Testing consensus implementations using communication closure

Cezara Dragoi, Constantin Enea, Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic

Testing Consensus Implementations Using Communication Closure

CEZARA DRĂGOI, INRIA, France and Informal Systems
CONSTANTIN ENEA, IRIF, France
BURCU KULAHCIOGLU OZKAN, MPI-SWS, Germany
RUPAK MAJUMDAR, MPI-SWS, Germany
FILIP NIKSIC, University of Pennsylvania, USA

Large scale production distributed systems are difficult to design and test. Correctness must be ensured when processes run asynchronously, at arbitrary rates relative to each other, and in the presence of failures, e.g., process crashes or message losses. These conditions create a huge space of executions that is difficult to explore in a principled way. Current testing techniques focus on systematic or randomized exploration of all executions of an implementation while treating the implemented algorithms as black boxes. On the other hand, proofs of correctness of many of the underlying algorithms often exploit semantic properties that reduce reasoning about correctness to a subset of behaviors. For example, the *communication-closure* property, used in many proofs of distributed consensus algorithms, shows that every asynchronous execution of the algorithm is equivalent to a *lossy synchronous* execution, thus reducing the burden of proof to only that subset. In a lossy synchronous execution, processes execute in lock-step rounds, and messages are either received in the same round or lost forever—such executions form a small subset of all asynchronous ones.

We formulate the *communication-closure hypothesis*, which states that bugs in implementations of distributed consensus algorithms will already manifest in lossy synchronous executions and present a testing algorithm based on this hypothesis. We prioritize the search space based on a bound on the number of failures in the execution and the rate at which these failures are recovered. We show that a random testing algorithm based on sampling lossy synchronous executions can empirically find a number of bugs—including previously unknown ones—in production distributed systems such as Zookeeper, Cassandra, and Ratis, and also produce more understandable bug traces.

1 INTRODUCTION

Large-scale, fault-tolerant, distributed systems are the backbone for many critical software services. Since they must execute correctly and efficiently in the presence of concurrent and asynchronous message exchanges as well as benign (message loss, process crash) or Byzantine failures (message corruption), the underlying algorithms are intricate. Moreover, even when the algorithms are proven correct, testing production *implementations* of these algorithms remains a significant challenge, precisely because of the enormous number of exceptional conditions that may arise in production.

Testing such distributed systems raises several important challenges:

- **(C0)** *Test oracle*: Formulating a correctness specification that should hold for the system and a checker for the property on a given execution.
- **(C1)** *Test harness discovery*: Devising a suitable set of test harnesses (combinations of user requests) that are more likely to expose vulnerabilities, e.g., sets of transactions that access a common set of data fields in the case of a distributed database.
- **(C2)** *Enumerating executions:* Even if the test harness contains few user requests, the number of possible executions can still be enormous because of a large number of internal steps that can interleave in arbitrary ways (the number of executions can be infinite if failures occur

Authors' addresses: Cezara Drăgoi, INRIA, France , Informal Systems; Constantin Enea, IRIF, France; Burcu Kulahcioglu Ozkan, MPI-SWS, Germany; Rupak Majumdar, MPI-SWS, Germany; Filip Niksic, University of Pennsylvania, USA.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

frequently and infinitely-often). An important challenge is to define efficient strategies for enumerating the execution space that maximizes the probability of exposing vulnerabilities.

(C3) *Improving interpretability:* Since a vulnerability can be exposed in many different ways, it is desirable to prioritize showing the user executions that are "easily" interpretable and that simplify the task of extracting the root cause and a possible repair.

Specifications for distributed systems is a well-studied topic, e.g., [Lynch 1996], and our paper assumes that a correctness specification is provided. We shall focus on the challenges **C1**–**C3**.

Challenge **C1** is usually addressed using an exhaustive enumeration of test harnesses with few user requests. Empirically, these harnesses seem to be enough for exposing most vulnerabilities (an instance of the so-called "small scope" hypothesis). Therefore, testing techniques today focus on addressing the challenge **C2** and explore message orderings, systematically or randomly, a major concern being to prioritize the search order [Desai et al. 2015; Izrailevsky and Tseitlin 2011; Killian et al. 2007; Kingsbury 2018; Leesatapornwongsa et al. 2014a; Lukman et al. 2019a; Ozkan et al. 2018]. In most existing testing approaches, the underlying distributed protocols are treated as black boxes: tests explore possible schedules of messages and faults in the implementation without considering properties of the underlying algorithms. Up to our knowledge, none of the existing techniques address the challenge **C3** of improving interpretability.

In this paper, we describe a testing strategy that addresses both **C2** and **C3**. We pick a subset of executions of a distributed system that, under some reasonable and frequently occurring assumption on the underlying algorithms, represents every other possible execution (any other execution is equivalent to one in this subset). The subset of executions is chosen to follow a symmetric and regular scheduling policy, e.g., synchronizing message exchanges between different processes. Our testing strategy explores only this subset of executions, and it is complete in the limit under the hypothesis that the semantic reduction holds. Since it explores concrete executions of the system it is clearly sound, in the sense that all reported bugs are genuine. The restriction to a subset of executions improves the likelihood that a bounded enumeration is able to expose vulnerabilities (challenge **C2**) while restricting the scheduling policy improves interpretability (challenge **C3**). This semantic reduction is mainly based on a property called *communication-closure* [Elrad and Francez 1982], which has been used extensively in designing or proving distributed protocols like Paxos [Chou and Gafni 1988; Damian et al. 2019; Dragoi et al. 2016; Moses and Rajsbaum 2002; von Gleissenthall et al. 2019].

A Semantic Reduction Based on Communication-Closure We model a fault-tolerant distributed system as a set of processes communicating through message passing. Each process maintains local state and executes a sequence of send, receive, and state update actions. Under the standard asynchronous semantics, processes may execute at arbitrarily different speeds and messages can be arbitrarily delayed or lost (process crashes can be modeled as losing all messages sent by or to a process). The space of possible executions is enormous since it is defined by all the interleavings between process actions and all possible ways of introducing message delays or losses.

As stated above, we consider a semantic reduction for such systems which is based on *communication closure*. This property relies on a restricted semantics, that we call *lossy synchronous* [Charron-Bost and Schiper 2009; Gafni 1998; Santoro and Widmayer 1989], and ensures that every asynchronous execution is *indistinguishable* from a lossy synchronous execution. Indistinguishability means that processes go through the same sequence of local states, modulo stuttering, in the two executions. Assuming that the system specification cannot make the difference between indistinguishable executions, which is the case in practice for many specifications of interest, communication-closure ensures that exploring only lossy synchronous executions is complete.

While our method is not complete for systems that violate communication closure, it is sound (any reported bug is a true bug).

To define the lossy synchronous semantics, we consider that the behavior of each process is structured as a sequence of *rounds*: sequences of send-receive-update actions (this decomposition can be assumed without loss of generality modulo introducing fictitious actions for sending/receiving an empty set of messages and update actions leaving the state unchanged). For example, in a distributed consensus protocol, rounds correspond to preparing a new ballot/view/term, sending and receiving acknowledgments, proposing values, and communicating promises. The lossy synchronous semantics imposes that processes execute rounds synchronously and in lock-step, but messages can be lost. Any two processes are in the same round at each point during the execution and all messages sent in a round are either received in the same round or lost forever (messages exchanged in one round may be lost while the ones exchanged in the next round delivered without failure). In contrast, under the asynchronous semantics, processes may be executing different rounds at a point of time and be ready to receive messages from any round in the past or future.

We reduce the execution space even further for "leader-based" protocols, a widely used technique for implementing state machine replication. In a leader-based protocol, the communication in each round goes from one process, called *leader*, to all the other processes, or from all processes to the leader. We introduce a restriction of the lossy synchronous semantics, which restricts the way messages are lost in a given round. We define a *uniform* lossy synchronous semantics where the messages that are lost in a given round are precisely those sent or received by a set of processes. Intuitively, this corresponds to isolating each such process from all the other processes in the network. This is a restriction of the lossy synchronous semantics. For instance, in the presence of three processes p_1, p_2, p_3, the uniform semantics does not allow that a message from p_1 to p_2 is lost while a message from p_1 to p_3 is delivered, or it does not allow that a message from p_2 to p_1 is lost while a message from p_3 to p_1 is delivered (p_1 is not isolated from all the other processes, but only from p_2). It is rather easy to see that the uniform lossy synchronous semantics is complete for leader-based protocols (we show in Section 5 that it is complete for a larger class of protocols).

Our testing algorithm enumerates only executions under the uniform lossy synchronous semantics. While proving the validity of the reduction to such a semantics (i.e., that our testing algorithm is complete in the limit) is very difficult for production systems (the kind we consider in the experimental evaluation), the goal of our work is investigating the following *uniform communication-closure hypothesis*: bugs in many distributed systems manifest already at the level of uniform lossy synchronous executions. The validity of this hypothesis leads to a solution for challenge **C2** since the space of uniform lossy synchronous executions is much smaller than the whole set of asynchronous executions (see Section 2 for an example) and challenge **C3** because the exchange of messages in such executions is quite regular and easy to interpret in comparison to an arbitrary asynchronous execution. While it is hard to evaluate the degree of interpretability in an objective manner, we believe through our own experience that the simple communication patterns in uniform lossy synchronous executions, the lock-step exchange of messages in particular, are definitely easier to debug than an arbitrary schedule of such actions.

Testing Algorithm We define a randomized testing algorithm which samples uniform lossy synchronous executions. The algorithm takes as input a harness consisting of n processes running for a maximum of r rounds. Our algorithm limits the sampling space according to several parameters that bound the choice of isolated processes in each round. Note that process isolation is the only source of non-determinism in the uniform lossy synchronous semantics since processes execute rounds in lock-step (the interleaving between actions of different processes is fixed modulo actions which commute trivially like sends done in parallel by two different processes). The first parameter is a bound d on the number of isolated processes across all the rounds in the execution while the second parameter k sets the frequency at which isolated processes re-join the network.

While the choice of the parameter d is motivated by an empirical "small scope" observation that many bugs in implementations already occur under a rather small number of isolated processes (transient faults), the second parameter k is motivated by the structure of standard distributed algorithms, e.g., state machine replication algorithms. Typically, the sequence of rounds in a process is further decomposed into a sequence of *phases* (a phase is a sequence of rounds) with *successful phases*, when the system makes progress towards its specification, and *unsuccessful phases*, when progress is not possible because of failures (e.g., message loss), but some computation needs to be performed to ensure that the system remains safe. For example, in a state machine replication algorithm, a successful phase corresponds to committing a single command (transition) of the machine, provided that enough messages are delivered in each of its rounds. In more faulty scenarios, i.e., when the network is temporarily partitioned such that there is no majority that can communicate reliably, the system will execute several unsuccessful phases until the network delivers sufficiently many messages in a phase to commit a client request. The desirable choice for the rate k at which processes re-join the network equals the length of a phase in the system under test. The testing algorithm uses k and d to generate executions that alternate successful phases (having few to no processes isolated) and unsuccessful phases (having sufficiently many processes isolated to prevent progress). However, the user is not required to have protocol specific insights about the length of a phase. The testing algorithm drives the exploration through executions where the set of isolated processes changes at every k rounds. The sampling space grows as d is increased and k is decreased, covering the whole space of uniform lossy synchronous executions when d grows to infinity and $k = 1$.

Our algorithm samples executions of the harness satisfying the bounds d and k, and guarantees that each execution is picked with a certain minimum probability. This leads to precise probabilistic guarantees about hitting a specific execution. This algorithm is sound, i.e., the reported bugs are not spurious, and complete in the limit when the reduction to the uniform lossy semantics is valid.

