7 illustrates the changes needed. Replicas first establish a connection to Netrix and communicate the relevant information. Earlier, replicas would communicate messages directly to each other. While testing with Netrix, for every message that a replica intends to send, the replica should first communicate an event of type message-send tagged with the message identifier and then submit the message to Netrix. In addition to the messages, Netrix can issue directives such as `start`, `stop` and `restart` to each replica. The implementation has to be changed to handle the directives. Netrix receives messages as a sequence of bytes and the developer writing the tests should provide a serialization and deserialization adapter.

Snippet 1 Initialization code

```
server, err := NewStrategyDriver(
  Config{
    APIServerAddr: ":7074",
    NumReplicas: 4,
  },
  PBFTMessageParser(),
  NewPCTStrategyWithTestCase(
    PCTStrategyConfig{
      MaxEvents: 1000,
      Depth: 10,
    },
    TestCaseOne(),
  ),
  StrategyConfig{
    Iterations: 1000,
    IterationTimeout: "30s",
  },
)
server.Start()
<-onInterrupt
server.Stop()
```

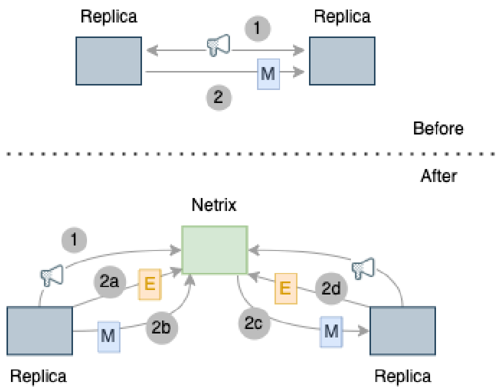

Figure 7 Changes to the implementation
(1) Discovery process, replicas establish connection to Netrix (**After**) as opposed to discovering each other (**Before**). (2) **Before**: replicas exchange messages. **After**: (a) Replicas send a message send event to Netrix (b) followed by the message. (c) Replica receives the message and (d) submits a receive event

The central testing server of Netrix is available as a ”go” library. Implementing the `strategy` interface provided by the library allows a developer to define the logic for exploration. The interface allows the framework to encapsulate and hide all the wireframe needed to connect to the replicas, receive events and messages. The interface defines a function `Step` which is invoked for every event observed by Netrix. To instantiate the testing server, developers should invoke the function `NewStrategyDriver` with server configuration, a strategy and strategy configuration (e.g. number of iterations to run). For all our experiments, we use the strategy `PCTStrategyWithTestCase` which accepts a unit test specified as a `TestCase` object. Additionally, `NewStrategyDriver` requires as argument, an instance of `MessageParser` which is used to serialize and deserialize the messages received. Snippet 1 contains an example instantiation of `NewStrategyDriver`. Developers should define a message parser that is specific to the protocol and implementation. A developer should invoke the `Start` function to run the tests.

Internally, we wait for replicas to connect before running the test iterations. Netrix stores the events received from the replicas in a queue `EventQueue` and the messages received in a pool `MessagePool`. We create the `EventQueue` and `MessagePool` objects when the developer instantiates `StrategyDriver`. Between successive iterations of the strategy, Netrix clears `EventQueue` and `MessagePool`, and sends a `restart` directive to all replicas.

A unit test is represented by a `TestCase` object. Developers should invoke `NewTestCase` to create unit tests. `NewTestCase` accepts the test case name, a timeout duration, the state machine and the filters. Developer should create the state machine by invoking `NewStateMachine` and the filters by invoking `NewFilterSet`. Snippet 2 defines an example unit test `TestCaseOne` for PBFT from Example 2.2 where we drop `Prepare` messages to replicas p, q and r. Developers can add more filters by calling `filter.AddFilter` function. Internally, Netrix invokes `Strategy.Step` function with an `Event` and `Context`. The step

Snippet 2 TestOne

```
function TestOne() *TestCase {
  ...
  filters.AddFilter(
    If( IsMessageType("Prepare").
        And( MessageTo("p").
             Or(MessageTo("q")).
             Or(MessageTo("r")),
      )).Then(DropMessage()) )
  return  NewTestCase("TestOne",sm,filters)
}
```

Snippet 3 State Machine

```
sm := NewStateMachine()
initial := sm.Builder()
decidedOnce := initial.On(
  AddToLog(α,v,i),
  "DecidedOnce")
decidedOnce.MarkSuccess()
decidedOnce.On(
  AddToLog(β,v, i),
  FailState
)
```

function serves the purpose of δ_μ defined in Section 3 and `Context` is the equivalent of a monitor state $s \in \Sigma_\mu$. We model the monitor state as a key-value store. `Context` contains a reference to this store along with a reference to `MessagePool`.

