CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems
EPFL Technical Report LARA-REPORT-2008-006

Maysam Yabandeh, Nikola Knežević, Dejan Kostić and Viktor Kuncak
School of Computer and Communication Sciences, EPFL, Switzerland
email: `firstname.lastname@epfl.ch`

Abstract

We propose a new approach for developing and deploying distributed systems, in which nodes predict distributed consequences of their actions, and use this information to detect and avoid errors. Each node continuously runs a state exploration algorithm on a recent consistent snapshot of its neighborhood and predicts possible future violations of specified safety properties. We describe a new state exploration algorithm, consequence prediction, which explores causally related chains of events that lead to property violation.

This paper describes the design and implementation of this approach, termed CrystalBall. We evaluate CrystalBall on RandTree, BulletPrime, Paxos, and Chord distributed system implementations. We identified new bugs in mature Mace implementations of three systems. Furthermore, we show that if the bug is not corrected during system development, CrystalBall is effective in steering the execution away from inconsistent states at runtime.

1 Introduction

Distributed systems form the foundation of our society's infrastructure. Complex distributed protocols and algorithms are used in enterprise storage systems, distributed databases, large-scale planetary systems, and sensor networks. Errors in these protocols translate to denial of service to some clients, potential loss of data, and monetary losses. The Internet itself is a large-scale distributed system, and there are recent proposals [19] to improve its routing reliability by further treating routing as a distributed consensus problem [26]. Design and implementation problems in these protocols have the potential to deny vital network connectivity to a large fraction of users.

Unfortunately, it is notoriously difficult to develop reliable high-performance distributed systems that run over

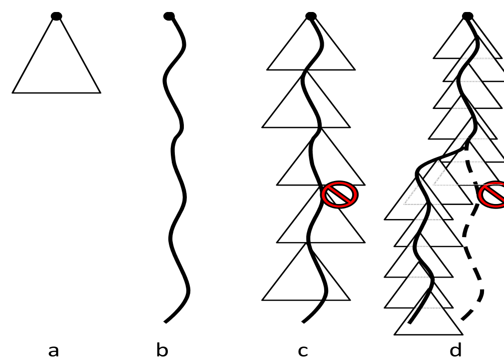

Figure 1: Execution path coverage by a) classic model checking, b) replay-based or live predicate checking, c) CrystalBall in deep online debugging mode, and d) CrystalBall in execution steering mode. Thick, curved lines are execution paths, dashed curved line is the avoided execution path that leads to an inconsistency, while triangles represent the state space searched by model checking.

asynchronous networks. Even if a distributed system is based on a well-understood distributed algorithm, its implementation can contain errors arising from complexities of realistic distributed environments or simply coding errors [27]. Many of these errors can only manifest after the system has been running for a long time, has developed a complex topology, and has experienced a particular sequence of low-probability events such as node resets. Consequently, it is difficult to detect such errors using testing and model checking, and many of such errors remain unfixed after the system is deployed.

We propose to leverage increases in computing power and bandwidth to make it easier to find errors in distributed systems, and to increase the resilience of the deployed systems with respect to any remaining errors. In our approach, distributed system nodes predict consequences of their actions while the system is running. Each node runs a state exploration algorithm on a consistent snapshot of its neighborhood and predicts which actions can lead to violations of user-specified consistency

properties. As Figure 1 illustrates, the ability to detect future inconsistencies allows us to address the problem of reliability in distributed systems on two fronts: debugging and resilience.

- Our technique enables deep online debugging because it explores more states than live runs alone or model checking. For each state that a running system experiences, our technique checks many additional states that the system did not go through, but that it could reach in similar executions. This approach combines benefits of distributed debugging and model checking.
- Our technique aids resilience because a node can modify its behavior to avoid a predicted inconsistency. We call this approach *execution steering*. Execution steering enables nodes to resolve non-determinism in ways that aim to minimize future inconsistencies.

To make this approach feasible, we need a fast state exploration algorithm. We describe a new algorithm, termed *consequence prediction*, which is efficient enough to detect future violations of safety properties in a running system. Using this approach we identified bugs in Mace implementations of a random overlay tree, and the Chord distributed hash table. These implementations were previously tested as well as model-checked by exhaustive state exploration starting from the initial system state. Our approach therefore enables the developer to uncover and correct bugs that were not detected using previous techniques. Moreover, we show that, if a bug is not detected during system development, our approach is effective in steering the execution away from erroneous states, without significantly degrading the performance of the distributed system service.

1.1 Contributions

We summarize the contributions of this paper as follows:

- We introduce the concept of continuously executing a state space exploration algorithm in parallel with a deployed distributed system, and introduce an algorithm that produces useful results even under tight time constraints arising from runtime deployment;
- We describe a mechanism for feeding a consistent snapshot of the neighborhood of a node in a large-scale distributed system into a running model checker; the mechanism enables reliable consequence prediction within limited time and bandwidth constraints;
- We present execution steering, a technique that enables the system to steer execution away from possible inconsistencies;
- We describe CrystalBall, the implementation of our approach on top of the Mace framework [21]. We evaluate CrystalBall on RandTree, Bullet′, Paxos, and Chord distributed system implementations. CrystalBall detected several previously unknown bugs that can cause system nodes to reach inconsistent states. Moreover, if the developer is not in a position to fix these bugs, CrystalBall's execution steering predicts them in a deployed system and steers execution away from them, all with an acceptable impact on the overall system performance.

1.2 Example

We next describe an example of an inconsistency exhibited by a distributed system, then show how CrystalBall predicts and avoids it. The inconsistency appears in the Mace [21] implementation of the RandTree overlay. RandTree implements a random, degree-constrained overlay tree designed to be resilient to node failures and network partitions. Trees built by an earlier version of this protocol serve as a control tree for a number of large-scale distributed services such as Bullet [23] and RanSub [24]. In general, trees are used in a variety of multicast scenarios (e.g., [3, 7]) and data collection/monitoring environments [17]. Inconsistencies in these environments translate to denial of service to users, data loss, inconsistent measurements, and suboptimal control decisions. The RandTree implementation was previously manually debugged both in local- and wide-area settings over a period of three years, as well as debugged using an existing model checking approach [22], but, to our knowledge, this inconsistency has not been discovered before (see Section 5 for some of the additional bugs that CrystalBall discovered).

RandTree topology. Nodes in a RandTree overlay form a directed tree of bounded degree. Each node maintains a list of its children and the address of the root. A node with the numerically smallest IP address acts as the root of the tree. Each non-root node contains an address of its parent. Children of the root maintain a sibling list. Note that, for a fixed node, its parent, children, and siblings are all distinct nodes. The seemingly simple task of maintaining a consistent tree topology is complicated by the requirement for groups of nodes to agree on their roles (root, parent, child, sibling) across asynchronous networks, in the face of node failures, and machine slowdowns.

Joining the overlay. A node n_j joins the overlay by issuing a Join request to one of the designated nodes. If the node receiving the join request is not the root, it forwards the request to the root. If the root already has the maximal number of children, it asks one of its children to incorporate the node into the overlay. Once the

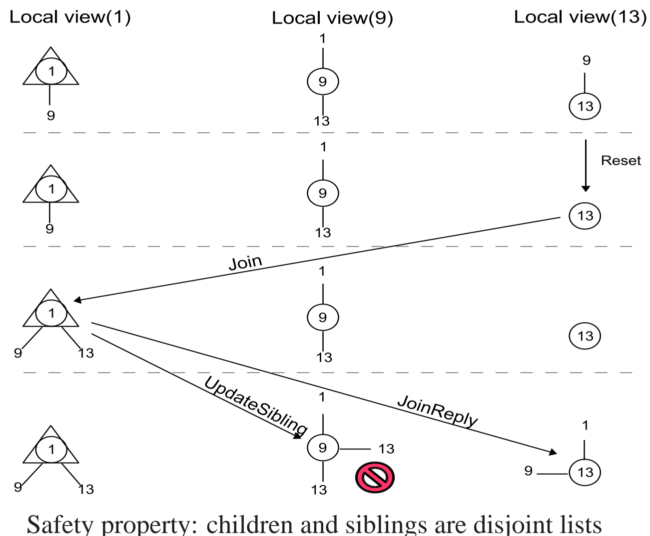

Figure 2: An inconsistency in a run of RandTree

request reaches a node n_p whose number of children is less than maximum allowed, node n_p inserts n_j as one of its children, and notifies n_j about a successful join using a JoinReply message (if n_p is the root, it also notifies its other children about their new sibling n_j using an UpdateSibling message).

Example system state. The first row of Figure 2 shows a state of the system that we encountered by running RandTree in the ModelNet cluster [43] starting from the initial state. We examine the local states of nodes n_1, n_9, and n_{13}. For each node n we display its neighborhood view as a small graph whose central node is n itself, marked with a circle. If a node is root and in a "joined" state, we mark it with a triangle in its own view.