Evaluation We evaluated the effectiveness of our testing algorithm on large scale distributed systems such as Cassandra, Ratis, and Zookeeper. Our evaluation focuses on detecting consistency violations, a major source of bugs in distributed systems. We experimentally show that our testing algorithm (1) compares favorably with testing based on random search: it detects several known and novel bugs by sampling from a much smaller subset of executions (showing that uniform lossy executions already cover many bugs), and (2) enables exploration even with little instrumentation of the source code. In particular, our testing tool was able to detect several previously unknown bugs in recent versions of Zookeeper and Ratis. Moreover, the buggy traces produced by our algorithm are informative. The synchronous traces are more understandable when compared with the usual asynchronous ones produced by other state-of-the-art techniques.

The generality of our method goes beyond the evaluated benchmarks. Distributed systems are all about coordination in the absence of a global clock. Communication-closure highlights rounds, an encoding of a local notion of time which is used by processes to coordinate and accomplish collective tasks. Rounds are a good abstraction of timestamps, vector clocks, or any other synchronizations mechanism that must be implemented by a distributed protocol. The communication-closed executions of a systems are the core of any protocol (even if the protocol has not been shown communication-closed), because they include the executions for which local time can be mapped on a global notion of time. Therefore, even for systems where our testing is not complete, prioritizing communication-closed executions is an important heuristic.

Contributions and Outline In this paper we propose a framework for reducing the search space in testing based on communication-closure, a well established design and reasoning principle for fault-tolerant distributed systems.

Our testing framework complements theoretical concepts from the distributed computing community (communication closure) with novel search prioritization and randomization techniques (which are specific to the use of communication closure and the systems under study). Despite the fact that communication closure is a rather established and well-studied concept in theoretical terms, it has never been proposed as a way of building better testing tools. Our work transfers the theoretical insight to testing tools that find bugs in real-world, deployed, applications.

Our contributions and outline are summarized as follows:

- we develop a theoretical framework for stating and using the communication-closure hypothesis in testing (§3 and §4),
- we define the uniform restriction of the lossy synchronous semantics prescribed by communication-closure which limits message losses to isolating a set of processes and which is complete for a large class of practical distributed algorithms (§5),
- we define a randomized testing algorithm with precise probabilistic guarantees that samples, uniform lossy synchronous executions under certain bounds on the occurrence of network link failures (§6)
- we conduct an empirical evaluation on production distributed systems (§7).

2 OVERVIEW

We demonstrate our testing framework on the distributed protocol listed in Fig. 1, where a set of processes must agree on a total order between a set of commands. These commands are inputed one by one while the protocol is running, and possibly concurrently, at different processes at the same time. This is a simplified version of state machine replication (based on Paxos [Lamport 2005]) in which we omit how commands are communicated to the protocol and assume that they are created by invoking a `new_command` function (see line 28). Each process maintains a sequence of commands (in a local variable `log`) which is outputted when certain conditions are fulfilled (see line 42). The intended specification is that any two such outputs, possibly from different processes or from the same process but at different points in time, must be comparable with respect to the standard prefix order between sequences. The protocol would be incorrect if for instance, two processes would output *a* and *b*, respectively (since neither is a prefix of the other one).

The pseudocode in Fig. 1 is executed an arbitrary number of times by each process participating in the protocol, and an execution of the protocol is a standard interleaving of steps from different processes. Like in many other distributed protocols, each process executes a sequence of *rounds*. Each round consists of a sequence of message sends, receives, and state updates, in this order. The protocol periodically tries to extend the sequence of commands on which processes agree with a new command by executing a sequence of four rounds, called *phase*,[1] in each process. In each phase, a process, called the leader, gets a new command and tries to store into the log of a quorum formed of more than half of the processes. The quorum is essential for fault tolerance. If all processes execute synchronously (in lockstep) and all messages are delivered, then each process ends up extending their local sequence `log` with the new command. Such an execution is given in Fig. 2(a). Each process in this execution executes two phases: the first phase appends command `a` while the second appends command `b`. The possible outputs are related by prefix order as expected. If too many messages are lost (the cardinality constraints at lines 26 and 41 that check for a quorum are not satisfied) while a process is executing a phase to process a new command, then its `log` remains

[1] In other works, a phase may be called *ballot* or *view*.

```
1  //Local variables
2  int last = phase = 0
3  var log = ϵ
4  var my_id, leader
5  var step
6
7  //@Round Prepare
8  //@Snd:
9  if (getLeader(phase) == my_id)
10    send to all ("Prepare", phase+1, my_id)
11   receive_messages()
12 //@Upd:
13 if received m=("Prepare", m.phase, m.sender)
14       with m.phase >= phase
15   last = phase //@bugfix remove
16   phase = m.phase
17   leader = m.sender;
18   step = "Ack"
19
20 //@Round Ack
21 //@Snd:
22 if(step=="Ack") send to leader
23  ("Ack", phase, (last, log))
24 receive_messages()
25 //@Upd:
26 if (step=="Ack") && received > n/2 messages ("Ack",phase,_)
27   log = select_log_from_received_messages()
28   log = log @ new_command()
29   step = "Propose"
30 if(my_id != leader) step = "Propose"
```

```
25 //@Round Propose
26 //@Snd
27 if (leader == my_id && step == "Propose")
28   send to all ("Propose", phase, log)
29 receive_messages()
30 //@Upd:
31 if received from leader a message
         m=("Propose",phase, m.log)
32   log = m.log
33   step = "Promise"
34   //@bugfix add last = phase
35
36 //@Round Promise
37 //@Snd:
38 if(step == "Promise") send to all
         ("Promise", phase, log)
39 receive_messages()
40 //@Upd:
41 if received more than n/2 messages
         ("Promise", phase, log)
42   output log
```

Fig. 1. A Paxos-like state machine replication protocol. The number of processes participating in the protocol is denoted by n. Each process has a number of local variables, listed at lines 2–5: `my_id` is a constant storing the id of the process, and `log` stores the sequence of commands to be outputted (@ denotes sequence concatenation at line 28). The code represents a *phase* defined as a sequence of four *rounds*. Each round consists of message sends (annotation `@Snd`), receives, and state updates (annotation `@Upd`). Each phase has a designated *leader* which is set by the call to the deterministic getLeader function.

unchanged and it begins a new phase (the same happens if the process executes much faster than many other processes). Although the protocol should tolerate any such exceptional conditions and satisfy the intended specification, this is *not* actually true.

Fig. 3 shows an execution that violates this specification where the processes output sequences a and b, which are incomparable w.r.t. prefix order. This execution contains four phases: during the first phase, enough messages are delivered so that two processes can output a; many messages are lost in the next two phases, so no process can extend their `log`; enough messages are delivered during the fourth phase, but processes end up "forgetting" about command a, and output the singleton sequence b.

In order to understand the details of the bug in Figure 3, we take a closer look at the implementation. Each process keeps track of the current phase it executes (using the local variable `phase`). Due to faults processes may be in different phases. In each phase a processes executed the four rounds in Fig. 1; the rounds are named in comments (lines 7, 20, 25, and 36). In the first round the leader looks for a quorum of processes to learn the most up-to-date log stored by its peers (the leader might have a stale local log due to faults). To this, the leader broadcasts a `Prepare` message that contains the leader's phase (line 10). The processes that receive the leader's message join the leader's phase by updating the local `phase` variable to the leader's phase, unless they are already in

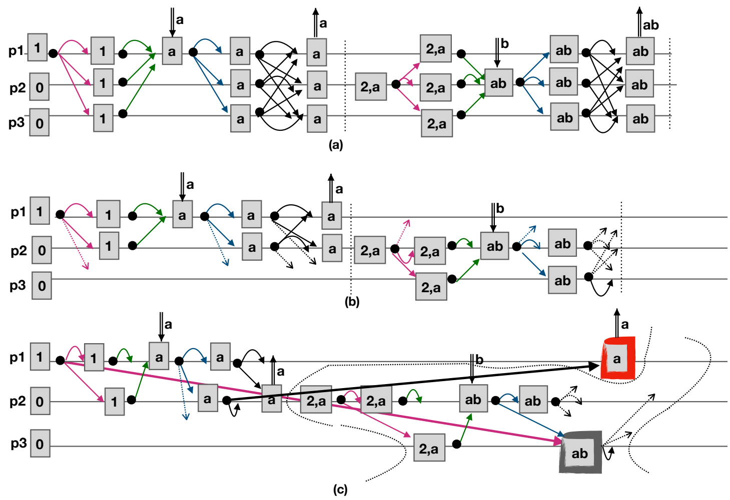

Fig. 2. Three executions of the protocol in Fig. 1 that involve three processes p1, p2, and p3: (a) a synchronous execution where no message is lost, (b) a lossy synchronous execution, (c) an asynchronous execution indistinguishable from the lossy synchronous execution in (b). Each horizontal line shows time progressing for each process. Boxes contain fragments of local state: the numbers represent the value of the phase variable while the strings represent the value of `log`. Colored arrows between the horizontal lines show the messages exchanged. Dotted arrows in (b) and (c) indicate dropped messages. Double arrows ⇓ denote input commands (values returned by `new_command`) while ⇑ denote output command sequences. Each phase ends with a vertical dotted line in the figures.

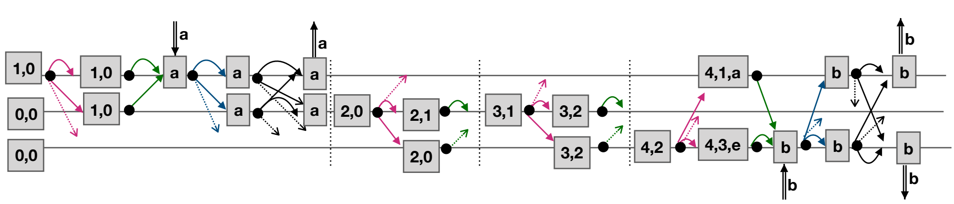

Fig. 3. An incorrect execution of the protocol in Fig. 1 (with three processes). We use the same conventions as in Fig. 2. The pairs of integers, e.g., (1,0) and (2,0), represent the values of the local variables (phase, `last`). The log values are e, a and b, where e denotes the empty log.

a higher one (line 16). The processes that join the leader's phase, called followers, store in `last` (line 15) the number of the last phase they participated in. In the second round, each follower sends an `Ack` message to the leader (line 23), including the values of its local sequence `log` and the `last` phase the follower participated in, which is used to date the sent log. If the leader receives more than $n/2$ `Ack` messages, it has a quorum and sufficient information to compute the most recent value of the log: it selects the log coming from the process that participated in the most recent phase, i.e., the one with the highest value of `last` (line 27). In the third round, the leader sends the most recent log extended with a new command to all processes in a `Propose` message (line 28). The processes that receive the leader's message update their log accordingly. In the last round, processes exchange their current log and phase, by sending `Promise` messages to all the processes (line 38). A process that receives $n/2$ `Promise` messages for the same phase and the same value of the log, outputs this log value.

The value of the last phase a process participated in is sent along with the value of the log in the round Ack. This is crucial for correctness because it prevents losing requests. The bug in Fig. 3 is caused by an incorrect computation of the most recent log after two phases when too many messages were lost. In the fourth phase the leader receives two log values in round "Ack": a from p1 and the empty log from p3. The leader picks the empty log of $p3$ as being the most recent one, because the last phase number accompanying it is higher than the last phase number accompanying the log of p1 containing a. The bug happens because of a misinterpretation of what "participating" in a ballot means. Processes should recall the last phase when they received a new log from the leader, not the last phase they joined. Process p3 joins phases 2 and 3 but its algorithmically meaningful state, i.e., the log, does not change in these phases. Therefore by updating the value of `last` when receives a `Prepare` message from the leader, p3 incorrectly makes its log more recent than it is. A correct implementation requires removing the update of `last` from the round `Prepare` (line 15) and adding an update of `last` to `phase` in round `Propose` when the process receives the leader's new log proposal, in line 34.