`SetupFunc` allows the developer to perform any initialization that is specific to that unit test. For example, in Snippet 2, the setup function can be used to randomly choose the replicas p, q and r. The syntax for filters is of the general form: `If(condition).Then(actions)`. We define **Conditions** as functions that accept an event and a context object and return a boolean. **Actions** are defined as functions that accept an event and a context object and return a set of messages.

Conditions and actions can only retrieve messages from the `MessagePool`. Conditions and actions should use the key-value store in `Context` to record auxiliary state information. For example, to reorder messages, filters can store the message temporarily in the auxiliary state. Apart from storing message, the auxiliary state can be used to keep count of the number of messages/events of a specific type. We define `MessageSet` and `Count` primitives to represent data stored in the auxiliary state. The primitives are necessary to introduce message replay faults or constrain the number of messages to be delivered. Figure 8 illustrates an invocation of a filter within the `PCTStrategyWithTestCase.Step` function. For each event from the `EventQueue`, we execute the filters similar to a switch case. We check the conditions in the sequential order of the filters and invoke the corresponding actions when a filter condition returns true. When none of the filter conditions match, the event is passed to PCTCP. Additionally, we deliver messages scheduled by PCTCP at each step.

```
condition :=  IsEventType(t) | IsMessageType(t) |
IsMessageSend | IsMessageReceive |
```

```
action := DeliverMessage |
DropMessage |
```

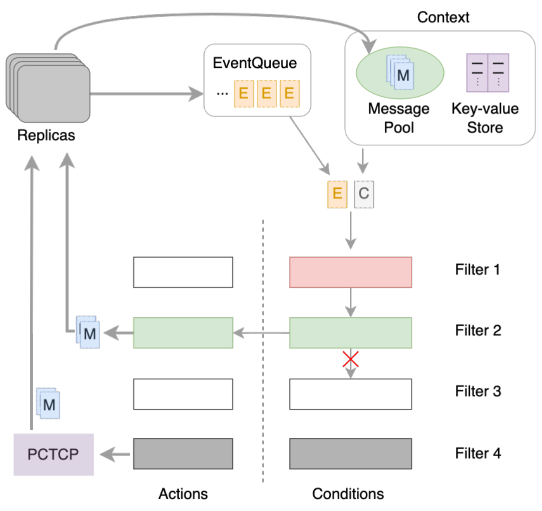

Figure 8 Invoking filters with an event and context: When a filter condition is true for the event E and context C, the messages defined by the action is delivered

```
IsMessageFrom(r) | IsMessageTo(r) |                       Count(c).Incr |
Count(c).Lt(v) | Count(c).Gt(v) |                         MessageSet(s).Store |
Count(c).Leq(v) | Count(c).Gte(v) |                       MessageSet(s).DeliverAll
MessageSet(s).Contains |
condition.And(condition) | condition.Or(condition) | condition.Not
```

Our DSL allows the developer to compose conditions/actions and reuse them in the filters of different unit tests. Table 1 describes the semantics of each condition. The conditions do not modify the auxiliary state. However, some of the condition's return values depends on the contents of the auxiliary state. Table 2 describes the semantics of each action. Some of the actions only modify the auxiliary state and do not return any messages. Developers should write custom actions to modify the contents of the messages which are delivered. We do not provide default actions as the message contents are protocol specific. As we model conditions and actions as functions, developers can define custom conditions and actions specific to the implementation under test. Conditions and actions can be one of those specified earlier. Additionally, we provide more filters such as `IsolateNode`, `RecordMessage` and ability to partition replicas as syntactic sugar to help with more concise tests.

Apart from the key-value store in `Context` object, the state machine also contributes to the monitor state (as defined in Section 3). Developers can construct the state machine using a builder design pattern. Snippet 3 constructs the state machine from Figure 2. To recap, the property we are testing for is - in a given view v of PBFT, we should not observe decisions for two different client requests r, r'. The state machine by default consists of an initial state, an accepting success state, and a fail state. The transition labels are conditions. `On` accepts a condition and the next state label. `On` constructs a transition from the current state to the next state and returns the next state object. Builder returns the initial state. MarkSuccess allows one to mark the current state as an accepting state. In Snippet 3, `AddToLog` is true when the event is a decision to add α to the log in view v. The state machine is used to assert global properties on the execution by requiring that a certain sequence of events occur in the replicas. Similar to the filters, the state machine takes a step for every event. We transition to the next state for a given event when the transition label (condition) returns true. At the start of the unit test, the state machine is in the initial state. When the iteration