The state in the first row of Figure 2 is formed by n_{13} joining as the only child of n_9 and then n_1 joining and assuming the role of the new root with n_9 as its only child (n_{13} remains as the only child of n_9). Although the final state shown in first row of Figure 2 is simple, it takes 13 steps of the distributed system (such as atomic handler executions, including application events) to reach this state from the initial state.

Scenario exhibiting inconsistency. Figure 2 describes a sequence of actions that leads to a state that violates the consistency of the tree. We use arrows to represent the sending and the receiving of some of the relevant messages. A dashed line separates distinct distributed system states (for simplicity we skip certain intermediate states and omit some messages).

The sequence begins by a silent reset of node n_{13} (such reset can be caused by, for example, a power failure). After the reset, n_{13} attempts to join the overlay again. The root n_1 accepts the join request and adds n_{13} as its child. Up to this point node n_9 received no information on actions that followed the reset of n_{13}, so n_9 maintains n_{13} as its own child. When n_1 accepts n_{13} as a child, it sends an UpdateSibling message to n_9. At this point, n_9 simply inserts n_{13} into the set of its sibling. As a result, n_{13} *appears both in the list of children and in the list of siblings of* n_9, which is inconsistent with the notion of a tree.

Challenges in finding inconsistencies. We would clearly like to avoid inconsistencies such as the one appearing in Figure 2. Once we have realized the presence of such inconsistency, we can, for example, modify the handler for the UpdateSibling message to remove the new sibling from the children list. Previously, researchers had successfully used explicit-state model checking to identify inconsistencies in distributed systems [22] and reported a number of safety and liveness bugs in Mace implementations. However, due to an exponential explosion of possible states, current techniques capable of model checking distributed system implementations take a prohibitively long time to identify inconsistencies, even for seemingly short sequences such as the ones needed to generate states in Figure 2. For example, when we applied the Mace Model Checker's [22] exhaustive search to the safety properties of RandTree starting from the initial state, it failed to identify the inconsistency in Figure 2 even after running for 17 hours (on a 3.4-GHz Pentium-4 Xeon that we used for all our experiments in Section 5). The reason for this long running time is the large number of states reachable from the initial state up to the depth at which the bug occurs, all of which are examined by an exhaustive search.

1.3 CrystalBall Overview

Instead of running the model checker from the initial state, we propose to execute a model checker concurrently with the running distributed system, and continuously feed current system states into the model checker. When, in our example, the system reaches the state at the beginning of Figure 2, the model checker will predict the state at the end of Figure 2 as a possible future inconsistency. In summary, instead of trying to predict all possible inconsistencies starting from the initial state (which for complex protocols means never exploring states beyond the initialization phase), our model checker predicts inconsistencies that can occur in a system that has been running for a significant amount of time in a realistic environment.

As Figure 1 suggests, compared to the standard model checking approach, this approach identifies inconsistencies that can occur within much longer system executions. Compared to simply running the system for a long time, our approach has two advantages.

1. Our approach systematically covers a large number of executions that contain low-probability events,

such as node resets that ultimately triggered the inconsistency in Figure 2. It can take a very long time for a running system to encounter such a scenario, which makes testing for possible bugs difficult. Our technique therefore improves system debugging by providing a new technique that combines some of the advantages of testing and static analysis.

2. Our approach identifies inconsistencies before they actually occur. This aspect of our approach opens an entirely new possibility: adapt the behavior of the running system on the fly and avoid an inconsistency. We call this technique *execution steering*. Because it does not rely on a history of past inconsistencies, execution steering is applicable even to inconsistencies that were previously never observed in past executions.

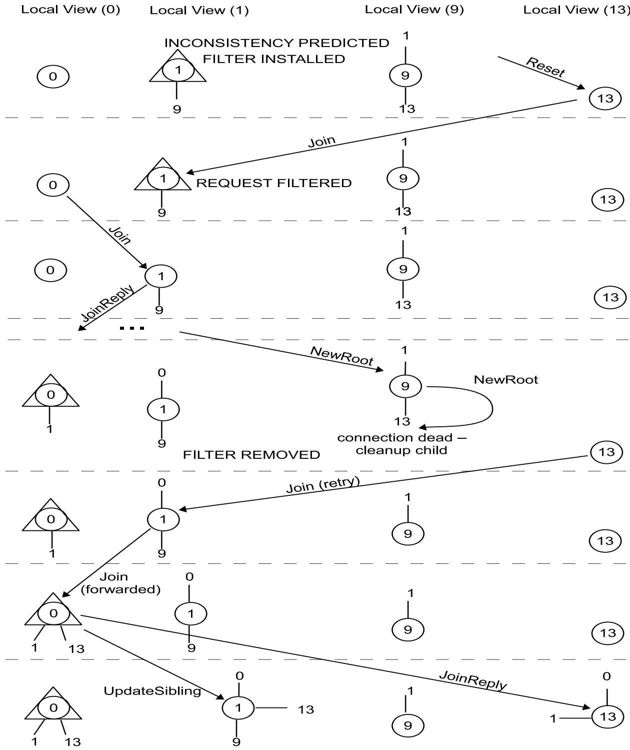

Figure 3: An Example execution sequence that avoids the inconsistency from Figure 2 thanks to execution steering

Preventing inconsistency by execution steering. In our example, a model checking algorithm running in n_1 detects the violation at the end of Figure 2. Given this knowledge, execution steering causes node n_1 not to respond to the join request of n_{13} and to break the TCP connection with it. Node n_{13} eventually succeeds joining the random tree (perhaps after some other nodes have joined first). The stale information about n_{13} in n_9 is removed once n_9 discovers that the stale communication channel with n_{13} is closed, which occurs the first time when n_9 attempts to communicate with n_{13}. Figure 3 presents one scenario illustrating this alternate execution sequence. Effectively, execution steering has explored the non-determinism and robustness of the system to choose an alternative execution path that does not contain the inconsistency.

Detecting the inconsistency using consequence prediction. We believe that inconsistency detection and execution steering are compelling reasons to use an approach where a model checker is deployed online to find future inconsistencies. But to make this approach feasible, it is essential to have a model checking technique capable of quickly discovering potential inconsistencies at significant depths in a very short amount of time. The previous model checking technique is not sufficient for this purpose: when we tried deploying it online, by the time a future inconsistency was identified, the system had already passed the execution depth at which the inconsistency occurs. We need an exploration technique that is sufficiently fast and focused to be able to discover a future inconsistency in the time that it takes node interaction to cause the inconsistency in a distributed system. We present such an exploration technique, termed *consequence prediction*.

Consequence prediction focuses on exploring causally related chains of events. Our system identifies the scenario in Figure 2 by running consequence prediction on node n_1. Consequence prediction considers, among others, the Reset action on node n_{13}. It then uses the fact that the Reset action brings the node into a state where it can issue a Join request. Even though there are many transitions that a distributed system could take at this point, consequence prediction focuses on the transitions that were enabled by the recent state change. It will therefore examine the consequences of the response of n_1 to the Join request and, using the knowledge of the state of its neighborhood, discover a possible inconsistency that could occur in n_9. Consequence prediction also explores other possible sequences of events, but, as we explain in Section 3.2, it avoids certain sequences, which makes it faster than applying the standard search to the same search depth.

Obtaining neighborhood snapshots under limited resources. Consequence prediction addresses the question of performing a fast search from the given state of the distributed system. An important question that remains to be answered is how to obtain such a global state of the distributed system. Our approach considers a subset of the distributed system nodes visible within some neighborhood. To obtain a consistent neighborhood snapshot, our system uses logical clocks, with each node taking a checkpoint before increasing its logical clock. To predict

consequences of its actions, a node first issues requests to its neighbors to obtain their checkpoints at its current logical time, and then uses the collected neighborhood snapshot as the starting point for the consequence prediction algorithm.

2 Background

We next present a simple model of distributed systems and describe a basic model checking algorithm based on breadth-first search and state caching.

2.1 System Model

Figure 4 describes a simple model of a distributed system. We use this model to describe system execution at a high level of abstraction, describe an existing model checking algorithm, and present our new algorithm, consequence prediction. (The model does not attempt to describe our technique for obtaining consistent snapshots.)

System state. The state of the entire distributed system is given by 1) local state of each node, and 2) in-flight network messages. We assume a finite set of node identifiers N (corresponding to, for, example, IP addresses). Each node $n \in N$ has a local state $L(n) \in S$. Local state models all node-local information such as explicit state variables of the distributed node implementation, the status of timers, and the state that determines application calls. Network state is given by in-flight messages, I. We represent each in-flight message by a pair (N, M) where N is the destination node of the message and M is the remaining message content (including sender node information and message body).

Node behavior. Each node in our system runs the same state-machine implementation. The state machine is given by two kinds of handlers: a message handler executes in response to a network message; an internal handler executes in response to a node-local event such as a timer and an application call.

We represent message handlers by a set of tuples H_M. The condition $((s_1, m), (s_2, c)) \in H_M$ means that, if a node is in state s_1 and it receives a message m, then it transitions into state s_2 and sends the set c of messages. Each element $(n', m') \in c$ is a message with target destination node n' and content m'. Internal node action handler is analogous to a message handler, but it does not consume a network message. Instead, $((s_1, a), (s_2, c)) \in H_A$ represents handling of an internal node action $a \in A$. (In both handlers, if c is the empty set, it means that the handler did not generate any messages.)