Asynchronous vs. Lossy Synchronous Semantics The standard asynchronous semantics of this protocol allows arbitrary interleavings of steps from different processes under a non-deterministic network that can drop arbitrarily many messages. Different processes may execute different rounds at the same time and they may receive arbitrarily delayed messages. For example, Fig. 2(c) shows an asynchronous execution where some messages are lost and others are delayed. In this execution, processes go through two phases: in the first one, the leader p1 transmits the command a to $\{p1, p2\}$, and in the second phase, the leader p2 transmits the second command b to $\{p2, p3\}$. The non-determinism in this semantics due to scheduling and message loss leads to an enormous number of executions. Standard exhaustive or random enumerations of this space of executions are very unlikely to be effective in exposing potential vulnerabilities like the one in Fig. 3.

A smaller space of executions can be defined by considering a *synchronous* semantics in which each round is executed at the same time by all processes and every message is delivered. An execution fragment where each process executes a round is called a *synchronized round*. Fig. 2(a) shows such an execution with 8 synchronized rounds. This semantics is however too restricted since it cannot exercise the protocol's capabilities of tolerating faults, e.g., message loss.

An intermediate point is the *lossy synchronous* semantics, which is a weakening of the synchronous semantics where messages can be dropped arbitrarily. An execution under this semantics is still a sequence of synchronized rounds, but messages can be either delivered in the same synchronized round they were sent or dropped and never delivered in the future. Fig 2(b) shows such an execution: the leader p1 of the first phase sends the command a but only p1 and p2 receive it, and in the second phase, the command b sent by the leader p2 is received only by p2 and p3.

In general, the lossy synchronous semantics contains a subset of the possible executions (under the asynchronous semantics). However, most distributed protocols are designed to be *communication-closed*, i.e., so that the two semantics are equivalent (every asynchronous execution is equivalent to a lossy synchronous one) [Dragoi et al. 2016; Elrad and Francez 1982]. The protocol in Fig. 1 is communication-closed. For example, the asynchronous execution in Fig. 2(c) is equivalent to the one in Fig. 2(b), in the sense that each process passes through the same sequence of local states in both executions.

The key observation in our testing algorithm is that, when testing for a given specification, we can restrict attention only to lossy synchronous executions, instead of the much larger class of asynchronous executions. Note that the bug in Fig. 3 is an incorrect lossy synchronous execution. This execution represents a large class of equivalent asynchronous executions (all bugs). When the underlying protocol is communication-closed, there is no loss of generality.

Uniform Lossy Synchronous Semantics In fact, our testing algorithm considers a further restriction of the lossy synchronous semantics, called *uniform*, which limits the choice of messages to be dropped in a synchronized round. Consider for instance the first synchronized round in Fig. 3. Choosing to drop the message from the first to the third process is the same as choosing to isolate the third process from the rest of the processes (meaning that all the messages sent by or to the third process are dropped). Equating dropping messages with isolating a set of processes is valid for synchronized rounds where a single process sends messages or when all messages are sent to the same process. In practice, to minimize the number of exchanged messages, many distributed protocols are defined in such a way, each round (phase) having a designated *leader* (like in the first three rounds in Fig. 1). For synchronized rounds with a different communication structure, e.g., the last round in our protocol, choosing only to isolate a set of processes instead of dropping a specific set of messages *may* be a restriction that leads to incompleteness. For instance, the last synchronized round in Fig. 3 corresponds to isolating the second process. Dropping another message, say from p1 to p3, could not be simulated as a set of isolated processes. As we show in Section 5 this restriction is actually complete even for this protocol.

Our Testing Algorithm. Our testing algorithm randomly samples uniform lossy synchronous executions where the number of isolated processes in the run is at most d and where every isolated process can reconnect to the network every k-th round. The values of d and k are inputs to the algorithm. Intuitively, d is a bound on the number of messages that can be dropped during an execution while k should ideally correspond to the number of rounds in a phase of the protocol (this is however not a requirement and k can be arbitrary). The latter is motivated by the fact that in many algorithms, once a process becomes isolated in a phase, it cannot make progress (change its local state) during the same phase even if it reconnects later. For the protocol in Fig. 1, if a process does not receive a “Prepare” message in the first round, it cannot change its state because of messages received in the later rounds of the same phase. As d increases and k decreases, the algorithm covers more and more of the execution space. For each execution, the algorithm applies a user-provided procedure for checking the intended specification.

Advantage of Our Algorithm: Smaller Sample Set of Executions. Sampling from uniform lossy synchronous executions reduces the size of the sample set of executions significantly. Consider the protocol execution in Fig. 1 with 3 processes, 16 synchronized rounds, and $k = 4$ (these constraints are those satisfied by the buggy execution in Fig. 3; the picture omits the last two rounds from the second and third phase because no process sends any message). The number of uniform lossy synchronous executions of the protocol is about 10^7 (each one of 3 processes can be isolated at one of $k = 4$ rounds in $16/4 = 4$ phases). In comparison, the number of lossy synchronous executions which are not necessarily uniform is about 10^{43} (any subset of the 9 communication links between the processes can be lossy in a round).

3 DISTRIBUTED PROTOCOLS

We describe the theoretical foundation of our work in the context of an abstract notion of protocols that abstracts away from a particular syntax. We define the standard asynchronous semantics for such protocols, which allows arbitrary interleavings of steps from different processes and arbitrary loss of messages.

Protocols. We fix a set $\mathbb{P}$ of process identifiers and an arbitrary set $\mathbb{V}$ of message payloads. A *message* is a triple $(p, q, v) \in \mathbb{P}\times\mathbb{P}\times\mathbb{V}$ where p represents the source of the message, q its destination, and v the payload. The set of all messages is denoted by $\mathbb{M}$. A *process with identifier* p is a tuple $A = (\Sigma, s_0, Snd, Upd)$ where:

- Σ is a set of process local states, and s_0 is the initial state of the process,

$$\textsc{Send}\quad \frac{\mathcal{P}(p).\mathit{Snd}(\mathit{ls}(p)) = M}{(\mathit{pool}, \mathit{ls}) \xrightarrow{\text{send}(p)} (\mathit{pool} \cup M, \mathit{ls})} \qquad \textsc{Environment}\quad \frac{M \subseteq \mathit{pool}}{(\mathit{pool}, \mathit{ls}) \xrightarrow{\text{env}(M)} (M, \mathit{ls})}$$

$$\textsc{A-Update}\quad \frac{M \subseteq \mathit{pool} \cap (\mathbb{P} \times \{p\} \times \mathbb{V}) \text{ and } \mathcal{P}(p).\mathit{Upd}(\mathit{ls}(p), M) = s}{(\mathit{pool}, \mathit{ls}) \xrightarrow{\text{a-update}(p)} (\mathit{pool} \setminus M, \mathit{ls}[p \mapsto s])}$$

$$\textsc{S-Update}\quad \frac{M = \mathit{pool} \cap (\mathbb{P} \times \{p\} \times \mathbb{V}) \text{ and } \mathcal{P}(p).\mathit{Upd}(\mathit{ls}(p), M) = s}{(\mathit{pool}, \mathit{ls}) \xrightarrow{\text{s-update}(p)} (\mathit{pool} \setminus M, \mathit{ls}[p \mapsto s])}$$

Fig. 4. Transition rules for protocol semantics.

- $\mathit{Snd} : \Sigma \longrightarrow 2^{\mathbb{M}}$ is the message sending function: $\mathit{Snd}(s) = M$ denotes the fact that p sends the set of messages M when in local state s. As expected, we assume that p is the source of all the messages in M.
- $\mathit{Upd} : \Sigma \times 2^{\mathbb{M}} \longrightarrow \Sigma$ is the state-update function: $\mathit{Upd}(s, M)$ is the next state of the process p given its current state s and that it received the set of messages M (we assume that p is the destination of all the messages in M).

Given a process A, we refer to components of A using $A.\Sigma$, $A.s_0$, and so on.

A *protocol* $\mathcal{P}$ maps each process identifier $p \in \mathbb{P}$ to a process $\mathcal{P}(p)$ with identifier p.

Example 3.1. Consider the protocol in Fig. 1. A state is a valuation of the process local variables (declared in the protocol) including a variable representing the control location. The initial state s_0 of any process, has an the empty log of requests, $s_0(\texttt{log\_val}) = \epsilon$, the ballot counter is zero, $s_0(\texttt{ballot}) = 0$, $s_0(\texttt{step}) = \texttt{Prepare}$, and $s_0(\texttt{last}) = 0$.

The functions *Snd* and *Upd* are based on the code snippets that send messages, respectively update the local state (highlighted in the figure with matching labels). For example, for any process p, given a state $s \in \Sigma$ with the program counter at lines 10 (the send of the round `Prepare`),

$$\mathsf{Snd}(s) = \begin{cases} \{(p, q, (\text{“}\texttt{Prepare}\text{”}, s(\texttt{ballot}))) \mid q \in \mathbb{P}\} & \text{if } \texttt{get\_leader()==}\ p, \\ \emptyset & \text{otherwise.} \end{cases}$$

For any process in some state s, if the program counter is at line 14 (the update of the round `Prepare`) then $\mathsf{Upd}(s, M) = s'$ if there is $\mathtt{m} \in M$ s.t. $\mathtt{m.ballot} > s(\texttt{ballot})$ and $\mathsf{Upd}(s, M) = s$ otherwise, where M is the current set of received messages and s' differs from s on the following variables: $s'(\texttt{last}) = s(\texttt{ballot})$, $s'(\texttt{ballot}) = s(\mathtt{m.ballot})$, $s'(\texttt{step}) = \texttt{Ack}$, $s'(\texttt{leader}) = \mathtt{m.sender}$.

A *configuration* of a protocol $\mathcal{P}$ is a tuple $(\mathit{pool}, \mathit{ls})$ where *pool* is a set of messages in transit and *ls* maps each process identifier $p \in \mathbb{P}$ to a process local state in $\mathcal{P}(p).\Sigma$. Given a configuration $c = (\mathit{pool}, \mathit{ls})$ we use $c.\mathit{pool}$ and $c.\mathit{ls}$ to refer to its components.

Asynchronous Semantics. The asynchronous semantics of a protocol $\mathcal{P}$ is defined using a set of transition rules given in Figure 4. The rule Send represents a transition in which a given process p sends all messages prescribed by its message sending function *Snd* in a given state. These messages are added to the pool of messages in transit and the process local states remain unchanged. The rule A-Update represents a transition in which a set of messages M is delivered to a process p and p updates its local state according to its state-update function *Upd*. The set of messages M is chosen non-deterministically from the set *pool* of messages in transit with destination p. This models adversarial networks in which messages can be delayed arbitrarily. The rule Environment is used to model networks that can also drop messages arbitrarily. It defines a set of transitions that can delete an arbitrary set of messages from the pool of messages in transit. These transitions are labeled by send(p), a-update(p), and env(M) where M is the set of messages kept by an Environment transition, respectively.

An *asynchronous execution* of a protocol $\mathcal{P}$ is a sequence of transitions between configurations $c_0 \xrightarrow{\ell_0} c_1 \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_{m-1}} c_m$ where each $\ell_i \in \{\text{send}(p), \text{a-update}(p), \text{env}(M) : p \in \mathbb{P}, M \subseteq \mathbb{M}\}$, for all $0 \leq i \leq m-1$. The set of asynchronous executions of a protocol $\mathcal{P}$ is denoted by $AsyncEx(\mathcal{P})$.