Table 1 Semantics of conditions for event *e* and context *ctx*

Condition	Return value
IsEventType(t)	true if the e.type = t
IsMessageType(t)	true if e.type=messagesend(m)/messagereceive(m) and m.type=t
IsMessageSend	true if e.type=messagesend
IsMessageReceive	true if e.type=messagereceive
IsMessageFrom(r)	true if e is message send/receive and m.from=r
IsMessageTo(r)	true if e is message send/receive and m.to=r
Count(c).Lt(v)	true if ctx[c] < v
Count(c).Gt(v)	true if ctx[c] > v
Count(c).Leq(v)	true if ctx[c] <= v
Count(c).Gte(v)	true if ctx[c] >= v
MessageSet(s).Contains	true if e is messagesend(m)/messagereceive(m) and m $\in$ ctx[s]
c1.And(c2)	true if c1(e,ctx) $\wedge$ c2(e,ctx)
c1.Or(c2)	true if c1(e,ctx) $\vee$ c2(e,ctx)
c.Not	true if !c(e,ctx)

Table 2 Semantics of actions for an event *e* and context *ctx*

Action	Return value	Context changes
DeliverMessage	if e.type=send(m) returns m	-
MessageSet(s).Store	empty set	ctx[s] = ctx[s] $\cup$ m where e.type=send(m)/receive(m)
DropMessage	empty set	-
MessageSet(s).DeliverAll	returns ctx[s]	ctx[s] is set to empty
Count(c).Incr	empty set	ctx[c]++

timeout expires, we say that the iteration is successful if and only if the state machine is in an accepting state.

5 Case studies

Using our DSL, we created unit tests to test open source implementations of **Tendermint**[1] [5] (version 34.3), **Raft**[2] [21] (version v3.5.2) and **BFTSmart**[3]. The Tendermint protocol is the backbone of the cosmos network[4] of blockchains and is inspired by PBFT. Tendermint solves the consensus problem for a distributed network of replicas in a byzantine fault model. Similarly, BFTSmart[4, 25] is a byzantine consensus algorithm that is used to build key-value stores and distributed file systems. Raft is a popular benign consensus protocol that tolerates crash failures. It is used in many cloud services such as `etcd` and distributed graph databases such as `dgraph`

With our evaluation of the three protocols, we aim to answer the following questions,

Q1 Can our DSL and Netrix be used to write unit tests that are short and effective?

[1] https://github.com/tendermint/tendermint
[2] https://github.com/etcd-io/etcd/tree/main/raft
[3] http://github.com/bft-smart/library
[4] https://cosmos.network

Q2 Can we write tests to replicate known bugs?

We inject benign faults in Raft, Tendermint and BFTSmart to induce deviations from a normal execution and in Tendermint and BFTSmart, we inject byzantine faults by changing the contents of the messages that are observed by Netrix. For all the protocols, in our evaluation we record: (1) Outcomes of unit tests (number of iterations where the assertion is true), (2) Developer effort in terms of LOC, and (3) Complexity of tests (number of filters, state machine states and filter distance)

Additionally, we introduce a measure of filter distance (Section 5.4) to aid the developer in writing unit tests. With this measure, the developer can chose to forego writing some of the filters and rely on PCTCP's exploration.

5.0.0.1 Summary of results

For **Tendermint**, we describe 15 test scenarios in total. We found that 4 unit tests fail, demonstrating behavior where the implementation deviates from the protocol specification. Among them, one of the unit tests demonstrates a liveness bug. The bug increases the time to achieve consensus by a magnitude of 10 (from 30 seconds to 6 mins). We reported the findings to the Tendermint developer team. As a result, changes were made to correct the deviations and align the implementation closer to the protocol's specification. We check the fixes made by running the tests (with no changes) on the updated version. For **Raft**, we write a total of 11 unit tests. 4 of these tests describe scenarios of previously known bugs. Our tests confirm the absence of these bugs in the implementation. Furthermore, Netrix is effective in reproducing bugs when we reintroduce them. For **BFTSmart**, we write a total of 8 tests describe diverse interesting scenarios. However, we are not able to find any violations or bugs with the implementation.