System behavior. The behavior of the system specifies one step of a transition from one global distributed system state (L, I) to another global state (L', I'). We denote this transition by $(L, I) \rightsquigarrow (L', I')$ and describe it in Figure 4 in terms of handlers H_M and H_A. The handler that sends the message directly inserts the message into the network state I, whereas the handler receiving the message simply removes it from I. To keep the model simple, we assume that transport errors are particular messages, generated and processed by message handlers.

basic notions:
N – node identifiers
S – node states
M – message contents
$N \times M$ – (destination process, message)-pair
$C = 2^{N \times M}$ – set of messages with destination
A – local node actions (timers, application calls)

system state : $(L, I) \in G, \;\; G = 2^{N \times S} \times 2^{N \times M}$
local node states : $L \subseteq N \times S$ (function from N to S)
in-flight messages (network) : $I \subseteq N \times M$

behavior functions for each node :
message handler : $H_M \subseteq (S \times M) \times (S \times C)$
internal action handler : $H_A \subseteq (S \times A) \times (S \times C)$

transition function for distributed system :

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H_M}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I_0 \uplus \{(n, m)\}) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I_0 \uplus c)\end{array}}$$

internal node action (timer, application calls) :

$$\frac{((s_1, a), (s_2, c)) \in H_A}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2\}, I \uplus c))\end{array}}$$

Figure 4: A Simple Model of a Distributed System

2.2 Model-Checking Distributed Systems

Figure 5 presents a standard search for finding safety violations in a transition system given by relation $\rightsquigarrow$. The search starts from a given global state firstState, which, in the standard approach, is the initial state of the system. The search systematically explores reachable global states at larger and larger depths and checks whether the states satisfy the given property condition. In practice, the number of reachable states is very large and the search needs to be terminated upon exceeding some bound such as running time or search depth. The condition of exceeding some bound is denoted StopCriterion in Figure 5.

```
1  proc findErrors(firstState : G, property : (G → boolean)) {
2    explored = emptySet(); errors = emptySet();
3    frontier = emptyQueue();
4    frontier.addLast(firstState);
5    while (!StopCriterion) {
6      state = frontier.popFirst();
7      if (!property(state))
8        errors.add(state);
9      explored.add(hash(state));
10     foreach (nextState where (state ⇝ nextState))
11       if (!explored.contains(hash(nextState)))
12         frontier.addLast(nextState);
13   }
14 }
```

Figure 5: Finding errors using state space exploration

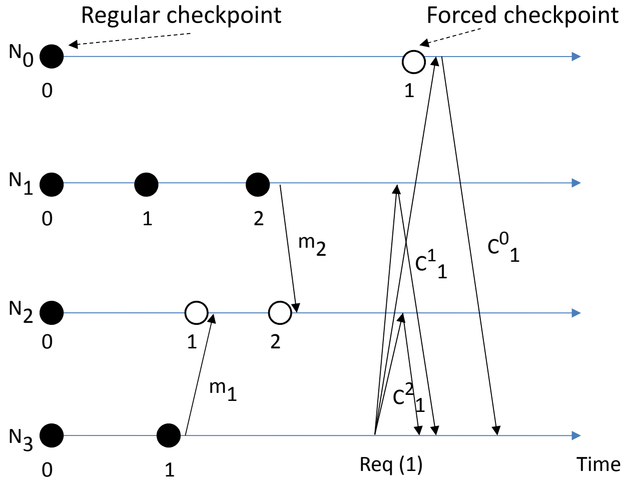

Figure 6: Example illustrating the consistent snapshot collection algorithm. Black ovals represent regular checkpoints. Messages m_1 and m_2 force checkpoints (white ovals) to be taken before messages are processed at nodes 2 and 1, respectively, and so does the checkpoint request from node 3 when it arrives at node 0.

2.3 Consistent Global Snapshots

Examining global state of a distributed system is useful in a variety of scenarios, such as checkpointing/recovery, debugging, and, in our case, running a model checking algorithm in parallel with the system. A *snapshot* consists of *checkpoints* of nodes' states. For the snapshot to be useful, it needs to be consistent. There has been a large body of work in this area, starting with the seminal paper by Chandy and Lamport [5]. We next describe one of the recent algorithms for obtaining consistent snapshots [29]. The general idea is to collect a set of checkpoints which do not violate the happens-before relationship [25] established by messages sent by the distributed service.

In this algorithm, the runtime of each node n_i keeps track of the checkpoint number cn_i (the role of

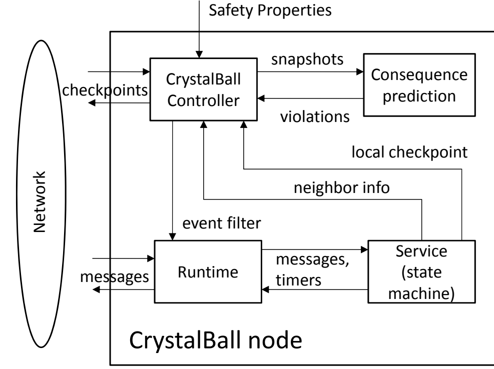

Figure 7: High-level overview of CrystalBall

checkpoint number is similar to the Lamport's logical clock [25]). Whenever n_i sends a message M, it stores cn_i in it (denote this value $M.cn$). When node n_j receives a message, it compares cn_j with $M.cn$. If $M.cn > cn_j$, then n_j takes a checkpoint C, assigns $C.cn = M.cn$, and sets $cn_j = M.cn$. This is the key step of the algorithm that avoids violating the happens-before relationship. A node n_i can take snapshots on its own, and this is done whenever the cn_i is locally incremented, which happens periodically.

To collect the required checkpoints, a node n_i sends a checkpoint request message containing a checkpoint request number cr_i. Upon receiving the request, a node n_j responds with the appropriate checkpoint. There are two cases: 1) if $cr_i > cn_j$ (the request number is greater than any number n_j has seen), then n_j takes a checkpoint, stamps it with $C.cn = cr_i$, sets $cn_j = cr_i$, and sends that checkpoint; 2) if $cr_i \leq cn_j$, the request is for a checkpoint taken in the past, and n_j responds with the earliest checkpoint C for which $C.cn \geq cr_i$.

3 CrystalBall Design

Figure 7 shows the high-level overview of a CrystalBall-enabled node. We concentrate on distributed systems implemented as state machines, as this is a widely-used approach [21, 25, 26, 37, 39].

The state machine interfaces with the outside world via the runtime module. The runtime receives the messages coming from the network, demultiplexes them, and invokes the appropriate state machine handlers. The runtime also accepts application level messages from the state machines and manages the appropriate network connections to deliver them to the target machines. This module also maintains the timers on behalf of all services that are running.

The CrystalBall controller contains a checkpoint manager that periodically collects consistent snapshots of a node's neighborhood. The controller feeds them to the

model checker, along with a checkpoint of the local state. The model checker runs the consequence prediction algorithm which checks user- or developer-defined properties and reports any violation in the form of a sequence of events that leads to an erroneous state.

CrystalBall can operate in two modes. In the deep online debugging mode the controller only outputs the information about the property violation. In the execution steering mode the controller examines the report from the model checker, prepares an *event filter* that can avoid the erroneous condition, checks the filter's impact, and installs it into the runtime if it is deemed to be safe.

3.1 Consistent Neighborhood Snapshots

To check system properties, the model checker requires a snapshot of the system-wide state. Ideally, every node would have a consistent, up-to-date checkpoint of every other participant's state. Doing so would give every node high confidence in the reports produced by the model checker. However, given that the nodes could be spread over a high-latency wide-area network, this goal is unattainable. In addition, the sheer amount of bandwidth required to disseminate checkpoints might be excessive.

Given these fundamental limitations, we use a solution that aims for scalability: we apply model checking to a *subset* of all states in a distributed system. We leverage the fact that in scalable systems a node typically communicates with a small subset of other participants ("neighbors") and perform model checking only on this neighborhood. For example, a distributed hash table node keeps track of $O(\log n)$ other nodes. Similarly, in mesh-based content distribution systems nodes communicate with a constant number of peers, or this number does not explicitly grow with the size of the system. In a random overlay tree, a node is typically aware of the root, its parent, its children, and its siblings. Therefore, we arrange for a node to distribute its state checkpoints to its neighbors, and we refer to them as *snapshot neighborhood*.

The *checkpoint manager* maintains checkpoints and snapshots. Other CrystalBall components can request an on-demand snapshot to be gathered by invoking an appropriate call on the checkpoint manager.

Discovering and Managing Snapshot Neighborhoods. To propagate checkpoints, the checkpoint manager needs to know the set of a node's neighbors. This set is dependent upon a particular distributed service. We use two techniques to provide this list. In the first scheme, we ask the developer to implement a method that will return the list of neighbors. The checkpoint manager then periodically queries the service and updates its snapshot neighborhood.