Example 3.2. Fig. 2(c) shows an asynchronous execution of the protocol in Fig. 1. Each square represents a state update transition, each filled circle represents a send transition, and each edge represents a message produced by the sending function. The initial states are given by circles labeled with the initial ballot number. The execution omits send transitions that produce an empty set of messages, e.g. the first and third send actions of process $p2$. The interleaving of transitions performed by different processes is represented by the order between squares and filled circles.

Each solid edge represents a message produced during the send transition where it starts and delivered during the state update transition where it ends. Each dotted edge represents a message dropped by environment transitions. Note that some messages are delayed, i.e., they are delivered during a state update transition that occurs later in the execution and not immediately after the send transition that generated them. For instance, the message ($p2$, (“`Promise`”, 1, a), $p1$) represented by the bold edge arrives with a long delay to process $p1$. The fact that the asynchronous executions can interleave send and update transitions arbitrarily is essential for modeling such delays.

4 COMMUNICATION-CLOSED PROTOCOLS

In this section we define the lossy synchronous semantics exploited by the testing algorithm, and the communication-closure property stating that this semantics is indistinguishable from the standard asynchronous semantics.

Lossy Synchronous Semantics. We consider a lossy synchronous semantics where executions are sequences of *synchronized* rounds in which all processes start by sending the set of messages determined by their local state before updating their local state using a non-deterministically chosen set of messages to receive. These rounds are communication-closed in the sense that the messages which are sent but not received within one round are lost. There is no fixed relation between the messages lost in different rounds.

Formally, a *synchronized round* between two configurations c_0 and $c_{2\cdot n+1}$ with a set of processes $\mathbb{P} = \{p_0, \ldots, p_{n-1}\}$ is a sequence of transitions

$$c_0 \xrightarrow{\text{send}(p_0)} c_1 \ldots \xrightarrow{\text{send}(p_{n-1})} c_n \xrightarrow{\text{env}(M)} c_{n+1} \xrightarrow{\text{s-update}(p_0)} c_{n+2} \ldots \xrightarrow{\text{s-update}(p_{n-1})} c_{2\cdot n+1}$$

where the s-update(·) transitions are defined by the rule S-Update in Figure 4. These transitions represent a variation of the update transitions from the asynchronous semantics where *all* messages which are still in transit are received and used to update the state of a process. A process may still receive a subset of the sent messages because of the env(·) transition scheduled before all update transitions.

We use $c_0 \xrightarrow{\text{round}(M)} c_{2\cdot n+1}$ to denote the sequence of transitions in a synchronized round. A *lossy synchronous* execution is a sequence of synchronized rounds $c_0 \xrightarrow{\text{round}(M_0)} c_1 \ldots \xrightarrow{\text{round}(M_{m-1})} c_m$. The set of lossy synchronous executions of a protocol $\mathcal{P}$ is denoted by $SyncEx(\mathcal{P})$. All synchronous executions we consider are lossy synchronous.

Example 4.1. Fig. 2(a) and Fig. 2(b) show two lossy synchronous executions of the protocol in Fig. 1. The conventions for representing send and update transitions, and messages are the same as in Example 3.2. The transitions that are aligned vertically are ordered from top to bottom.

For the execution in Fig. 2(a), it is assumed that the environment transitions preserve the content of the pool of messages in transit (no messages are dropped). Under the synchronous semantics no messages are delayed and all send and update transitions are executed in lock-step: the k^{th} send (resp., update) is executed simultaneously on all processes. This execution goes through eight rounds, each process iterating twice over the code in Fig. 1.

For the execution in Fig. 2(b), the environment transitions drop the messages represented by dotted edges.

Communication-Closed Protocols. The *behavior* of a process p in a (synchronous or asynchronous) execution $\eta = c_0 \xrightarrow{\ell_0} c_1 \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_{m-1}} c_m$, denoted by $\eta \downarrow p$, is the sequence of states of p in the configurations $c_0, \ldots, c_m$, i.e., $\eta \downarrow p = c_0.ls(p) \ldots c_m.ls(p)$. Two sequences of local states σ and σ' are called *equivalent up to stuttering*, denoted $\sigma \equiv \sigma'$, when they coincide modulo removing consecutive repetitions of the same state. An execution η_1 is *indistinguishable* from another execution η_2, which is denoted by $\eta_1 \equiv \eta_2$, if $\eta_1 \downarrow p \equiv \eta_2 \downarrow p$ for each $p \in \mathbb{P}$.

Example 4.2. The executions in Fig. 2(b) and Fig. 2(c) are indistinguishable. The executions show only (the modification of) the values of the variables `ballot` and `log_val`. The values of the other variables are also equal modulo stuttering. For example, $p1$ goes through the states s_0, s_1, s_2, s_3, s_4 in both executions where s_0 is the initial state, s_1(`ballot`) = 1, s_1(`log_val`) = ϵ, s_1(`step`) = “`Prepare`” s_2(`ballot`) = 1, s_2(`log_val`) = `a` and s_2(`step`) = “`Propose`”. The states s_3 and s_4 differ from s_2 only in the value of the variable `step`, i.e. s_3(`step`) = “`Promise`” and s_4(`step`) = “`Prepare`”.

Definition 4.3. A protocol $\mathcal{P}$ is called *communication-closed* when for each asynchronous execution $\eta_1 \in AsyncEx(\mathcal{P})$ there is a lossy synchronous execution $\eta_2 \in SyncEx(\mathcal{P})$ such that $\eta_1 \equiv \eta_2$.

Communication-closure is a property which is met by all the replicated state machine or consensus protocols we are aware of, e.g., Paxos [Lamport 2005], Multi-Paxos [Chandra et al. 2007], EPaxos [Moraru et al. 2013], ViewStamped [Oki and Liskov 1988]. Intuitively, this property is achieved using the following principles: (1) each process uses a set of variables to encode a local notion of time, called round number, which is monotonically increasing, (2) every message carries some metadata that associates it with some unique round number, and (3) a process updates its state using only messages whose round number equals the process's local round number. Assuming these constraints, any asynchronous execution can be rewritten to an indistinguishable synchronous execution by essentially, reordering commutative transitions [Damian et al. 2019; Elrad and Francez 1982; Moses and Rajsbaum 2002].

For example, the round number of the protocol in Fig. 1 is defined by the values of the pair of variables (`ballot`, `step`). We consider the lexicographic order over the values of (`ballot`, `step`) where the four values of the variable `step` are ordered as “Prepare” < “Ack” < “Propose” < “Promise” (`ballot` is an integer variable and its values are ordered as usual), and define the round number of a process in state s as the position in the lexicographic order of the values of (`ballot`, `step`) in s. Then, every sent message `m` has two fields `m.ballot` and `m.step` that represent its round number (in the same way as the pair of local variables (`ballot`, `step`) represents the process's local round number). The third condition relates message round numbers with process round numbers. Before using the payload of a received message to update the local state, e.g., before reading `m.sender` at line 18 or `m.log_val` at line 27 and storing their values in some local variable, the code ensures that the round number of the message equals the process's local round number, i.e., `m.ballot == ballot` and `m.step == step`. If this is not the case, the message is either not used to update the local state or the round number of the process is first increased to match the message's round number at line 16 before using the message's content to update the state at line 18.

When systems are not known to be communication-closed, one can identify the subset of communication-closed executions. In this case, the lossy synchronous executions represent a subset of the set of executions of the distributed system.

5 UNIFORM EXECUTIONS

In this section, we present a restriction of the lossy synchronous semantics in which the faults (message losses) modeled by the environment transitions are uniform, that is the messages send by a subset of the process are received. This restriction is complete for standard state machine replication and consensus protocols, up to indistiguishability.

A synchronized round $c \xrightarrow{\text{round}(M)} c'$ is called *uniform* if there exists a set of processes Π such that the set of messages received in the round (by some process) is *exactly* the set of messages sent by a process from Π to a process in Π, i.e,

$$\big(\,(p, q, v) \in \mathcal{P}(p).Snd(c.ls(p)) \wedge \{p, q\} \subseteq \Pi\,\big) \Leftrightarrow (p, q, v) \in M,$$

for every p, q, v. The set of processes Π is called the *kernel* of the round. A lossy synchronous execution is called *uniform* when it is a sequence of uniform rounds.

Example 5.1. The synchronous executions in Fig. 2(a), Fig. 2(b) (described also in Example 4.1), and Fig 3 are uniform. In Fig. 2(a), the kernel of each synchronized round is the set of all processes. For the execution in Fig. 2(b), $\Pi_1 = \{p1, p2\}$ is the kernel of the first four synchronized rounds (the first phase), $\Pi_2 = \{p2, p3\}$ is the kernel of the next three synchronized rounds (the first three rounds of the second phase), and $\Pi_3 = \{p3\}$ is the kernel of the last synchronized round.

Figure 5(a) shows a non-uniform execution, where in the last synchronized round, process $p1$ receives messages from $\{p1, p2\}$, the message from $p3$ being lost, and $p2$ receives messages from $\{p2, p3\}$, the message from $p1$ being lost.

A synchronized round $c \xrightarrow{\text{round}(M)} c'$ is *one-to-all* if there exists at most one process p sending messages in this round, i.e., $\mathcal{P}(q).Snd(c.ls(q)) = \emptyset$ for every $q \neq p$, and *all-to-one* if all processes send messages to a single process p, i.e., for every q, if $\mathcal{P}(q).Snd(c.ls(q)) \neq \emptyset$, then there exists $v \in \mathbb{V}$ such that $\mathcal{P}(q).Snd(c.ls(q)) = \{(q, p, v)\}$. A protocol $\mathcal{P}$ is *leader-based* iff all its synchronous executions are sequences of one-to-all or all-to-one synchronized rounds.

Example 5.2. For the protocol in Example 3.1 (Figure 1), a synchronized round where the leader sends a "`Prepare`" message to all processes (the first round a phase) is *one-to-all* while a synchronized round where processes send an "`Ack`" message to the leader is an *all-to-one* round (the second round a phase). Note that *all* refers to the maximum number of processes that can receive, resp., send messages, in a synchronized round (messages can be dropped during an environment transition).

The following theorem implies that the restriction to uniform lossy synchronous executions is complete for leader-based protocols which are also communication closed.

Theorem 5.3. *Every lossy synchronous execution of a leader-based protocol is uniform.*

All benign consensus and replicated state machine implementations [Junqueira et al. 2011; Lakshman and Malik 2010a; Moraru et al. 2013] are leader-based, and hence satisfy Th. 5.3. However, our running example in Fig. 1 is not leader-based since the last round uses an *all-to-all* communication. In the `Promise` round, all processes that received the leader's proposed log (in the previous round)

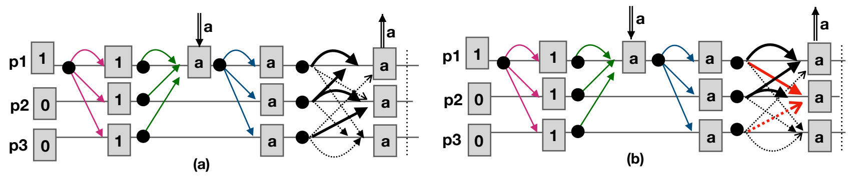

Fig. 5. Two synchronous executions of the protocol in Figure 1.

Table 1. The comparison of sample set sizes of different algorithms.