5.1 Tendermint

5.1.1 Protocol and instrumentation

Tendermint relies on Gossip protocol [11] for reliable communication. In Tendermint, a set of n validators (with at the most $f < \frac{n}{3}$ faulty) receive requests from clients. Tendermint groups the requests into blocks, and validators should agree on the order of blocks. Each block is associated with a height. In each height, validators exchange messages in rounds. In round r of height h, one validator is chosen to be the proposer and sends `Propose`(h, r, bID) message. Validators acknowledge the proposal by broadcasting a `Prevote`(h, r, bID) message. When a validator v (including the proposer) receives $2f + 1$ distinct `Prevote` messages that match the proposal, the validators lock onto the proposed block bID ($lockedValue_v = bID$) and broadcast a `Precommit`(h, r, bID) message. Initially, all validators have $lockedValue_v = nil$. The block is decided (agreed upon) by a validator in height h if in round r it received $2f + 1$ distinct `Precommit` messages that match the proposal.

The existing implementation [5] of Tendermint is written in `go`. We modified the implementation's `Transport` and `Connection` interfaces to enable communication with Netrix's API. For theses changes we added 600 LOC to an existing codebase of 150k LOC. Additionally, implementing this was straightforward as the necessary abstractions were already in place.

[5] http://github.com/tendermint/tendermint

5.1.2 Unit tests for Tendermint

To describe unit test scenarios, we referred to the protocol specification as described in the original paper [5], the invariants in the proofs of the protocol, and consulted with the Tendermint developer team.

5.1.2.1 Interesting scenarios

The interesting scenarios are motivated by deviations to the synchronous execution path (no failures) of the protocol. For example, when a validator receives 2/3rd *nil* `Prevotes`, it should send a *nil* `Precommit`. We simulate this with a filter that drops `Propose` messages. As a consequence, when a validator does not receive the `Propose` message, it sends a *nil* `Prevote`. Additionally, we write a unit test to force the validators to move to round 1 and the state machines asserts that all the validators moved to round 1.

The developer can infer the property to assert, like the scenario, from the protocol specification. It can be generic safety properties (e.g., replicas do not commit on different blocks), or specific properties related to the scenario at hand. For example, if a validator sends a *nil* `Precommit` upon receiving 2/3rds *nil* `Prevotes`. We were particularly interested in exploring behaviors where validators moved to higher rounds for two reasons. First, we are able to test for protocol clauses that required messages from more than one round. Second, in production, the validators achieve consensus in the first round due to absence of faults. To explore other scenarios which are not commonly observed in production, we referred to the protocol specification and the invariants in the proof. For example, the protocol defines the following clause,

```
Upon (f+1) messages from a higher round r
  Transition to round r
```

To simulate this scenario, we isolate one validator p and do not deliver any messages from round 0. We then force the remaining validators to move to round 1. According to the protocol specification, after receiving $f + 1$ messages from the round 1, the isolated replica transitions to round 1. For the assertion, we define a custom condition `ReplicaNewRound(p,r)` which is true when we observe a round change event from the validator p for the round r.

5.1.2.2 Liveness bug

We observe that this unit test fails as the implementation does not behave according to the protocol. The failed scenario demonstrates a liveness bug as the lagging replica fails to catch up to the remaining replicas immediately. The time required for the catchup increases as the gap in rounds between the lagging replica and the remaining replica increases. The Tendermint team has acknowledged the bug and is working on fixing them.

5.1.2.3 Protocol coverage

We introduce a notion of protocol coverage to justify the expressiveness of the DSL. As mentioned above, a protocol is typically specified as a sequence of "`Upon`" clauses followed by a set of execution rules. A unit test covers a protocol clause if, in the executions explored by the unit test, the clause is satisfied at least in one replica. With our unit tests, we covered all the clauses as defined in the protocol specification for both Tendermint and Raft.

Apart from the protocol specification, the developer can derive test scenarios from the proofs of the protocols. For example, consider the following inductive invariants used in the proof of the Tendermint protocol[6]:

$$v \neq nil \wedge \exists p.precommitted(p, r, v) \rightarrow \\ \exists quorum.\forall p.p \in quorum \rightarrow prevoted(p, r, v)$$

This formula states that if a validator p `Precommit` a non nil value v in round r, then a quorum of validators should have sent `Prevote` messages for v in round r. This is an implication of the form $A \rightarrow B$, and the corresponding unit test contains filters to ensure $\neg B$, i.e., a quorum of validators do not `Prevote` on the `Proposed` block, and a state machine that reaches the fail state if it observes A, i.e., it observes a `Precommit` from any validator.