Since changing the service code might not always be possible, our second technique uses a heuristic to determine the snapshot neighborhood. Specifically, we periodically query the runtime to obtain the list of open connections (for TCP), and recent message recipients (for UDP). We then cluster connection endpoints according to the communication times, and selects a sufficiently large cluster of recent connections. After filtering duplicate addresses, we initialize the snapshot neighborhood to the resulting list.

Enforcing Snapshot Consistency. To avoid false positives, we ensure that the neighborhood snapshot corresponds to a consistent view of a distributed system at some point of logical time. Our starting point is a technique similar to the one described in Section 2.3. However, instead of gathering a global snapshot, a node periodically sends a checkpoint request to the members of its snapshot neighborhood.

Node failures are commonplace in distributed systems, and our algorithm has to deal with them. The checkpoint manager proclaims a node to be dead if it experiences a communication error (e.g., a broken TCP connection) with it while collecting a snapshot. An additional cause for an apparent node failure is a change of a node's snapshot neighborhood in the normal course of operation (e.g., when a node changes parents in the random tree). In this case, the node triggers a new snapshot gather operation.

Managing Checkpoint Storage. The checkpoint manager keeps track of checkpoints via their checkpoint numbers. Over the course of its operation, a node can collect a large number of checkpoints, and a long-running system might demand an excessive amount of memory and storage for this task. It is therefore important to prune old checkpoints in a way that nevertheless leaves the ability to gather consistent snapshots.

Our approach to managing checkpoint storage is to enforce a per-node storage quota for checkpoints. Older checkpoints are removed first to make room. Removing older checkpoints might cause a checkpoint request to fail when the request is asking for a checkpoint that is outside of the remaining range of checkpoints at the node. In this case, the node responds negatively to the checkpoint requester and inserts its current checkpoint number in the response ($R.cn = cn_i$). Then, upon receiving the responses from all nodes in the snapshot neighborhood, the requestor chooses the greatest among the $R.cn$ received, and initiates another snapshot round. Provided that the rate at which the snapshots are removed is not greater than the rate at which the nodes are communicating, this second snapshot collection will likely succeed.

Managing Bandwidth Consumption. For a large class of services, the relevant per-node state is relatively small, e.g., a few KB. It is nevertheless important to limit band-

```
1  proc findConseq(currentState : G, property : (G → boolean)) {
2    explored = emptySet(); errors = emptySet();
3    localExplored = emptySet();
4    frontier = emptyQueue();
5    frontier.addLast(currentState);
6    while (!STOP_CRITERION) {
7      state = frontier.popFirst();
8      if (!property(state))
9        errors.add(state); // predicted inconsistency found
10     explored.add(hash(state));
11     foreach ((n,s) ∈ state.L) { // node n in local state s
12       // process all network handlers
13       foreach (((s,m),(s',c)) ∈ H_M where (n,m) ∈ state.I)
14         // node n handles message m according to st. machine
15         addNextState(state,n,s,s',{m},c);
16       // process local actions only for fresh local states
17       if (!localExplored.contains(hash(n,s)))
18         foreach (((s,a),(s',c)) ∈ H_A)
19           addNextState(state,n,s,s',{},c);
20       localExplored.add(hash(n,s));
21     }
22   }
23 }
24 proc addNextState(state,n,s,s',c0,c) {
25   nextState.L = (state.L \ {(n,s)}) ∪ {(n,s')};
26   nextState.I = (state.I \ c0) ∪ c;
27   if (!explored.contains(hash(nextState)))
28     frontier.addLast(nextState);
29 }
```

Figure 8: Consequence Prediction Algorithm

width consumed by state checkpoints for a number of reasons: 1) sending large amounts of data might congest the node's outbound link, and 2) consuming bandwidth for checkpoints might adversely affect the performance and the reaction time of the system.

To reduce the amount of checkpoint data we transmit, CrystalBall can use a number of techniques. First, it can employ "diffs" that enable a node to transmit only parts of state that are different from the last sent checkpoint. Second, the checkpoints can be compressed on-the-fly. Finally, CrystalBall can enforce a bandwidth limit by: 1) making the checkpoint data be a fraction of all data sent by a node, or 2) enforcing an absolute bandwidth limit (e.g., 10 kbps). If the checkpoint manager is above the bandwidth limit, it responds with a negative response to a checkpoint request and the requester temporarily removes the node from the current snapshot. A node that wishes to reduce its inbound bandwidth consumption can reduce the rate at which it requests checkpoints from other nodes.

3.2 Consequence Prediction Algorithm

The key to enabling fast prediction of future inconsistencies in CrystalBall is our consequence prediction algorithm, presented in Figure 8. In its overall structure, the algorithm is similar to the standard state-space search in Figure 5. (We present the algorithm at a more concrete level, where the relation $\rightsquigarrow$ is expressed in terms of action handlers H_A and H_M introduced in Figure 4.) In fact, if we omitted the test in Line 16,

if (!localExplored.contains(hash(n,s)))

the algorithm would reduce precisely to Figure 5. The test in Line 17 removes from the search the transitions generated by local action handlers of node n if node n has been previously explored with the same state s. As a result, local actions of node n in state s will never be considered more than once, regardless of what other components of global state are explored.

Avoiding Interleavings. Although simple, the idea of removing from the search actions of nodes with previously seen states has a profound impact on the search depth that the model checker can feasibly reach with a limited time budget. This change was therefore key to enabling the use of the model checker at runtime.

Exploring Consequence Chains. Knowing that consequence prediction avoids considering certain states, the question remains whether the remaining states are sufficient to make the search useful. Ultimately, the answer to this question comes from our experimental evaluation (Section 5). In addition, there are several intuitive reasons to expect consequence prediction to give good results. Note first that consequence prediction explores all possible transitions from the initial state (because at that point localExplored is empty).

Furthermore, consequence prediction considers all chains of actions where one action causes a state change that triggers the next action. The reason is simply that the algorithm explores all outgoing transitions at a node whose state has changed into a previously unseen state.

3.3 Execution Steering

CrystalBall's execution steering mode enables the system to avoid entering an erroneous state by steering its execution path away from predicted inconsistencies. If a protocol was designed with execution steering in mind, the runtime system could report a predicted inconsistency as a special programming language exception, and allow the service to react to the problem using a service-specific policy. However, to measure the impact on existing implementations, this paper focuses on generic runtime mechanisms that do not require the developer to insert exception-handling code.

Choice of Corrective Actions. Recall that a node in our framework operates as a state machine and processes messages, timer events, and application calls via handlers. Upon noticing that running a certain handler can lead to an erroneous state, CrystalBall installs an *event filter*, which temporarily blocks the invocation of the state machine handler for messages from the relevant sender. The rationale is that a distributed system often

contains a large amount of non-determinism that allows it to proceed even if certain transitions are disabled. For example, if the offending message is a Join request in a random tree, ignoring the message can prevent violating a local state property. The joining nodes can later retry the procedure with an alternative potential parent and successfully join the tree. Similarly, if handling a message causes an equivalent of a race condition manifested as an inconsistency, delaying message handling allows the system to proceed to the point where handling the message becomes safe again.

Distributed systems that use TCP typically include failure handling code that deals with broken TCP connections. Therefore, in case of network messages sent over TCP, an alternative to simple blocking is to additionally reset the connection with the sender of the message. The reason for resetting the connection is to signal to the sender of the offending message that something went wrong. In many cases, cleaning out the relevant state at the target node can prevent other bugs from manifesting themselves.

In general, execution steering can intervene at several points in the execution path. Our current policy is to steer the execution as early as possible. For example, if the erroneous execution path involves a node issuing a Join request after resetting, the system's first interaction with that node occurs at the node which receives its join request. If this node discoveres the erroneous path, it can install the event filter.

Ensuring Safety of Event Filter Actions. Ideally, execution steering would always prevent inconsistencies from occurring, without introducing new inconsistencies due to a change in behavior. In general, however, guaranteeing the absence of inconsistencies is as difficult as guaranteeing that the entire program is error-free. CrystalBall therefore makes execution steering safe in practice through the following two design decisions.

First, CrystalBall chooses as steering actions those behaviors that could normally occur in a realistic distributed system. For example, breaking the TCP connection is an event that could anyway occur in a distributed system, so the protocols are designed to tolerate it.

Second, before allowing the event filter to perform an execution steering action, CrystalBall runs the consequence prediction algorithm to check the effect of the event filter action on the distributed system. If the consequence prediction algorithm does not suggest that the filter actions are safe, CrystalBall does not attempt execution steering and leaves the system to proceed as usual.

Rechecking Previously Discovered Violations. An event filter reflects possible future inconsistencies reachable from the current state, and leaving an event filter in place indefinitely could deny service to some distributed system participants. CrystalBall therefore removes the filters from the runtime after every model checking run. However, it is useful to quickly check whether the previously identified error path can still lead to an erroneous condition in a new model checking run. This is especially important given the asynchronous nature of the model checker relative to the system messages, which can prevent the model checker from running long enough to rediscover the problem. To prevent this from happening, the first step executed by the model checker is to replay the previously discovered error paths. If the problem reappears, CrystalBall immediately reinstalls the appropriate filter.