Set of executions with:	**Upper bound on the size**
Arbitrary message losses	$2^{n^2 r}$
***k*-periodic losses**	$k^{n^2 r/k}$
***d*-bounded *k*-periodic losses**	$\leq C(n^2 r, d) \leq (n^2 r)^d$
Arbitrary uniform executions	2^{nr}
***k*-periodic uniform executions**	$k^{nr/k}$
***d*-bounded *k*-periodic uniform executions**	$\leq C(nr, d) \leq (nr)^d$
Arbitrary reorderings	$\leq (n^2 r)!$
***d*-bounded reorderings (PCT)**	$\leq w \cdot C(nr, (d-1)) \cdot (d-1)!$

broadcast this proposal to all the processes in the network. A process that receives more than $n/2$ messages, having as payload the same log value, transmits this log to the client. The uniform executions with all-to-all communication are a strict subset of the lossy synchronous executions.

The protocol in Fig. 1 confirms, beyond leader-based algorithms, the hypothesis that bugs manifest in uniform executions, as the incorrect execution from Fig. 3 is uniform. The underlying principle is that for any non-uniform execution of the protocol in Fig. 1 either there exists an indistinguishable uniform execution or there exists a uniform execution that exposes the same log values to the client. Fig. 5(b) shows a uniform execution that is indistinguishable from the execution in Fig. 5(a). The equivalence relation between non-uniform and uniform executions (w.r.t. the client's observations) is proved using a key insight from consensus proofs: a process communicates with the client only when the system is in a univalent (global) state, i.e., $|\{p \mid \mathtt{log}(p) = val \wedge \mathtt{last}(p) \geq b\}| > n/2$ for some integer b, which means that val is a stable prefix of the log.

Finally, note that the protocol in Fig. 1 continues to solve state machine replication if we replace the last all-to-all round with an all-to-one round. In the modified `Promise` round processes send a `Promise` message only to the leader (instead of broadcasting it) acknowledging its proposal and only the leader transmits the log to the client, in case it received $n/2$ `Promise` messages.

6 RANDOM SAMPLING FROM UNIFORM EXECUTIONS

We now present our testing algorithm. Theoretically, the effectiveness of the testing algorithm is based on the fact that it samples from a relatively small (yet, complete in the limit) set of executions, rather than from all possible executions of a protocol.

6.1 The Space of Executions

Before giving the sampling procedure, let us consider the size of the space of executions, and compare the space of executions to other techniques. For the comparison, we consider a test harness consisting of n processes running a total of r rounds. In addition to these parameters, we consider two additional parameters to prioritize the search: the *periodicity* k and the number of isolated processes d. Given a lossy synchronous execution τ, we say that a process p *starts at round* i *in* τ if p is included in the kernel of the i-th round in τ but it is not included in the kernel of the previous round (round $i-1$). Then, a uniform execution τ is k*-periodic* if a process can start only at a round which is a multiple of k.

Consider the uniform execution in Figure 3. It has 4 phases and 4 rounds in each phase. The figure omits the last two "empty" rounds in the second and third phase, where no messages are sent. The 4-periodic execution of this example recovers isolated processes after every $k = 4$ rounds, that is in the beginning the second phase with `ballot` 2, the third phase with `ballot` 3, and the fourth phase with `ballot` 4. The k-periodic executions take the empty rounds into account.

k-periodic uniformity. Consider an execution with n processes running a protocol with r rounds. In a non-uniform execution, any subset of the n^2 communication links can have a message loss in each round, resulting in $2^{n^2 r}$ possible executions. In a uniform execution, the corresponding number is 2^{nr}. In a k-periodic non-uniform execution, each of the links can be broken at any k rounds in all r/k phases, resulting in $k^{n^2 r/k}$ executions. In a k-periodic uniform execution, by a similar argument, the number of executions is $k^{nr/k}$. For the example in Figure 3 with 3 processes, 4 rounds and 4 phases, the sample set of executions is around 10^{43} for non-uniform executions and only around 10^7 for 4-periodic uniform executions.

d-bounding. While k-periodic uniformity already reduces the size of the execution space, bounding the set to d-bounded k-periodic uniform executions, i.e., executions with d isolated processes over all rounds, further reduces it. This bound reduces the asymptotic size of the space of executions so that it is exponential only in the bounding parameter d but polynomial in the number of rounds and processes. The bounded version of the non-uniform case has an upper bound of $(n^2 r)^d$. When we further restrict to the uniform case, we get an upper bound of $(nr)^d$. The actual sample set is smaller for $d > n$ since we cannot isolate more than n processes into a period of k rounds.[2]

Table 1 summarizes upper bounds on the number of executions for various choices (arbitrary message losses vs. uniform executions, k-periodic, and d-bounded k-periodic). Additionally, it shows the number of executions explored by a state-of-the-art sampling algorithm (PCT [Kulahcioglu Ozkan et al. 2018]) that is oblivious to rounds.

The size of the set of d-bounded k-periodic uniform executions is asymptotically smaller than the others on Table 1. Moreover, the characterization of the bounding parameter for k-periodic uniform executions requires a smaller value of d to reproduce an execution.

6.2 The Sampling Algorithm

Our testing algorithm (Algorithm 1) takes a test harness consisting of a set $\mathbb{P}$ of n processes running at most r rounds, and randomly samples from the set of k-periodic uniform executions with at most d isolated processes, i.e., from a sample space of size at most $(nr)^d$. The algorithm ensures that each execution is picked with probability at least $1/(nr)^d$.

Given the set of processes $\mathbb{P}$, upper bound on rounds r, and the parameters k and d, the algorithm distributes the d failures into r/k phases (line 1). For each phase, in its first round (line 5), the algorithm selects a set of d_{phase} processes to isolate in the current phase (line 6). For each of the d_{phase} selected processes, the algorithm chooses the first round in which the process is isolated (line 7). The algorithm isolates these processes by simply dropping them from the kernel of the corresponding rounds (line 8). We write $f^{-1}([0,n])$ to denote $\bigcup_{0 \le i \le n} f^{-1}(i)$ and use this to propagate process isolation in a phase until the end of that phase. The algorithm simulates re-establishment of faulty links by resetting the isolated set of processes in every k rounds.

[2] The size of d-bounded k-periodic set of executions can be more precisely characterized by inclusion-exclusion principle [Charalambides 2018] or using q-binomial coefficients [Kac and Cheung 2001].

Input: A test harness with a set $\mathbb{P}$ of n processes and at most r rounds
Input Parameters : A period k and a bound d on the number of isolated processes

1 distribute d into $d_0, \ldots, d_{(r/k-1)}$ s.t. $\sum_{0 \le i < r/k} d_i = d$ and $d_{0 \le i < r/k} \le |\mathbb{P}|$;
2 **for** $i := 0$ *to* $r - 1$ **do**
3 | $phase := i \,/\, k$;
4 | $roundInPhase := i \,\%\, k$;
5 | **if** $roundInPhase = 0$ **then**
6 | | choose u.a.r. d_{phase} processes from $\mathbb{P}$ as $\mathbb{P}_{phase}$;
7 | | choose u.a.r. $f : \mathbb{P}_{phase} \rightharpoonup [0, k-1]$;
8 | schedule round with kernel $\mathbb{P} \setminus f^{-1}([0, roundInPhase])$;
9 check specification on execution trace

Algorithm 1: Randomized sampling from k-periodic uniform executions with bound d.

The algorithm can be modified to sample executions with an unbounded number of isolated processes. For this, we omit the parameter d together with the lines 1 and 6 in the algorithm. On line 7, we isolate any process at any round.

Proposition 6.1 (Soundness and Relative Completeness). *(1) Algorithm 1 samples each synchronous uniform executions of periodicity k and up to d isolated processes with probability at least $1/(nr)^d$. (2) Let $\mathcal{P}$ be a leader-based communication-closed distributed protocol. For any asynchronous execution of $\mathcal{P}$, there is a test harness and parameters d and k such that Algorithm 1 run on the harness with (d, k) samples an indistinguishable execution with positive probability.*

The bugs reported by the testing algorithm are not spurious as the testings enumerates actual executions of the system under test. The applicability does not depend on whether the system under test is indeed communication-closed, that is if all asynchronous executions have a synchronous indistinguishable counter-part. If the system is not communication closed the algorithm will cover an important sub-set of executions.

7 EXPERIMENTAL EVALUATION

We present an empirical evaluation of our approach on production implementations of three fault-tolerant protocols: Cassandra's Paxos [Lakshman and Malik 2010b], Zookeeper's atomic broadcast (ZAB) [Hunt et al. 2010], and the Raft [Ongaro and Ousterhout 2014] implementation in Ratis. This evaluation addresses the following research questions:

RQ1 Is our testing algorithm effective at detecting fault tolerance bugs in large scale systems?

RQ2 How do the algorithm parameters affect the efficacy in detecting bugs?

RQ3 How do different implementations of our algorithm affect the effectiveness at detecting bugs?

To address **RQ1** we show that our framework is indeed able to discover bugs in these implementations, some of them being unknown before our work. We also compare its effectiveness with a baseline approach that explores arbitrary asynchronous executions with arbitrary message losses.

For **RQ2**, we tested each system under varying bounds for the number of isolated processes. For Cassandra, we also evaluated the effect of varying the periodicity of isolation recovery.

For **RQ3**, we experimented with three implementations of Algorithm 1, that provide different approximations of the lossy synchronous semantics. These implementations differ in the instrumentation effort and required information about the internals of the system under test.

Heavy system instrumentation. This is a precise implementation of Algorithm 1 that instruments the system in order to enforce the lossy synchronous semantics and to control the isolation of processes precisely. This requires identifying the messages sent in a certain round and controlling their delivery so that they are delivered only in the context of the same synchronized round they were sent (or dropped). The round of a message is identified by looking at the metadata stored in that message. The presence of such metadata is actually a common design principle for fault-tolerant systems [Fekete and Lynch 1990]. To control the delivery of messages, the instrumentation adds a layer on top of the network which collects the messages in flight, and enforces their delivery to be synchronous. We used this implementation to test Cassandra.

Lightweight system instrumentation. This implementation looks at the metadata stored in the messages to identify those that should be dropped according to Algorithm 1, but it only approximates the lossy synchronous semantics. In this approximation, processes execute a *phase* in lockstep, but they may run the rounds inside the same phase asynchronously. The lockstep execution of phases is enforced using high-enough timeouts, which ensure that each process terminates a phase before advancing in the execution (a phase usually corresponds to handling one client request). We implemented this approach for testing Ratis.

No system instrumentation. A coarse version of Algorithm 1 can be implemented using only the API methods of the system under test (treating the system as a black-box). The tester uses timeouts to enforce a lockstep execution of phases, but does not look inside messages to decide which ones should be dropped. Instead, it uses API methods for stopping or starting a process at the beginning of a phase as an approximation for isolating/deisolating a process during a phase. We used this approach for testing Ratis and Zookeeper.

7.1 Cassandra

Cassandra ensures serializability of transactions using an implementation of Paxos. This protocol is used to make different processes (replicas) agree on an order in which to execute the transactions submitted by the client. Each phase consists of six "one-to-all" or "all-to-one" rounds similar to those in Fig. 1: `Prepare`/`Promise`, `Propose`/`Accept`, and `Commit`/`Ack` (therefore all its lossy synchronous executions are uniform).

We test Cassandra using a harness with three processes and three transactions, two of which update the same key. At the end of the tests, we read the values of the keys and check for the serializability of the processed transactions. This harness admits a difficult to detect buggy behavior in Cassandra 2.0.0 when messages are lost at subtle points of execution [Apache 2013]: one of the processes does not receive the messages sent during the rounds processing the first two transactions, and when this process becomes a leader instead of trying to process a third transaction, it recommits the first one that was already executed, violating serializability.