We write 15 unit tests to describe scenarios that are not commonly observed in production out of which 5 unit tests fail. The failed unit tests demonstrate scenarios where the implementation does not behave as defined by the protocol specification. However, the deviations do not lead to safety violations. As described earlier, among the failed unit tests, one demonstrates a performance bug.

5.1.2.4 Regression tests

The tendermint team independently identified 3 of the deviations captured by our unit tests and were working on changes[7] to the implementation to correct for them. 3 of the unit tests that failed on an earlier version, succeeded on the fixed newer version of the implementation. We ran the unit tests with no modification on the newer implementation.

5.2 Raft

5.2.1 Protocol and instrumentation

Raft [22] is a crash fault tolerant benign consensus algorithm. A consensus instance starts with a leader election phase followed by a replication phase. In the leader election phase, a candidate leader requests for votes (`RequestVote` messages) from all processes. Upon receiving a majority of accepting votes (`RequestVoteReply`), the candidate transitions to a leader. In the replication phase, a leader receives requests from clients, adds it to the log and replicates the log (`AppendEntries` message). A request is committed if a majority of processes add it to their respective logs.

We instrument an open source version of Raft (etcd/raft). The Raft implementation exposes the protocol primitives as a library and does not contain any communication primitives. Therefore, to build applications using the Raft implementation we need to introduce a communication layer. To integrate the Netrix library into the communication layer, we added an additional 120 LOC over the 16k LOC of the implementation.

5.2.2 Unit tests

Interesting scenarios. Similar to Tendermint, we systematically explore different fault models to explore deviating behavior. Since the system can tolerate f failures, a developer can write a test to drop f messages. For example, we expect the candidate to transition to a leader despite dropping f `RequestVoteReply` messages to a candidate. The state machine

[6] https://github.com/tendermint/spec/tree/master/ivy-proofs
[7] https://github.com/tendermint/tendermint/issues/6849,
https://github.com/tendermint/tendermint/issues/6850

describing the assertion for this unit test contains two states. From the initial state, we transition to a successful *leader-elected* state when a replica becomes the leader.

5.2.2.1 Known bugs

Many safety/liveness bugs have been reported on the raft protocol [8] [9] which lead to liveness and safety bugs. In the liveness bug, the current leader is not able to make progress and the replicas cannot elect a new leader. The safety bug occurs due to concurrent reconfiguration requests and is caused due to a split network with two valid quorums.

The scenario for the liveness bug is as follows. Consider a 5 replica system and replica 1 being the current leader. 1 is disconnected from all other processes except for replica 2 and replica 5 is completely isolated. In this scenario, 2 will not initiate a leader election as it is still connected to the current leader. A new leader (other than 1 or 2) cannot receive a majority vote as 2 is still connected to the current leader. Also, the current leader 1 cannot make progress because it is not connected to a majority of the processes (only 2 is connected). The state machine for this test transitions to success state after observing a threshold of recurrent leader election attempts. The filter drops all messages between disconnected replicas and is activated only when the state machine transitions to a particular state. Note that filters can access the state machine. We believe this allows developers to write more complex filters to explore intricate scenarios.

In the current implementation, two parameters - `Prevote` and `CheckQuorum` - when turned on overcome the liveness bug. With our tests, we have been able to check that when the parameters are turned off, the bug is present and when turned on, the bug does not occur. We skip details of the safety bug for the sake of brevity and refer the reader to the linked bug report. A total of 4 tests describe known bugs.

5.3 BFTSmart

5.3.1 Protocol and instrumentation

BFT-Smart [4, 25] is a byzantine fault tolerant consensus algorithm. The implementation is modular and can be configured to run with either crash fault tolerance or byzantine fault tolerance. The byzantine algorithm is analogous to PBFT introduced in 2. For the sake of brevity we skip the details of the algorithm.

The BFTSmart implementation[10] is in Java and has a codebase size of 16k LOC. Our instrumentation uses the Java client library and consists of 150LOC changes to the implementation. Netrix can be used to test implementations written in any language and this is evident from our unit tests for BFTSmart. Additionally, testing BFTSmart helped us develop the java client library.