Immediate Safety Check. CrystalBall also supports *immediate safety check*, a mechanism that avoids inconsistencies that would be caused by executing the current handler. Such imminent inconsistencies can happen even in the presence of execution steering because 1) consequence prediction explores states given by only a subset of all distributed system nodes, and 2) the model checker runs asynchronously and may not always detect inconsistencies in time. The immediate safety check speculatively runs the handler, checks the consistency properties in the resulting state, and prevents actual handler execution if the resulting state is inconsistent.

4 Implementation Highlights

Our CrystalBall prototype is built on top of Mace [21]. Mace allows distributed systems to be specified succinctly, and it outputs high-performance C++ code. We run the model checker as a separate thread that communicates future inconsistencies to the runtime. Our implementation includes a checkpoint manager, which enables each service to collect and manage checkpoints to generate consistent neighborhood snapshots based on a notion of logical time. It also includes implementation of consequence prediction algorithm, with the ability to replay paths previously found to lead to inconsistencies. Finally, it contains implementation of the execution steering mechanism.

Checkpoint Manager. To collect and manage snapshots, we modified the Mace compiler and the runtime. We added a `snapshot on` directive to the service description to inform the Mace compiler and the runtime that the service requires checkpointing. The presence of this directive causes the compiler to generate the necessary code. For example, it automatically inserts a checkpoint number in every service message and adds the code to invoke the checkpoint manager when that is required by the snapshot algorithm.

The checkpoint manager itself is implemented as a Mace service, and it compresses the checkpoints using the LZW algorithm. To further reduce bandwidth consumption, a node checks if the previously sent check-

point is identical to the new one (on per-peer basis), and avoids transmitting duplicate data.

Consequence Prediction. Our starting point for the consequence prediction algorithm was the publicly available MaceMC implementation. This code was not designed to work with live state. For example, the node addresses in the code are assumed to be of the form 0,1,2,3, etc. To handle this issue, we added a mapping from live IP addresses to model checker addresses. Since the model checker is executing real code in the event and the message handlers, we did not encounter any additional addressing-related issues.

Another change we made allowed the model checker to scale to hundreds of nodes and deal with partial system state. We introduced a dummy node that represents all system nodes without checkpoints in the current snapshot. All messages sent to such nodes are redirected to the dummy node. The model checker does not consider the events of this node during state exploration.

To minimize the impact on distributed service performance, we decouple the model checker from event processing path by running it as a separate process. On a multi-core machine this CPU-intensive process will likely be scheduled on a separate core.

Immediate safety check. Our current implementation of the immediate safety check executes the handler in a copy of the state machine's virtual memory (using fork()), and holds the transmission of messages until the successful completion of the consistency check. Upon encountering an inconsistency in the copy, the runtime does not execute the handler in the primary state machine. Other approaches [41, 32] are also possible in our context.

Replaying Past Erroneous Paths. To ascertain that an inconsistency can still occur from the current snapshot, we replay past erroneous paths. Strictly replaying a sequence of events and messages that form a path on a new a neighborhood snapshot might be incorrect. For example, some messages could have only been generated by the old state checkpoint and are inconsistent with new state. Our replay technique therefore replays only timer and application events, and relies on the distributed service code to generate any messages. We then follow the causality of the newly generated messages throughout the system. We deterministically replay pseudo-random number generation.

Event Filtering for Execution steering. Execution steering is driven by the report from the model checker, which produces a sequence of events and messages. Upon checking the existence and the potential impact of a corrective action, the CrystalBall controller installs an event filter into the runtime. In case of network messages, this filter contains a message type, message source and the destination. For other events, e.g., a local timer event or application call, the filter just contains the identity of the handler that handles the event. Unlike the network messages that the filter drops when it triggers, the timer events are rescheduled.

Checking Safety of Event Filters. To check for safety of event filters, we modified our baseline consequence prediction algorithm. Specifically, upon encountering an inconsistency, we allow consequence prediction to pursue actions that an event filter could perform.

5 Evaluation

Our experimental evaluation addresses the following questions: **1)** Is CrystalBall effective in finding bugs in live runs? **2)** Can any of the bugs found by CrystalBall also be identified by the MaceMC model checker alone? **3)** Is execution steering capable of avoiding inconsistencies in deployed distributed systems? **4)** Are the overheads introduced by CrystalBall within acceptable levels?

5.1 Experimental Setup

We conducted our live experiments using ModelNet [43]. ModelNet allows us to run live code in a cluster of machines, while application packets are subjected to packet delay, loss, and congestion typical of the Internet. Our cluster consists of 17 older machines with dual 3.4 GHz Pentium-4 Xeons with hyper-threading and 8 machines with dual 2.33 Ghz dual-core Xeon 5140s. Older machines have 2 GB of RAM, while the newer ones have 4 GB. These machines run GNU/Linux 2.6.17. One 3.4 GHz Pentium-4 machine running FreeBSD 4.9 served as the ModelNet packet forwarder for these experiments. All machines are interconnected with a full-rate 1-Gbps Ethernet switch.

We consider two deployment scenarios. For our large-scale experiments with deep online debugging, we multiplex 100 logical end hosts running the distributed service across the 20 Linux machines, with 2 participants running the model checker on 2 different machines. We run with 6 participants for small-scale debugging experiments, one per machine.

We use a 5,000-node INET [6] topology that we further annotate with bandwidth capacities for each link. The INET topology preserves the power law distribution of node degrees in the Internet. We keep the latencies generated by the topology generator; the average network RTT is 130ms. We randomly assign participants to act as clients connected to one-degree stub nodes in the topology. We set transit-transit links to be 100 Mbps, while we set access links to 5 Mbps/1 Mbps inbound-/outbound bandwidth. To emulate the effects of cross traffic, we instruct ModelNet to drop packets at random with a probability chosen uniformly at random between [0.001,0.005] separately for each link.

5.2 Deep Online Debugging Experience

We have used CrystalBall to find inconsistencies (violations of safety properties) in two mature implemented protocols in Mace, namely an overlay tree (RandTree) and a distributed hash table (Chord [42]). These implementation were not only manually debugged both in local- and wide-area settings, but were also model checked using MaceMC [22]. We have also used our tool to find inconsistencies in Bullet′, a file distribution system that was originally implemented in MACEDON [37], and then ported to Mace. We found 13 new subtle bugs in these three systems that caused violation of safety properties. Except one, the violations were beyond the scope of exhaustive search by existing software model checker, typically because the errors manifested themselves at depths far beyond what can be exhaustively searched.

System	Bugs found	LOC Mace/C++
RandTree	7	309 / 2000
Chord	3	254 / 2200
Bullet′	3	2870 / 19628

Table 1: Summary of inconsistencies found for each system using CrystalBall. LOC stands for lines of code and reflects both the MACE code size and the generated C++ code size. The low LOC counts for Mace service implementations are a result of Mace's ability to express these services succintly.

Table 1 summarizes the inconsistencies that CrystalBall found in RandTree, Chord and Bullet′. Typical elapsed times (wall clock time) until finding an inconsistency in our runs have been from less than an hour up to a day. This time allowed the system being debugged to go through complex realistic scenarios. CrystalBall identified inconsistencies by running consequence prediction from the current state of the system for up to several hundred seconds. To demonstrate their depth and complexity, we detail four out of 13 inconsistencies we found in the three services we examined.

5.2.1 Example RandTree Bugs Found

We next discuss bugs we identified in the RandTree overlay protocol presented in Section 1.2. We name bugs according to the consistency properties that they violate.

Children and Siblings Disjoint. The first safety property we considered is that the children and sibling lists should be disjoint. CrystalBall identified the scenario from Figure 2 in Section 1.2 that violates this property. The problem can be corrected by removing the stale information about children in the handler for the UpdateSibling message. CrystalBall also identified variations of this bug that requires changes in other handlers.

Root is Not a Child or Sibling. CrystalBall found violation of property that root node should not appear as a child, identifying a node 9 that considers itself a root but at the same time another node 69 considers it to be a child.

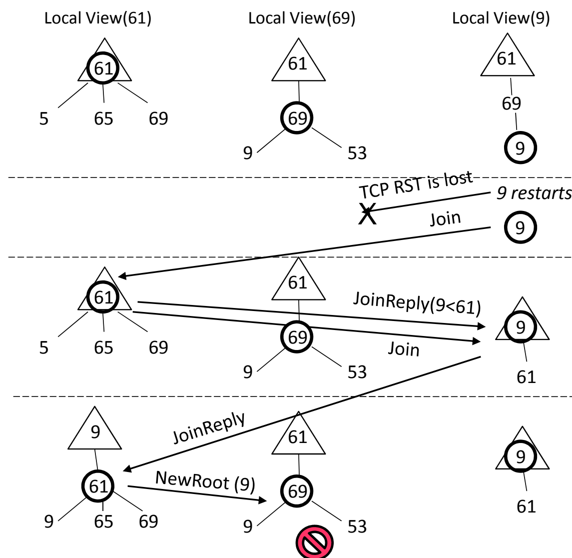

Figure 9: An inconsistency in a run of RandTree. Root (9) appears as a child).