We tested Cassandra using a precise implementation of Algorithm 1, that controls the messages to be dropped or their delivery (the "heavy system instrumentation" described above). We bounded the length of the executions to at most 24 rounds. [3]

The effect of varying parameters. We evaluate the effect of varying the values of the parameters d and k when testing with the harness described above. For each assignment of parameters, we sampled 1000 executions. For each set of tests, we report in Table 2 the average number of rounds and phases that are executed by a quorum of processes[4] (as **#rnds**✓ and **#phs**✓), in addition to the

[3]Source code at https://github.com/burcuku/explorer-server

[4]The parameters d and k affect the distribution of the isolated processes in an execution, which in turn may affect the length of an execution. The processing of the three transactions can finish in 18 rounds if no messages are lost, or more rounds when processes are isolated and quorums cannot be formed.

Table 2. The number of buggy executions detected by sampling from d-bounded k-uniform executions. On the left, we list the results for $d = 8$ and varying k. On the right, we list them for $k = 6$ and varying d.

k-uniform	#rnds	#rnds✓	#phs	#phs✓	#msgs	#buggy
$k = 1$	21.67	18.14	3.61	2.87	49.13	0
$k = 2$	21.60	18.07	3.60	2.87	48.87	0
$k = 4$	22.80	17.53	3.80	2.64	46.76	0
$k = 6$	22.86	17.10	3.81	2.63	44.78	2
$k = 8$	23.71	6.61	3.95	1.03	20.23	0
$k = 10$	23.81	6.36	3.97	0.94	19.60	0

d-bounded	#rnds	#rnds✓	#phs	#phs✓	#msgs	#buggy
$d = 3$	19.08	18.08	3.18	3.00	48.47	0
$d = 4$	20.11	18.29	3.35	2.99	48.62	0
$d = 5$	20.91	18.17	3.48	2.93	47.90	1
$d = 6$	21.69	17.98	3.61	2.86	47.13	1
$d = 8$	22.86	17.10	3.81	2.63	44.78	2
$d = 10$	23.61	15.72	3.93	2.31	41.83	1

average number of rounds (**#rnds**), phases (**#phs**), messages (**#msgs**), and the number of times a buggy execution is sampled (**#buggy**). We mark a round to have a quorum if the kernel of that round consists of a majority of processes. Similarly, we mark a phase to have a quorum if the corresponding user request takes effect (i.e., a written value is committed) on a majority of processes.

The left of Table 2 lists the results when varying $k = \{1, 2, 3, 4, 6, 8\}$ and fixing $d = 8$ (this value of d is high-enough for reproducing the bug). For values of k smaller than the number of rounds in a phase, executions have a higher number of rounds and phases with a quorum. This can be explained by the fact that the isolated processes get a chance to recover from message losses during the execution of the phase. As k increases, fewer rounds have a quorum, resulting in an increase in the total number of rounds. When $k > 6$, links are not re-established at the beginning of a phase and faults propagate to succeeding phases. This causes the protocol to fail to process user requests in later phases. Only about a single phase is successful for $k = 8, 10$ on average.

The data on the right of Table 2 shows that as d increases, the average number of rounds and phases executed by a quorum of processes decreases due to a higher frequency of message losses. Consistently, the average of the total number of rounds and phases in an execution increases due to the repetition of no-quorum phases. In the extreme case with an unbounded number of isolated processes, a minority of rounds are executed by a quorum, failing to process even a single request on average. Tests with a bounded number of isolated processes produce executions with both quorum and no-quorum phases which are more likely produce a buggy behavior. In our experiments, we could reproduce the bug by taking $d \in \{5, 6, 8, 10\}$.

Testing Cassandra with a baseline algorithm. As a baseline for testing fault tolerance of a system against network failures, we consider a naive randomized algorithm. This algorithm samples from the set of executions with arbitrary message losses, by randomly dropping a message with some probability. We tested Cassandra 1000 times using different probabilities $p = 0.125, 0.25, 0.5$. In our evaluation, none of those tests could hit the bug in the system. The infrequency of hitting the bug is not surprising since the bug in Cassandra is known to be a difficult bug and it is reproduced only in few executions in previous works [Kulahcioglu Ozkan et al. 2019; Leesatapornwongsa et al. 2014b].

7.2 Ratis

Ratis [Apache 2020] is an implementation of the Raft protocol [Ongaro and Ousterhout 2014], usable in large-scale systems such as Hadoop Ozone key-value store. Ratis is in early stages of development, currently in version 0.6.0. Raft is a consensus protocol for state machine replication. Similarly to Paxos and our motivating example, operations on the state machine are sent to the leader of the Ratis cluster. The leader appends operations to its log and replicates the operations to other servers. An operation is committed once the leader receives acknowledgements from a majority of servers. Differently from Paxos, a server can become leader only if its log is at least as up-to-date with the other servers. Raft consists of *leader election* or *log replication* rounds. The servers exchange `RequestVote`/`RequestVoteReply` messages for leader election, and `AppendEntries/`

`AppendEntriesReply` messages for log replication and as heartbeat messages. Similarly to other consensus protocols, Raft uses only "one-to-all" and "all-to-one" rounds.

We tested Ratis using an implementation of our algorithm based on lightweight instrumentation.[5] A test harness consists of a number of client requests submitted to the Ratis cluster and the maximal number of rounds in an execution, approximated using a timeout. When processed, each request extends the replicated log with some message. During the processing of the requests, we introduced message losses as prescribed by our algorithm. At the end of a test, we ran the system without failures for some time to allow the cluster to recover and synchronize its servers. Finally, we checked whether the system could tolerate the introduced message losses by checking the following properties extracted from [Ongaro and Ousterhout 2014] and the unit tests in Ratis:

P1 The servers eventually elect a leader.
P2 All servers eventually store all log entries.
P3 After sending a request, a client eventually receives a reply.

While these specifications are liveness properties, we checked for bounded-liveness variations where they are required to be satisfied within a bounded amount of time. To define the time bounds we use a heuristic similar to [Killian et al. 2007]. We run the system without any message loss (failures) several times to determine the average time required to synchronize the servers. In our tests, we allowed the system to run significantly longer to recover after the message losses.

We tested Ratis using $n = 3$ servers, 4 client requests, and a varying number of failures (isolated processes) distributed into $r = 8$ rounds. The number of rounds is counted based on the size of the replicated log (which is observed by the instrumentation). We used a period $k = 2$ to recover isolated processes. At the end of the 8 rounds, we continue running the system without any failures leaving a timeout of 2 seconds to allow the servers synchronize. Ratis has significant amount of support code for the transport layer libraries it uses, namely gRPC and Netty. This can lead to different system behavior when run with different transport options. To cover both behaviors, we tested Ratis using both gRPC and Netty libraries.

Testing Ratis using the lightweight system instrumentation. We tested an instrumented version of Ratis which enables our algorithm to read the content of in flight messages and be able to drop them. The algorithm uses the information in the messages (more specifically, the size of the sender's log) to identify the current round of a server. Then, we isolate selected servers in selected rounds by dropping the messages of those rounds from/to the isolated servers.

We tested Ratis 1000 times using different values for the bound on the number of isolated processes $d = 1, .., 7$. In Table 3, we list the number of violations to the specifications P1, P2 and P3 detected in our tests for each value of d. In many test executions with gRPC, we observed violations to P2 or P3. In the failing tests, a follower server has inconsistent entries with the leader, and sends a negative reply to leader's `AppendEntries` message. Inconsistency in the servers logs can arise when the leader cannot fully replicate all of the entries in its log, e.g., when it disconnects before sending `AppendEntries` messages. In the problematic executions, the leader and the follower with inconsistent entries repeatedly send the same messages to each other and fail to synchronize in hundreds of exchanged messages. Our bug report for this problem is currently open.[6] In our failing tests with Netty, we discovered a liveness bug which causes the violation of P3. In the buggy execution, the leader gets disconnected from the cluster after it receives a client request. Then, the cluster elects a new leader. While the client is successfully redirected to the new leader in the implementation for the gRPC adapter, the implementation for Netty causes the client to indefinitely wait for a reply from the old leader. Our bug report for this problem is already acknowledged by the

[5] Source code is available at https://github.com/burcuku/explorer-server.
[6] https://issues.apache.org/jira/projects/RATIS/issues/RATIS-946

Table 3. The number of violations to properties P1, P2 and P3 in Ratis detected by our algorithm using lightweight system instrumentation.

	d	1	2	3	4	5	6	7
Ratis with gRPC	P1	0	0	0	0	0	0	0
	P2	121	199	242	192	103	65	61
	P3	0	0	2	5	22	64	111
Ratis with Netty	P1	17	291	418	576	917	986	995
	P1	362	592	710	778	958	989	995
	P2	151	285	331	472	888	984	992

Table 4. The number of violations to properties P1, P2 and P3 in Ratis detected by our algorithm *without* system instrumentation. On the left, we list the results for the implementation using server blocking methods in Ratis test API. On the right, we list them for the implementation using server kill/restart methods.

	d	1	2	3	4	5	6	7
Ratis with gRPC	P1	0	0	0	0	0	0	0
	P2	0	0	0	1	0	0	0
	P3	0	1	16	88	182	366	523
Ratis with Netty	P1	0	0	2	0	0	0	0
	P2	0	0	1	1	0	2	9
	P3	0	9	69	159	262	497	620

	d	1	2	3	4	5	6	7
Ratis with gRPC	P1	0	0	0	0	0	0	0
	P2	0	0	0	12	23	47	57
	P3	0	18	110	197	205	276	319
Ratis with Netty	P1	0	0	1	3	1	3	7
	P2	0	0	1	0	0	0	0
	P3	0	16	11	96	93	118	79

Table 5. The number of violations detected in Ratis by using a baseline randomized testing algorithm which drops messages with a given probability. We rely on our instrumentation for selectively dropping messages.

p: probability of dropping a message		0.125	0.25	0.50
Ratis with gRPC	P1	1	2	6
	P2	0	1	25
	P3	0	2	155

p: probability of dropping a message		0.125	0.25	0.50
Ratis with Netty	P1	994	971	497
	P2	998	983	462
	P3	999	996	179

Ratis developers.[7] We also observed high number of tests where the servers cannot elect a leader (failing P1) when some messages are dropped. This violation occurs frequently and it is produced by dropping almost any message in the log synchronization of the servers. Our bug report for this violation is also currently open.[8]

Testing Ratis without additional instrumentation. We also implemented two coarser versions of our algorithm where we only use the methods provided by Ratis test API. In one of the implementations, we isolated the servers by using Ratis test API's server isolation methods which block outgoing/incoming messages from/to servers. In the other one, we used server kill and restart methods to isolate servers for some duration. In our implementations, we distributed d number of process isolations into a number of phases which are approximately determined by some timeouts. At the beginning of each phase, we isolated a randomly sampled subset of processes. If the phase has a majority of processes alive, we wait until the system elects a leader (the Ratis API provides a method for checking the leader of a cluster) and submitted 3 client requests. After that, we isolated some other randomly sampled processes and we wait for 2 seconds for the servers to process the requests. At the end of the phase, we recover the isolated processes for the next phase. We ran the system 1000 times for each value of $d = 1, .., 7$.