5.3.2 Unit tests

The unit tests for BFTSmart are centered around moving processes to a new view. Different tests explore varying approaches of forcing a view change and subsequently check for outcomes by delivering messages from earlier views. For example, delivering a `PrePrepare` message from an earlier view along with the current one. The assertions (state machines) describe

[8] https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J
[9] https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/
[10] https://github.com/bft-smart/library

Table 3 List of unit tests. The table lists unit tests grouped by the protocol. The columns are number of filters, state machine states, LOC and outcomes in number of iterations that successfully caught the interesting scenario/bug. * indicates new bugs found and ˆ indicates tests for replicating known bugs

Name	#F	#S	LOC	Outcomes	Name	#F	#S	LOC	Outcomes
Tendermint					**Raft**				
ExpectUnlock*	3	5	90	41/100	Livenessˆ	5	3	64	15/100
Relocked*	4	5	115	53/100	LivenessNoCQˆ	5	3	64	100/100
LockedCommit*	3	5	85	100/100	NoLivenessˆ	5	3	33	100/100
LaggingReplica*	3	4	71	100/100	ConfChangeBugˆ	5	2	94	55/100
ForeverLaggingReplica*	5	5	89	100/100	DropHeartbeat	2	3	69	100/100
RoundSkip	3	4	74	90/100	DropVotes	1	3	44	80/100
BlockVotes	2	3	55	33/100	DropFVotes	1	2	57	100/100
PrecommitInvariant	1	3	68	100/100	DropAppend	1	3	81	100/100
CommitAfterRoundSkip	3	3	82	36/100	ReVote	2	3	54	74/100
DifferentDecisions	8	3	180	20/100	ManyReVote	2	4	64	92/100
NilPrevotes	2	3	61	99/100	MultiReVote	2	4	60	81/100
ProposalNilPrevote	1	3	56	56/100	**BFTSmart**				
NotNilDecide	2	2	49	100/100	DPropForP	2	3	60	81/100
GarbledMessage	1	2	68	30/100	DPropSame	2	2	40	100/100
HigherRound	1	3	91	37/100	DropWrite	1	2	30	100/100
					DropWriteForP	1	2	33	89/100
					ExpectNewEpoch	1	2	28	94/100
					ExpectStop	1	2	38	100/100
					ByzLeaderChange	3	2	46	89/100
					PrevEpochProposal	2	3	53	99/100

Table 4 List of Filter distances: table lists unit tests where filter distances are small and PCTCP is able to explore expected re-orderings. Distance - measures for each filter pair. Outcomes - number of successful iterations when we remove that filter and run PCTCP. We denote distance as an asymptotic measure where n is number of processes and r is the number of rounds.

Name	Protocol	Filter distances	PCTCP outcomes
Liveness	Raft	$\left[\infty, n^2, n\right]$	$[0, 0, 19/1000]$
LivenessNoCQ	Raft	$\left[\infty, n^2, n\right]$	$[0, 0, 19/1000]$
DropAppend	Raft	$[n]$	$[21/1000]$
ReVote	Raft	$[n]$	$[409/1000]$
ManyReVote	Raft	$[2n]$	$[14/1000]$
MultiReVote	Raft	$[3n]$	$[1/1000]$
RoundSkip	Tendermint	$[r \times n]$	$[6/1000]$

the expected outcomes along with generic safety properties. To simulate such view changes, our filters drop messages, re-order proposals and introduce byzantine failures.

We do not observe any deviations from the protocol specification of BFTSmart nor did we uncover any bugs. This was expected as BFTSmart is a well tested implementation that is robust and reliable. However, we would like to highlight that the tests allowed us to debug and inspect the implementation and check its conformance with the protocol specification without delving deep into understanding the codebase.

5.4 Filter distance

While some of the filters help impose constraints on the execution space that are absolutely necessary (byzantine faults), the other only aid exploration to obtain better outcomes in terms of number of successful iterations. Allowing PCTCP like algorithms to explore a diverse set of executions is desirable to gain confidence in the correctness of the implementation. We define a distance metric to be associated with filters. The developer can forego writing filters of short distance and expect PCTCP to explore the required message re-orderings.

From our unit tests, we identified three categories the filters can be grouped into — (1) **Byzantine** - Filters that introduce byzantine behavior, (2) **Drop** - Filters that drop messages, (3) **Reorder** - Filters that reorder messages. The Reorder filters can be grouped into pairs. Ones that capture the message and store it in a set and ones that release the messages stored in a set. The metric is defined for a pair of capture-release filters and can be determined syntactically. The Drop filters can be considered as a pair, where the release filter releases messages at the end of the execution; Byzantine filters have infinite distance by definition. In consensus protocols, messages are associated with rounds/phases. We define the distance as the number of messages between the capture and release filter in a normal execution of the protocol (without any faults). For example, if the capture filter corresponds to messages of round r and the release filter corresponds to message of round $r + 2$. The distance is then $2n$ where n (linear in number of processes) messages are sent each round. We observe that PCTCP fails to explore executions where the re-orderings are beyond a certain distance threshold. However, the filters with short distance measures (1-2 communication rounds) are not crucial and PCTCP is able to explore the respective re-ordering. Table 4 includes the results of some of the tests for which our experiments provided non zero outcomes.