Scenario exhibiting inconsistency. During live execution, node 61 is initially the root of the tree and parent of nodes 5, 65, and 69. At this point, consequence prediction detects the following scenario. Node 9 resets, but its TCP RST packet to its parent (69) is lost. 9 sends a Join request to 61. Based on 9's identifier, 61 considers 9 more eligible and selects it as the new root and sends it a Join . After receiving a JoinReply from 9, 61 informs its children about the new root (9) by sending NewRoot packets to them. However, 69 still thinks 9 is its child, which causes the inconsistency.

Possible correction. Check the children list whenever installing information about the new root node.

Root Has No Siblings. CrystalBall found violation of property that root node should contain no sibling pointers, identifying a node A that considers itself a root but at the same time has an address of another node B in its sibling list.

Scenario exhibiting inconsistency. During live execution, A is initially the root of the tree and parent of B and C. Node R sends a Join request to A. Based on R's identifier, A considers R more eligible and selects it as the new root. A informs its children about the new root by sending NewRoot packets to them. At this point, consequence prediction detects the following scenario. A experiences a node reset and resets the TCP connections with its children B and C. Upon receiving the error signal, B removes A from its parent pointer and promotes itself to be the root. However, it keeps its stale sibling

list, which causes the inconsistency.

Possible correction. Clean the sibling list whenever a node relinquishes the root position in favor of another node.

Recovery Timer Should Always Run. An important safety property for RandTree is that the recovery timer should always be scheduled. This timer periodically causes the nodes to send Probe messages to the peer list members with which it does not have direct connection. It is vital for the tree's consistency to keep nodes up-to-date about the global structure of the tree. The property was written by the authors of [22] but the authors did not report any violations of it. We believe that our approach discovered it in part because our experiments considered more complex join scenarios.

Scenario exhibiting inconsistency. CrystalBall found a violation of the property in a state where node A joins itself, and changes its state to "joined" but does not schedule any timers. Although this does not cause problems immediately, the inconsistency happens when another node B with smaller identifier tries to join, at which point A gives up the root position, selects B as the root, and adds B it to its peer list. At this point A has a non-empty peer list but no running timer.

Possible correction. Keep the timer scheduler even when a node has an empty peer list.

5.2.2 Example Chord Bugs Found

We next describe violations of consistency properties in Chord [42], a distributed hash table that provides key-based routing functionality. Chord and other related distributed hash tables form a backbone of a large number of proposed and deployed distributed systems [17, 35, 38].

Chord topology. Each Chord node is assigned a Chord id (effectively, a key). Nodes arrange themselves in an overlay ring where each node keeps pointers to its predecessor and successor. Even in the face of asynchronous message delivery and node failures, Chord has to maintain a ring in which the nodes are ordered according to their ids, and each node has a set of "fingers" that enables it to reach exponentially larger distances on the ring.

Joining the system. To join the Chord ring, a node A first identifies its potential predecessor by querying with its id. This request is routed to the appropriate node P, which in turn replies to A. Upon receiving the reply, A inserts itself between P and P's successor, and sends the appropriate messages to its predecessor and successor nodes to update their pointers. A "stabilize" timer periodically updates these pointers.

Property: If Successor is Self, So Is Predecessor. If a predecessor of a node A equals A, then its successor must also be A (because then A is the only node in the ring). This is a safety property of Chord that had been extensively checked using MaceMC, presumably using both exhaustive search and random walks.

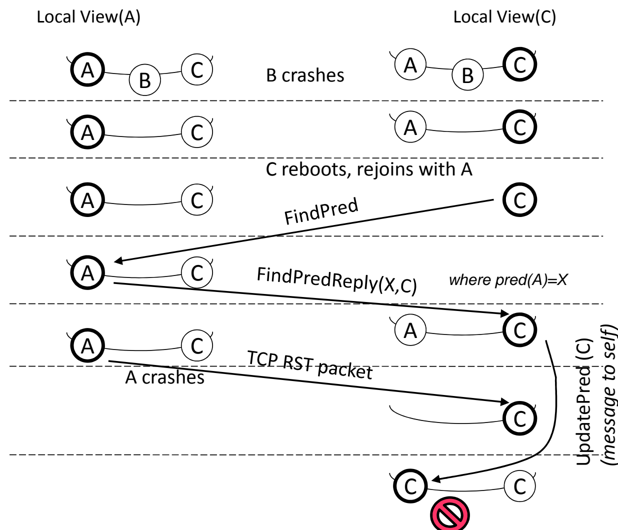

Figure 10: An inconsistency in a run of Chord. Node C has its predecessor pointing to itself while its successor list includes other nodes.

Scenario exhibiting inconsistency: CrystalBall found a state where node A has A as a predecessor but has another node B as its successor. This violation happens at depths that are beyond those reachable by exhaustive search from the initial state. During live execution, several nodes join the ring and all have a consistent view of the ring. Three nodes A, B, and C are placed consecutively on the ring, i.e., A is predecessor of B and B is predecessor of C. Then B experiences a node reset and other nodes which have established TCP connection with B receive a TCP RST. Upon receiving this error, node A removes B from its internal data structures. As a consequence, Node A considers C as its immediate successor.

Starting from this state, consequence prediction detects the following scenario that leads to violation. C experiences a node reset, losing all its state. C then tries to rejoin the ring and sends a FindPred message to A. Because nodes A and C did not have an established TCP connection, A does not observe the reset of C. Node A replies to C by a FindPredReply message that shows A's successor to be C. Upon receiving this message, node C i) sets its predecessor to A; ii) stores the successor list included in the message as its successor list; and iii) sends an UpdatePred message to A's successor which, in this case, is C itself. After sending this message, C receives a transport error from A and removes A from all of its internal structures including the predecessor pointer. In other words, C's predecessor would be unset. Upon receiving the (loopback) message to itself, C observes that the predecessor is unset and then sets it to the sender of the UpdatePred message which is C. Consequently, C

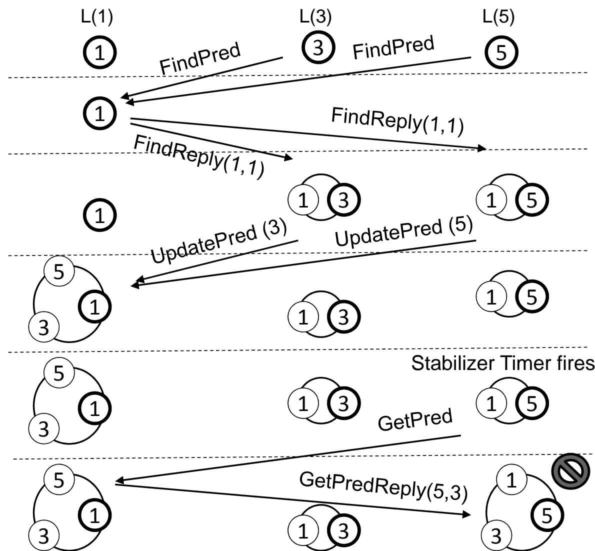

Figure 11: An inconsistency in a run of Chord. For node 5, its successor and predecessor do not obey in ordering constraint.

has its predecessor pointing to itself while its successor list includes other nodes.

Consequence of the inconsistency. Services implemented on top of distributed hash tables rely on its ability to route to any system participant. An incorrect successor can therefore disrupt the connectivity of the entire system by disconnecting the Chord ring.

Possible corrections. One possibility is for nodes to avoid sending UpdatePred messages to themselves (this appears to be a deliberate coding style in Mace Chord source code). If we wish to preserve such coding style, we can alternatively place a check after updating a node's predecessor: if the successor list includes nodes in addition to itself, avoid assigning the predecessor pointer to itself.

Node Ordering Constraint. According to Chord specification, a node's predecessor pointer contains the Chord identifier of the immediate predecessor of that node. Therefore, if a node A has a predecessor P and one of its successor is S, then the id of S should *not* be between the id of P and the id of A.

Scenario exhibiting inconsistency. CrystalBall found a safety violation where node A_{i-1} adds a new successor A_{i-2} to its successor list while its predecessor pointer is set to A_i and id of A_{i-2} is between the id of A_{i-1} and A_i. The scenario discovered is as follows (Figure 11). The id of A_i is less than the id of A_j where $i < j$. During live execution, node A_i joins the ring. Nodes A_{i-1} and A_{i-2} both try to join A_i by sending FindPred messages to it. Node A_i sends two FindPredReply back to A_{i-1} and A_{i-2} with exactly the same information. Upon receipt of this message, nodes A_{i-1} and A_{i-2} set their predecessor and successor to A_i and send UpdatePred message back to A_i. Finally, Node A_i sets its predecessor to A_{i-1} and successor to A_{i-2}.