On the right of Table 4, we list the number of violations to P1, P2 and P3 detected by testing the system using the Ratis API blocking methods. Some tests detects violations of P3, where the executions fail to serve some client requests within timeout. However, the frequency of executions

[7] https://issues.apache.org/jira/projects/RATIS/issues/RATIS-844
[8] https://issues.apache.org/jira/projects/RATIS/issues/RATIS-1048

violating P1 or P2 is very low. A reason for these tests to miss violations might be the behavior of process isolation methods in the Ratis test API. Instead of dropping messages, the isolation methods block messages of a process by sleeping the thread delivering the message until the server is deisolated. This might result in servers to process blocked messages once they are deisolated. In our instrumentation, messages from/to the isolated process are dropped completely.

On the left of Table 4, we list the number of violations by testing the system using the Ratis server kill/restart methods. In these tests we can observe violations to all P1, P2 and P3, in smaller numbers than the tests with instrumentation. A reason for that might be blocking processes for some duration is coarse grained and less selective on which particular messages will be dropped.

Testing Ratis with a baseline algorithm. Table 5 lists the number of violations detected by a naive random algorithm, which samples from the set of executions with arbitrary message losses. We rely on our instrumentation for dropping messages. The algorithm takes a probability value p as input and drops a message with the probability p. For each different value of the probability, $p = 0.125, 0.25, 0.5$ we tested the system with 1000 executions. In Netty, the tests produce executions which violate P1 and therefore P2 due to lack of synchronization in the absence of the leader. However, only a few of the tests could hit an execution with inconsistent servers using gRPC adapter.

In conclusion, in the context of Ratis, the implementation of Algorithm 1 based on a lightweight system instrumentation is quite effective and it hits a higher number of problematic executions in comparison to the coarse-grain implementation (based solely on the Ratis API without any instrumentation) or testing with a baseline randomized algorithm.

7.3 Zookeeper

We tested Apache Zookeeper, a strongly consistent distributed key-value store that relies on the ZAB (Zookeeper Atomic Broadcast) protocol, using a coarse-grained implementation of our sampling algorithm based exclusively on the API of the system, without additional instrumentation.[9]

Our implementation enforces lockstep execution of *abstract phases*, which subsume a sequence of phases at the algorithmic level, starting from an event that causes the servers to start exchanging messages to a steady state. The length of an abstract phase is approximated in two ways. First, after starting a set of servers, a steady state is reached once the client-facing handlers detect that the servers have been started. During this time, the servers will have executed part of the ZAB protocol to agree on the most recent log of client requests. Second, after a client request, reaching a steady state is approximated with a 100ms timeout, empirically sufficient for the servers to commit the request. We use the system API to approximate points in execution where the system reaches a steady state and to inject faults (isolate servers) only at these points. This relaxed approach loses completeness, but it is easier to deploy since it does not require instrumentation. As we demonstrate in this section, it is sufficient for exposing interesting behaviors and bugs in Zookeeper.

Our tool programmatically starts Zookeeper servers as threads, making them easier to manipulate than if they were separate processes. Each server is paired with a client-facing handler, which is also part of the Zookeeper API. The handler is used to detect a change in the server's state (is it up or down), and to initiate a client request (get or set a key-value pair).

A test is parameterized by the number of servers n, a fault budget d, and a *test harness*. The test harness is determined by the client requests and the number of abstract phases, which are organized as a sequence of *steps*. A step can be either an empty step or a request step. An empty step, denoted as empty, consists of a single abstract phase that involves starting a set of servers and waiting for them to reach steady state. A request step consists of two abstract phases: the first

[9] Source code is available at https://github.com/fniksic/zootester.

one is like in the empty step, and the second one involves initiating a client request and waiting for steady state, this time approximated with a 100ms timeout. We support two kinds of client requests: a write request and a conditional write request. A write request for setting key k to value v on server s is written as $s : k \leftarrow v$, and a conditional write request for setting key k_2 to value v_2 on server s, provided that key k_1 is set to v_1, is written as $s : k_1 = v_1 \; ? \; k_2 \leftarrow v_2$. In our tests we use integer values. We identify requests and request steps and use the same notation for both.

A test with n servers, a fault budget d, and a test harness with p steps is executed in the following way. First there is an initial step in which all keys appearing in the harness are set to zero. Then we use a version of Algorithm 1 to sample a random execution of the harness with d faults: we distribute d faults over p steps, and additionally, if a step is a request step consisting of two abstract phases, we randomly assign some of the faults to the second abstract phase in the step. At the beginning of a step, we randomly choose a kernel of servers to start according to the number of faults assigned to the first abstract phase in the step. If there is a second abstract phase, we randomly choose servers to stop, again according to the number of faults assigned to the abstract phase. At the end of a step, we stop all servers and proceed to the next step. Finally, once all steps are executed, we start all servers and check that they are in the same final state, and that the final state is allowed under some *sequentially consistent* execution of the requests.

In our first experiment, we focus on exposing bug ZK-2832[10], reported to occur in Zookeeper 3.4.9. The bug causes the servers to diverge; thus, we will refer to the bug as the *divergence* bug. The reporter of the bug provided a test with the exact steps to deterministically reproduce the bug. The steps involve three servers handling two client requests in presence of four faults. The client requests set new values to two different keys. At the end the servers diverge: two servers disagree on the value associated with one of the keys.

Interestingly, the deterministic test provided by the bug's reporter fails to reproduce the bug in releases of Zookeeper more recent than 3.4.9. Even though the bug report was still open at the time of writing, it may seem that the bug has disappeared. Unfortunately, this is not the case: we were able to reproduce the bug in Zookeeper 3.5.8, released in May 2020.

Using our tool, we can represent the steps from the deterministic test as the following harness involving servers s_0, s_1, s_2 and keys k_0, k_1: $H_{\mathsf{div}} = [s_1 : k_0 \leftarrow 101; \mathsf{empty}; s_2 : k_1 \leftarrow 302]$. The exact values assigned to the keys in the harness are not important, as long as they are distinct.

We ran the harness with different values of the fault budget d. For each d from 0 to 9 we ran 1,000 executions and observed divergence in 0 to 5 executions per test. As a comparison, we ran a baseline test in which we execute harness steps in 5-second intervals, while at the same time we crash and restart servers in intervals randomly distributed according to Poisson distribution with the mean of 2 seconds. In the baseline test, we observe divergence in 2 out of 1,000 executions. In addition to the divergence bug, one of the executions of the baseline test shows what seems to be a new issue: at the end, one of the clients is unable to connect to any of the servers. We believe this cannot be correct behavior. We refer to this issue as *client dropped*. The left of Table 6 summarizes the results. The last row in the table shows executions that were unsuccessful: occasionally a client fails to read a value from a server. These executions are more likely to be a result of our tool not being perfectly robust than of an actual issue with Zookeeper.

In our next experiment, we experimented with our tool in the context of a random enumeration of harnesses. To restrict the space of harnesses, we fixed the number of servers to 3, and the number of keys to 2. In one experiment, we additionally fixed the number of requests $req = 2$, the total number of steps $p = 3$, and the fault budget $d = 4$. In another experiment, we fixed the additional

[10]https://issues.apache.org/jira/browse/ZOOKEEPER-2832

Table 6. Testing Zookeeper. On the left, we list the number of Zookeeper executions with harness H_{div} exhibiting bugs listed in the first column, for varying d (we ran 1,000 executions for each value of d and for the baseline test). On the right, the number of Zookeeper executions exhibiting bugs listed in the first column for randomly sampled harnesses. For each of the two choices of parameters we randomly sampled 12 harnesses and ran 1,000 executions per harness.

d	0	1	2	3	4	5	6	7	8	9	baseline
divergence	0	0	0	2	2	0	5	4	3	0	2
client dropped	0	0	0	0	0	0	0	0	0	0	1
unsuccessful	0	0	0	1	0	1	0	0	0	1	8

	$req = 2, p = 3$ $d = 4$	$req = 4, p = 5$ $d = 6$
divergence	15	13
failure of SC	0	1
client dropped	0	7
unsuccessful	4	8

Table 7. Number of Zookeeper executions with harness H_{sc} exhibiting bugs listed in the first column, for varying d. We ran 1,000 executions for each value of d and for the baseline test.

d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	baseline
divergence	0	0	0	2	1	2	2	5	6	0	0	0	0	0	0	0	7
client dropped	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	2
unsuccessful	0	0	0	0	0	0	0	0	0	0	1	2	0	0	0	0	19

parameters as $req = 4, p = 5, d = 6$. For each choice of parameters, we sampled 12 harnesses and ran 1,000 executions per harness.

The highlight of our findings is that, in addition to observing more divergence and dropped clients, we observe a new issue: in 1 out of 12,000 executions with $req = 4, p = 5, d = 6$, the servers converge to the same state, but this state is not allowed under sequential consistency. We refer to the issue as *failure of sequential consistency*. We have created a test that deterministically reproduces the violating execution and reported the issue as ZK-3875.[11] The issue occurs in Zookeeper 3.5.8, but not in the more recent branch 3.6 of stable releases. At the time of writing it was still unclear which change in the 3.6 branch seems to resolve the issue. The results are summarized on the right of Table 6.

In our final experiment, we isolated the harness that yielded the execution exhibiting the failure of sequential consistency:

$$H_{\text{sc}} = [s_1 : k_1 = 0 \,?\, k_1 \longleftarrow 101; \text{empty}; s_0 : k_1 = 101 \,?\, k_0 \longleftarrow 200;$$
$$s_1 : k_1 = 0 \,?\, k_1 \longleftarrow 301; s_0 : k_1 = 0 \,?\, k_0 \longleftarrow 400]$$

In the incorrect execution, the final state on all servers is $\{k_0 = 200, k_1 = 301\}$. In the experiment, we wanted to see if we can detect failure of sequential consistency again, either by our sampling algorithm or by the baseline test. Therefore, we fixed the harness to H_{sc} and varied the fault budget d from 0 to 15. We observe divergence in 0 to 6 executions for our sampling algorithm, and in 7 executions for the baseline test. We observe clients dropped in 2 executions, both in our sampling algorithm and the baseline test. However, were not able to catch the failure of sequential consistency again, which shows that it is a rare bug. Table 7 summarizes the results.

7.4 Summary of Evaluation

Our experimental evaluation shows that our algorithm can detect new bugs in large scale systems as well as reproduce known bugs. In our tests, small values of d and values of k allowing a client request to be processed between recovery points could successfully detect bugs. This confirms our hypothesis that uniform executions with a small number of isolations are sufficient to find many bugs. We discovered new bugs in the recent versions of Zookeeper and Ratis. We inspected the

[11] https://issues.apache.org/jira/browse/ZOOKEEPER-3875

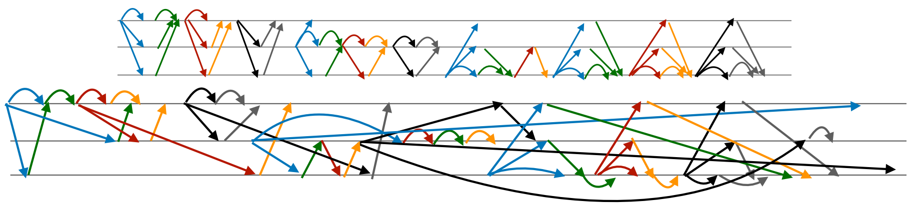

Fig. 6. A synchronous buggy trace sampled by our algorithm and a buggy trace sampled by PCT, both for Cassandra's Paxos bug.

buggy executions and have already reported some of them in the projects' issue tracker sites; some bugs in Ratis have already been fixed in the master branch of the project.

A limitation of some testing tools for distributed systems is the instrumentation burden. Our experimentation with different levels of precision on the identification of rounds and phases shows that sampling from uniform executions provides an effective approach for testing fault tolerance in general, even with coarse-grained instrumentation. All three versions of the implementation of our algorithm outperform a baseline random testing algorithm and can expose bugs in large scale systems.