5.5 Summary

We write a total of 34 unit tests for all the three protocols motivated by the protocol specification, proofs and reported bugs. As described in Table 3, all our unit tests are compact (<200 LOC) and require a small number of filters, state machine states. The tests produce capture the interesting scenario or bug with high probability. Existing tools however do not provide the same level of control and are limited in the kind of properties that can be tested for. As Netrix is modular, developers can integrate and automate testing with Netrix into their development workflow. In addition, Netrix empowers the developer to write tests without any formal model. The Tendermint team is integrating the tool into their development process and writing unit tests for planned changes. In addition, to answer Q2, we write tests to reproduce known bugs in Raft and find new ones in Tendermint.

6 Related work

Testing implementations of distributed systems has received considerable attention over the recent years. Probabilistic Concurrency Testing with Chain Partitioning (PCTCP) [23, 24] provides precise probabilistic guarantees about observing every possible order between a fixed number of events. The Jepsen tool [1] makes it possible to introduce benign faults randomly with a certain frequency, which provides very little control over the outcome of a test. Observing a bug or not depends on the interaction between the fault injection frequency and the scheduler which is not controllable. Jepsen has demonstrated its success empirically by finding bugs in well known systems such as Cassandra [17] and Redis [18]. Similarly, developers also use simulation-based testing where they run many heavily instrumented replica instances on a single machine and randomly introduce message drops or network partitions. This process can be enhanced with simulated virtual clocks to speed up or slow down replicas [2]. Both Jepsen and PCTCP does not offer any control over the executions that are explored. Furthermore, unlike in our approach, these techniques cannot explore executions that include byzantine failures.

MoDist [27], SAMC [19] and CrystalBall [26] adopt model checking techniques to test distributed systems implementations and exhaustively explore the execution space. These techniques require that tests be run for hours on compute intensive hardware (48 full machine days with SAMC). They cannot deal with Byzantive faults. Moreover, differently from these works, Netrix makes it possible to program the amount of asynchrony or faults in the executions, which simplifies the process of root causing and debugging potential violations.

Our work is inspired by Concurrit [15], which enables a similar scenario-based testing approach for *multi-threaded* concurrent programs. It introduces a DSL that enables developers to define tests where they can control the scheduling between threads with a minimal instrumentation effort. This DSL is specific to multi-threading and very different compared to Netrix's DSL which is specific to testing implementations of consensus protocols. GFuzz [12] applies the idea of exploring different message orderings between concurrent go channels and has demonstrated success in finding concurrency bugs in actual implementations. P# [9] is an actor based programming language that allows developers to write asynchronous systems. P# is embedded in the C# programming language and is accompanied by a systematic concurrency testing framework. Similar to GFuzz, P# explores arbitrary event orderings between the actors to find concurrency bugs. However, both GFuzz and P# do not allow describing specific scenarios to test.

Our DSL primitives are motivated by specification languages for protocols. DISTAL [8] programs are a sequence of *Upon* clauses. Each *Upon* is followed by a predicate on the state of the protocol and current message. Similar to our DSL, DISTAL predicates contain counting, sets of messages and comparing message types. ModP [13] language allows protocol designers to describe and test a model of the protocol. Similar to DISTAL, ModP machines contains a sequence of *on* event handlers that modify the state of the machine. The *on* handlers are followed by predicates similar to DISTAL. ModP also generates code for testing the programs. While these are effective in finding bugs in a *model* of a protocol, the results however do not help in testing production implementations. The main reason they do not help in production environments is because model checkers do not scale when applied directly on implementation of large systems. Therefore our DSL provides the only guided way to do exploration of the execution space on implementations.

6.1 Comparison in instrumentation effort

MoDist adds an interposition layer between the replica and the operating system and it is limited to distributed systems that run on the Windows operating system. Similarly, Jepsen relies on manipulating `iptables` rules to control communication between replicas. Jepsen, PCTCP, and MoDist require developers to write initialization scripts. Replicas are instantiated using the scripts and allow the tools to control (1) replica processes and (2) the communication between replicas. Netrix like PCTCP requires complete control over the messages but, unlike PCTCP, allows replicas to communicate the messages via RPC. Netrix delegates the responsibility of running the test network to the developers. As a result, it is easy to instrument and test all distributed systems that communicate messages via RPC.