In this state, consequence prediction discovers the following subsequent actions. Stabilizer timer of A_{i-1} fires and this node queries A_i by sending GetPred message. Node A_i replies back to A_{i-1} with a GetPredReply message that shows A_i's predecessor to be A_{i-1} and its successor list to contain A_{i-2}. Upon receiving this message, A_{i-1} adds A_{i-2} to its successor list while its predecessor pointer still points to A_i.

Possible correction. The bug occurs because node A_{i-1} adds information to its successor list but does not update its predecessor list. The bug could be fixed by updating the predecessor after updating the successor list.

5.2.3 Example Bullet′ Bug Found

Next, we describe our experience of applying CrystalBall to the Bullet′ [23] file distribution system. The Bullet′source sends the blocks of the file to a subset of nodes in the system; other nodes discover and retrieve these blocks by explicitly requesting them. Every node keeps a file map that describes blocks that it currently has. A node participates in the discovery protocol driven by RandTree, and peers with other nodes that have the most disjoint data to offer to it. These peering relationships form the overlay mesh.

Bullet′ is more complex than RandTree, Chord (and tree-based overlay multicast protocols) because of 1) the need for senders to keep their receivers up-to-date with file map information, 2) the block request logic at the receiver, and 3) the finely-tuned mechanisms for achieving high throughput under dynamic conditions. The starting point for our exploration was property 1):

Sender's file map and receivers view of it should be identical. Every sender keeps a "shadow" file map for each receiver telling it which are the blocks it has not told the receiver about. Similarly, a receiver keeps a file map that describes the blocks available at the sender. Senders use the shadow file map to compute "diffs" on-demand for receivers containing information about blocks that are "new" relative to the last diff.

Senders and receivers communicate over non-blocking TCP sockets that are under control of MaceTcpTransport. This transport queues data on top of the TCP socket buffer, and refuses new data when its buffer is full.

Scenario exhibiting inconsistency: In a live run lasting less than three minutes, CrystalBall quickly identified a mismatch between a sender's file map and the receiver's view of it. The problem occurs when the diff cannot be accepted by the underlying transport. The code then clears the receiver's shadow file map, which means that the sender will never try again to inform the receiver about the blocks containing that diff. Interest-

ingly enough, this bug existed in the original MACEDON implementation, but there was an attempt to fix it by the UCSD researchers working on Mace. The attempted fix consisted of retrying later on to send a diff to the receiver. Unfortunately, since the programmer left the code for clearing the shadow file map after a failed send, all subsequent diff computations will miss the affected blocks.

Consequence of the inconsistency. Having some receivers not learn about certain blocks can cause incomplete downloads because of the missing blocks (nodes cannot request blocks that they do not know about). Even when a node can learn about a block from multiple senders, this bug can also cause performance problems because the request logic uses a rarest-random policy to decide which block to request next. Incorrect file maps can skew the request decision toward blocks that are more popular and would normally need to be retrieved later during the download.

Possible corrections. Once the inconsistency is identified, the fix for the bug is easy and involves not clearing the sender's file map for the given receiver when a message cannot be queued in the underlying transport. The next successful enqueuing of the diff will then correctly include the block info.

5.3 Comparison with MaceMC

To establish the baseline for model checking performance and effectiveness, we installed our safety properties in the original version of MaceMC [22]. We then ran it for the three distributed services for which we identified safety violations. After 17 hours, exhaustive search did not identify any of the violations caught by CrystalBall. Some of the specific depths reached by the model checker are as follows 1) RandTree with 5 nodes: 12 levels, 2) RandTree with 100 nodes: 1 level, 3) Chord with 5 nodes: 14 levels, and Chord with 100 nodes: 2 levels. Figure 12 illustrates the performance of MaceMC when is used for exhaustive search. As depicted in figure, the exponential growth of elapsed time in terms of search depth hardly lets it search deeper than 12-13 steps. In another experiment, we additionally employed random walk feature of MaceMC. Using this setup, MaceMC identified some of the bugs found by CrystalBall, but it still failed to identify 2 Randtree, 2 Chord, and 3 Bullet′ bugs found by CrystalBall. In Bullet′, MaceMC found no bugs despite the fact that the search lasted 32 hours. Moreover, even for the bugs found, the long list of events that lead to a violation (on the order of hundreds) made it difficult for the programmer to identify the error (we spent five hours tracing one of the violations involving 30 steps). Such a long event list is unsuitable for execution steering, because it describes a low probability way of reaching the final erroneous state. In contrast, CrystalBall identified violations that are close to live executions and therefore more likely to occur in the immediate future.

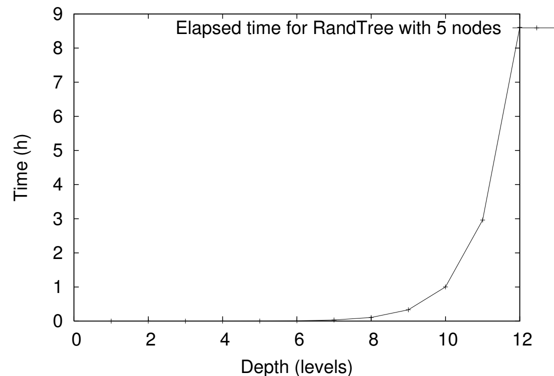

Figure 12: MaceMC performance: the elapsed time for exhaustively searching in RandTree state space.

5.4 Execution Steering Experience

We next evaluate the capability of CrystalBall as a runtime mechanism for steering execution away from previously unknown bugs.

5.4.1 RandTree Execution Steering

To estimate the impact of execution steering on deployed systems, we instructed the CrystalBall controller to check for violations of RandTree safety properties (including the one described in Section 5.2.1). We ran a live churn scenario in which one participant (process in a cluster) per minute leaves and enters the system on average, with 25 tree nodes mapped onto 25 physical cluster machines. Every node was configured to run the model checker. The experiment ran for 1.4 hours and resulted in the following data points, which suggest that in practice the execution steering mechanism is not disruptive for the behavior of the system.

When CrystalBall is not active, the system goes through a total of 121 states that contain inconsistencies. When only the immediate safety check but not the consequence prediction is active, the immediate safety check engages 325 times, a number that is higher because blocking a problematic action causes further problematic actions to appear and be blocked successfully. Finally, we consider the run in which both execution steering and the immediate safety check (as a fallback) are active. Execution steering detects a future inconsistency 480 times, with 65 times concluding that changing the behavior is unhelpful and 415 times modifying the behavior of the system. The immediate safety check fallback engages 160 times. Through a combined action

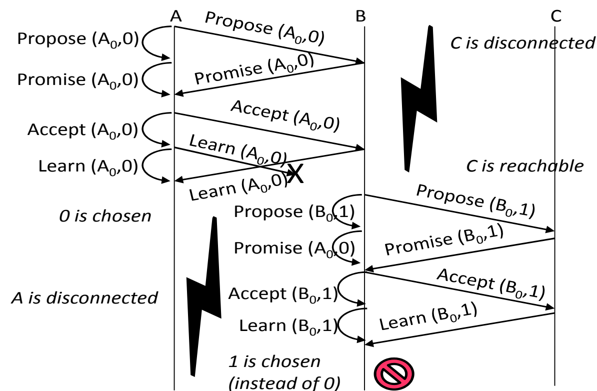

Figure 13: Scenario that exposes a previously reported Paxos violation of a safety property (two different values are chosen in the same round).

of execution steering and immediate safety check, CrystalBall avoided all inconsistencies, so there were no uncaught violations (false negatives) in this experiment.

To understand the impact of CrystalBall actions on the overall system behavior, we measured the time needed for nodes to join the tree. This allowed us to empirically address the concern that TCP reset and message blocking actions can in principle cause violations of liveness properties (in this case extending the time nodes need to join the tree). Our measurements indicated an average node join times between 0.8 and 0.9 seconds across different experiments, with variance exceeding any difference between the runs with and without CrystalBall. In summary, CrystalBall changed system actions 415 times (2.77% of the total of 14956 actions executed), avoided all specified inconsistencies, and did not degrade system performance.

5.4.2 Paxos Execution Steering

Paxos [26] is a well known fault-tolerant protocol for achieving consensus in distributed systems. Recently, it has been successfully integrated in a number of deployed [4, 28] and proposed [19] distributed systems. In this section, we show how execution steering can be applied to Paxos to steer away from realistic bugs that have occurred in existing implementations [4, 28].[1]

The implementation we used was a baseline Mace Paxos implementation that includes a minimal set of features. In general, a physical node can implement one or more of the roles described above; each node plays all the roles in our experiments. The safety property we installed is the original Paxos safety property: at most one value can be chosen, across all nodes. The first bug we injected [28] is related to an implementation error in step 3, and we refer to it as *bug1*. Once the leader receives the Promise message from the majority of nodes, it creates the Accept request by using the submitted value from the last Promise message instead of the Promise message with highest round number. Because the rate at which this error occurs was low, we had to schedule some events to lead the live run towards the violation. The setup we use comprises 3 nodes and two rounds, without any artificial packet delays. As illustrated in Figure 13, in the first round the communication between node C and the other nodes is broken. Also, a Learn packet is dropped from node 0 to 1. At the end of this round, A chooses the value proposed by itself (0). In the second round, the communication between node A and other nodes is broken. At the end of this round, the value proposed by node C is accepted by node B.