Debuggability. We conclude by demonstrating that buggy executions detected by our algorithm can be easier to understand than the traces obtained by exploring all (asynchronous) executions. While the interpretability of the generated traces depends on the precision of the analysis of rounds in implementation, in general our algorithm produces execution traces that omit messages in a more structured way than reordering or dropping messages arbitrarily. Fig. 6 shows two buggy executions from Cassandra found by our algorithm and PCT [Kulahcioglu Ozkan et al. 2018], respectively. Our algorithm returns a synchronous execution trace which lists messages in the expected protocol order, making it more explicit which processes are isolated in each round. On the other hand, the programmer needs to follow the complicated message interleavings across phases to discover the delayed/dropped messages in the asynchronous trace.

8 RELATED WORK AND CONCLUSION

We have proposed a new testing methodology based on *communication-closure* as the starting point. Communication closure offers an elegant abstraction at the level of algorithm design, and our testing methodology uses the abstraction as a way to focus on a much smaller sample space of executions. For many common classes of distributed algorithms, the reduction remains complete. We have shown that exploring *uniform* executions with a small number of faults is sufficient to find bugs in production distributed systems like Cassandra, Zookeeper, or Ratis.

Using algorithmic insights into testing distributed systems to reduce the space of executions is a major point of departure from existing work in randomized or systematic testing of implementations of distributed systems. At the same time, our insight is orthogonal to the many reduction techniques already exploited in existing tools, such as depth bounding [Kulahcioglu Ozkan et al. 2018], partial order reduction [Kulahcioglu Ozkan et al. 2019; Yuan et al. 2018], or semantics-aware analyses [Leesatapornwongsa et al. 2014b; Lukman et al. 2019b].

Several execution prioritization techniques are designed for efficient analysis of concurrent software [Thomson et al. 2014]. Context bounding [Qadeer and Rehof 2005] or preemption-bounding [Musuvathi and Qadeer 2007] are designed for shared memory programs, defining a prioritization scheme based on multithreading concepts. While delay bounding [Emmi et al. 2011] or probabilistic prioritization in PCT [Burckhardt et al. 2010] are applicable to message passing systems, they consider the state space of message reorderings, hence parameterize the set of asynchronous executions. In this work, we provide an approach for exploring the set of synchronous executions of a distributed system. Note that we are not aware of any notion similar to communication closure that applies to shared-memory programs.

While we address fault tolerance bugs due to message losses in this work, a related source of bugs is erroneous crash recovery of servers [Gao et al. 2018; Gunawi et al. 2015; Lu et al. 2019]. Erroneous recovery causes the servers not to restart properly and leads to bugs in the system. Since message losses in network and server crashes are orthogonal sources of faults, producing executions with both kinds of faults may be promising for more extensive testing.

Our work is inspired by the quest for an easier to understand subset of representative asynchronous executions, and simpler proofs of algorithms, which led to the communication closure property. Communication-closed layered systems [Charron-Bost and Schiper 2009; Chou and Gafni 1988; Gafni 1998; Moses and Rajsbaum 2002; Santoro and Widmayer 1989] capture both lossy synchronous and lossy asynchronous behaviors and solve consensus under the partial synchrony network assumption [Dwork et al. 1988]. They rely on easier to interpret synchronous lock-step executions and simpler proof arguments. For example an equivalence relation between asynchronous and communication closed executions is established for systems that solve consensus in [Chaouch-Saad et al. 2009; Elrad and Francez 1982; Moses and Rajsbaum 2002].

Motivated by the impossibility of solving consensus over asynchronous faulty networks [?] synchronous abstractions offer an alternative view of distributed systems. They have beed studied to simplify programming distributed, concurrent, and parallel systems, e.g., virtual synchrony [?], bulk programming [?], for designing theoretical solutions for consensus [Dwork et al. 1988], and to simplify reasoning about a system's traces [Elrad and Francez 1982]. Implementations of consensus protocols have been proposed for these synchronous programming paradigms, e.g., virtual synchrony [?] or PSync [Dragoi et al. 2016] (a programming paradigm based on communication-closure). However, in production asynchronous state machine replication systems are still to be understood if they have an implementation in synchronous programming models. In contrast, using communication-closure in testing increases the confidence we have in production systems without having to reimplement them. In [Damian et al. 2019] communication-closure is defined based on conditions on the sequential code independently of the specification of the systems and it is applied to semi-automatically prove correct several consensus protocols. The complexity and scale of the verified code is far from production system. No previous work studies the relation between communication closure and testing distributed systems.

Finally, recent developments in verifying replicated state machine and consensus protocols [Chaudhuri et al. 2010; Hawblitzel et al. 2015; Padon et al. 2017; von Gleissenthall et al. 2019; Wilcox et al. 2015] allow fully verified implementations to be developed. However, these verified implementations lack the performance of production systems, are small scale implementations that have prototype clients and minimal deployment. Formalization is important, however bugs may still arise [Fonseca et al. 2017; Sutra 2019].

ACKNOWLEDGMENTS

Kulahcioglu Ozkan and Majumdar were supported in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248 and by the European Research Council under the Grant Agreement 610150 (ERC Synergy Grant ImPACT). This work was done mainly when Cezara Drăgoi was affiliated with INRIA supported by the French National Research Agency ANR project SAFTA (12744-ANR-17-CE25-0008-01).

REFERENCES

Apache. 2013. *CASSANDRA-6023: CAS should distinguish promised and accepted ballots.* Retrieved January 26, 2020 from http://issues.apache.org/jira/browse/CASSANDRA-6023

Apache. 2020. *Apache Ratis.* Retrieved May 14, 2020 from http://ratis.incubator.apache.org/

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, James C. Hoe and Vikram S. Adve (Eds.). ACM, 167–178. https://doi.org/10.1145/1736020.1736040

Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, Indranil Gupta and Roger Wattenhofer (Eds.). ACM, 398–407. https://doi.org/10.1145/1281100.1281103

Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. 2009. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In *Reachability Problems, 3rd International Workshop, RP 2009, Palaiseau, France, September 23-25, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5797)*, Olivier Bournez and Igor Potapov (Eds.). Springer, 93–106. https://doi.org/10.1007/978-3-642-04420-5_10

Charalambos A Charalambides. 2018. *Enumerative combinatorics.* Chapman and Hall/CRC.

Bernadette Charron-Bost and André Schiper. 2009. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Comput.* 22, 1 (2009), 49–71. https://doi.org/10.1007/s00446-009-0084-6

Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2010. Verifying Safety Properties with the TLA+ Proof System. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6173)*, Jürgen Giesl and Reiner Hähnle (Eds.). Springer, 142–148. https://doi.org/10.1007/978-3-642-14203-1_12

Ching-Tsun Chou and Eli Gafni. 1988. Understanding and Verifying Distributed Algorithms Using Stratified Decomposition. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*, Danny Dolev (Ed.). ACM, 44–65. https://doi.org/10.1145/62546.62556

Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. 2019. Communication-Closed Asynchronous Protocols. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 344–363. https://doi.org/10.1007/978-3-030-25543-5_20

Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. 2015. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 73–83. https://doi.org/10.1145/2786805.2786861

Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 400–415. https://doi.org/10.1145/2837614.2837650

Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323. https://doi.org/10.1145/42282.42283

Tzilla Elrad and Nissim Francez. 1982. Decomposition of Distributed Programs into Communication-Closed Layers. *Sci. Comput. Program.* 2, 3 (1982), 155–173. https://doi.org/10.1016/0167-6423(83)90013-8

Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 411–422. https://doi.org/10.1145/1926385.1926432

Alan Fekete and Nancy A. Lynch. 1990. The Need for Headers: An Impossibility Result for Communication over Unreliable Channels. In *CONCUR '90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 27-30, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 458)*, Jos C. M. Baeten and Jan Willem Klop (Eds.). Springer, 199–215. https://doi.org/10.1007/BFb0039061

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017.* ACM, 328–343. https://doi.org/10.1145/3064176.3064183

Eli Gafni. 1998. Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony (Extended Abstract). In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, Brian A. Coan and Yehuda Afek (Eds.). ACM, 143–152. https://doi.org/10.1145/277697.277724

Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 ACM Joint Meeting* *on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 539–550. https://doi.org/10.1145/3236024.3236030

Haryadi S. Gunawi, Thanh Do, Agung Laksono, Mingzhe Hao, Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, and Riza O. Suminto. 2015. What Bugs Live in the Cloud?: A Study of Issues in Scalable Distributed Systems. *login Usenix Mag.* 40, 4 (2015). https://www.usenix.org/publications/login/aug15/gunawi

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 1–17. https://doi.org/10.1145/2815400.2815428

Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010.*

Yury Izrailevsky and Ariel Tseitlin. 2011. The Netflix Simian army. *The Netflix Tech Blog* (2011).

Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*. IEEE Compute Society, 245–256. https://doi.org/10.1109/DSN.2011.5958223

Victor Kac and Pokman Cheung. 2001. *Quantum calculus.* Springer Science & Business Media.

Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code (Awarded Best Paper). In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*, Hari Balakrishnan and Peter Druschel (Eds.). USENIX. http://www.usenix.org/events/nsdi07/tech/killian.html

Kyle Kingsbury. 2013–2018. *Jepsen.* Retrieved January 26, 2020 from http://jepsen.io/

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *PACMPL* 2, OOPSLA (2018), 160:1–160:28.

Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. 2019. Trace aware random testing for distributed systems. *PACMPL* 3, OOPSLA (2019), 180:1–180:29.

Avinash Lakshman and Prashant Malik. 2010a. Cassandra: a decentralized structured storage system. *Operating Systems Review* 44, 2 (2010), 35–40. https://doi.org/10.1145/1773912.1773922

Avinash Lakshman and Prashant Malik. 2010b. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

Leslie Lamport. 2005. *Generalized Consensus and Paxos.* Technical Report MSR-TR-2005-33. 60 pages. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014a. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 399–414. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014b. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* 399–414.

Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019.* 114–130. https://doi.org/10.1145/3341301.3359645

Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019a. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 20:1–20:16. https://doi.org/10.1145/3302424.3303986

Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019b. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019.* 20:1–20:16.

Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 358–372. https://doi.org/10.1145/2517349.2517350

Yoram Moses and Sergio Rajsbaum. 2002. A Layered Analysis of Consensus. *SIAM J. Comput.* 31, 4 (2002), 989–1021. https://doi.org/10.1137/S0097539799364006

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 446–455. https://doi.org/10.1145/1250734.1250785

Brian M. Oki and Barbara Liskov. 1988. Viewstamped Replication: A General Primary Copy. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*, Danny Dolev (Ed.). ACM, 8–17. https://doi.org/10.1145/62546.62549

Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 160:1–160:28. https://doi.org/10.1145/3276530

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 108:1–108:31. https://doi.org/10.1145/3140568

Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-540-31980-1_7

Nicola Santoro and Peter Widmayer. 1989. Time is Not a Healer. In *STACS 89, 6th Annual Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 16-18, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 349)*, Burkhard Monien and Robert Cori (Eds.). Springer, 304–313. https://doi.org/10.1007/BFb0028994

Pierre Sutra. 2019. On the correctness of Egalitarian Paxos. *CoRR* abs/1906.10917 (2019). arXiv:1906.10917 http://arxiv.org/abs/1906.10917

Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency testing using schedule bounding: an empirical study. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, José E. Moreira and James R. Larus (Eds.). ACM, 15–28. https://doi.org/10.1145/2555243.2555260

Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 59:1–59:30. https://doi.org/10.1145/3290372

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 357–368. https://doi.org/10.1145/2737924.2737958

Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial Order Aware Concurrency Sampling. In *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.* 317–335.