7 Conclusion

Distributed systems suffer from complex and intricate bugs. Existing tools adopt a black box testing approach and rely on either systematic or probabilistic exploration of possible executions. We propose a more guided scenario-based unit testing approach where developers program and reuse tests aided by probabilistic exploration techniques to obtain better bug reproducibility. To facilitate this, we introduce a domain specific language to describe unit tests and define its syntax and semantics. Furthermore, we build an open source tool Netrix that is based on the domain specific language. We use Netrix to instrument and test Tendermint [5], Raft [21] and BFTSmart [4, 25]. Our unit tests are effective in capturing deviations from the specification in the implementations. Furthermore, we demonstrate the re-usability of the tests by running them on different versions of the implementations and checking the bug fixes made in the implementation.

When testing consensus protocols, we must ensure that we do not violate the fault model. For example, in Tendermint, we must ensure that the number of faults introduced does not exceed f for $n \geq 3f + 1$ replicas. Currently, the developer writing the unit tests is responsible for constraining the number of faults introduced. Possible future work can identify and restrict the number of introduced faults based on configurable network assumptions.

As demonstrated, the protocol specification provides the developer insight into describing unit test scenarios to test the functional correctness of the implementation. Another avenue for future work is automated generation of interesting test scenarios given a protocol specification (and its formal correctness proof).

References

1 Jepsen, 2020. URL: `https://jepsen.io`.

2 Viewstamped Replication made famous, 2020. URL: `https://github.com/coilhq/viewstamped-replication-made-famous`.

3 Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014. `doi:10.1145/2535838.2535845`.

4 Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 355–362. IEEE Computer Society, 2014. `doi:10.1109/DSN.2014.43`.

5 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018. URL: `http://arxiv.org/abs/1807.04938`, `arXiv:1807.04938`.

6 Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGARCH Comput. Archit. News*, 38(1):167–178, mar 2010. `doi:10.1145/1735970.1736040`.

7 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

8 Pamela Delgado. Distal: Domain-specific language for implementing distributed algorithms. 2012. URL: `http://infoscience.epfl.ch/record/187164`.

9 Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. *SIGPLAN Not.*, 50(6):154–164, jun 2015. `doi:10.1145/2813885.2737996`.

10 Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 249–262. USENIX Association, 2016. URL: `https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis`.

11 Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, page 1–12, New York, NY, USA, 1987. Association for Computing Machinery. `doi:10.1145/41840.41841`.

12 Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. `doi:10.1145/3276529`.

13 Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. `doi:10.1145/3276529`.

14 Cezara Dragoi, Constantin Enea, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. Testing consensus implementations using communication closure. *Proc. ACM Program. Lang.*, 4(OOPSLA):210:1–210:29, 2020. `doi:10.1145/3428278`.

15 Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. Concurrit: A domain specific language for reproducing concurrency bugs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 153–164, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2491956.2462162`.

16 Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code (awarded best paper). In Hari Balakrishnan and Peter Druschel, editors, *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*. USENIX, 2007. URL: `http://www.usenix.org/events/nsdi07/tech/killian.html`.

17 Kyle Kingsbury. Jepsen: Cassandra, Sep 2013. URL: `https://aphyr.com/posts/294-call-me-maybe-cassandra`.

18 Kyle Kingsbury. Redis-raft 1b3fbf6, Jun 2020. URL: `https://jepsen.io/analyses/redis-raft-1b3fbf6`.

19 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, page 399–414, USA, 2014. USENIX Association.

20 Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar Heri Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. Flymc: Highly scalable testing of complex interleavings in distributed systems. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 20:1–20:16. ACM, 2019. `doi:10.1145/3302424.3303986`.

21 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

22 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

23 Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. `doi:10.1145/3276530`.

24 Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.*, 3(OOPSLA):180:1–180:29, 2019. `doi:10.1145/3360606`.

25 João Sousa and Alysson Neves Bessani. From byzantine consensus to BFT state machine replication: A latency-optimal transformation. In Cristian Constantinescu and Miguel P. Correia, editors, *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 37–48. IEEE Computer Society, 2012. `doi:10.1109/EDCC.2012.32`.

26 Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In Jennifer Rexford and Emin Gün Sirer, editors, *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 229–244. USENIX Association, 2009. URL: `http://www.usenix.org/events/nsdi09/tech/full_papers/yabandeh/yabandeh.pdf`.

27 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 *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, page 213–228, USA, 2009. USENIX Association.

28 Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 317–335. Springer, 2018. `doi:10.1007/978-3-319-96142-2\_20`.