The second bug we injected (inspired by [4]) involves keeping a promise made by an Acceptor, even after crashes and reboots. As pointed in [4], it is often difficult to implement this aspect correctly, especially under various hardware failures. Hence, we inject an error in the way a promise is kept by not writing it to disk (we refer to it as *bug2*). To expose this bug we use a scenario similar to the one used for *bug1*, with the addition of a reset of node B.

To stress test CrystalBall's ability to avoid inconsistencies at runtime, we repeat the live scenarios in the ModelNet cluster 200 times (100 times for each bug) while varying the time between rounds uniformly at random between 0 and 60 seconds. As we can see in Figure 14, CrystalBall's execution steering is successful in avoiding the inconsistency at runtime 87% and 85% of the time for *bug1* and *bug2*, respectively. In these cases, CrystalBall starts model checking after node C reconnects and receives checkpoints from other participants. After running the model checker for 6 seconds, C successfully predicts that the scenario in the second round would result in violation of the safety property, and it then installs the event filter. The avoidance by execution steering happens when C rejects the Prepare message sent by B. Immediate safety check engages 11% of the time for

[1] The Paxos protocol includes five steps:

1. A leader tries to take the leadership position by sending Prepare messages to acceptors, and it includes a unique round number in the message.

2. Upon receiving a Prepare message, each acceptor consults the last promised round number. If the message's round number is greater than that number, the acceptor responds with a Promise message that contains the last accepted value if there is any.

3. Once the leader receives a Promise message from the majority of acceptors, it broadcasts an Accept request to all acceptors. This message contains the value of the Promise message with the highest round number, or is any value if the responses reported no proposals.

4. Upon the receipt of the Accept request, each acceptor accepts it by broadcasting a Learn message containing the Accepted value to the learners, unless it had made a promise to another leader in the meanwhile.

5. By receiving Learn messages from the majority of the nodes, a learner considers the reported value as chosen.

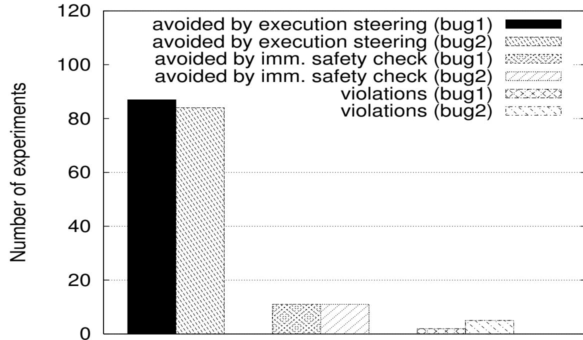

Figure 14: In 200 runs that expose Paxos safety violations due to two injected errors, CrystalBall successfully avoided the inconsistencies in all but 2 and 5 cases, respectively.

both bugs (in cases when model checking did not have enough time to uncover the inconsistency), and prevents the inconsistency from occurring later, by dropping the Learn message from C at node B. CrystalBall could not prevent the violation for only 2% and 5% of the runs, respectively. The cause for these false negatives was the incompleteness of the set of checkpoints.

5.5 Performance Impact of CrystalBall

Memory, CPU, and bandwidth consumption. Because consequence prediction runs in a separate process that is most likely mapped to a different CPU core on modern processors, we expect little impact on the service performance. In addition, since the model checker does not cache previously visited states (it only stores their hashes) the memory is unlikely to become a bottleneck between the model-checking CPU core and the rest of the system.

One concern with state exploration such as model-checking is the memory consumption. Figure 15 shows the consequence prediction memory footprint as a function of search depth for our RandTree experiments. As expected, the consumed memory increases exponentially with search depth. However, since the effective CrystalBall's search depth in is less than 7 or 8, the consumed memory by the search tree is less than 1MB and can thus easily fit in the L2 cache of the state of the art processors. Having the entire search tree in-cache reduces the access rate to memory and improves performance.

To precisely measure the consumed memory per each visited state by consequence prediction algorithm, we divided the total memory used by search tree by the number of visited states. As illustrated in the Figure 16, the per-state memory gets stable at about 150 bytes as we take more states into consideration, because of the reduced fixed cost in large number of states.

In the deep online debugging mode, the model checker

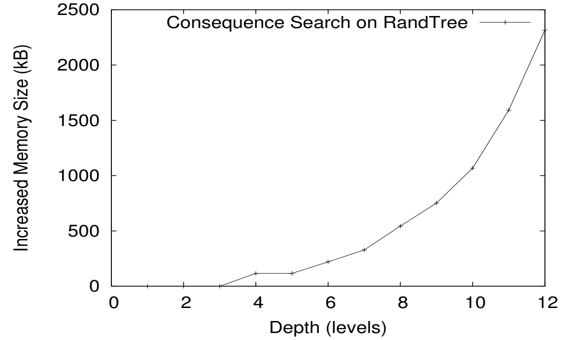

Figure 15: The memory consumed by consequence prediction (RandTree, depths 7 to 8) fits in an L2 CPU cache.

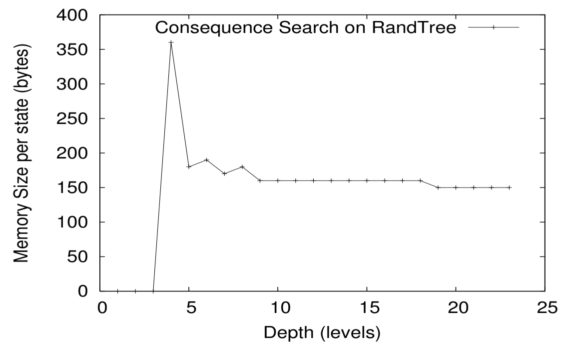

Figure 16: Consumed memory per each traversed state. The limit of this number is 150 bytes.

was running for 950 seconds on average in the 100-node case, and 253 seconds in the 6-node case. When running in the execution steering mode (25 nodes), the model checker ran for an average of about 10 seconds. The checkpointing interval was 10 seconds.

The average size of a RandTree node checkpoint is 176 bytes, while a Chord checkpoint requires 1028 bytes. Average per-node bandwidth consumed by checkpoints for RandTree and Chord (100-nodes) was 803 bps and 8224 bps, respectively. These figures show that overheads introduced by CrystalBall are low. Hence, we did not need to enforce any bandwidth limits in these cases.

Overall impact. Finally, we demonstrate that having CrystalBall monitor a bandwidth-intensive application featuring a non-negligible amount of state such as Bullet′ does not significantly impact the application's performance. In this experiment, we instructed 49 Bullet′ instances to download a 20 MB file. Bullet′ is not a CPU intensive application, although computing the next block to request from a sender has to be done quickly. It is therefore interesting to note that in 34 cases during this experiment the Bullet′ code was competing with the model checker for the older Xeon CPU with hyper-threading. Figure 17 shows that in this case using Crys-

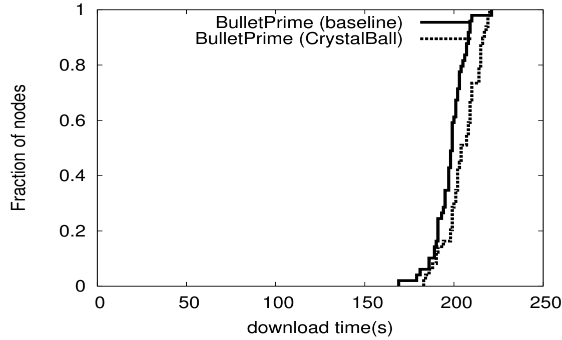

Figure 17: CrystalBall slows down Bullet′ by less than 10% for a 20 MB file download.

talBall reduced performance by less than 5%. Compressed Bullet′ checkpoints were about 3 kB in size, and the bandwidth that was used for checkpoints was about 30 Kbps per node (3% of a node's outbound bandwidth of 1 Mbps). The reduction in performance is therefore primarily due to checkpoints.

6 Related Work

Debugging distributed systems is a notoriously difficult and tedious process. Developers typically start by using an ad-hoc logging technique, coupled with strenuous rounds of writing custom scripts to identify problems. Several categories of approaches have gone further than the naive method, and we explain them in more detail in the remainder of this section.

Collecting and analyzing logs. Several approaches (Magpie [2], X-trace [13], Pip [34]) have successfully used extensive logging and off-line analysis to identify performance problems and correctness issues in distributed systems. Relative to these approaches, CrystalBall works on deployed systems, and performs an online analysis of the system state.

Deterministic replay with predicate checking. Friday [14] goes one step further than logging to enable a gdb-like replay of distributed systems, including watch points and checking for global predicates. WiDS-checker [28] is a similar system that relies on a combination of logging/checkpointing to replay recorded runs and check for user predicate violations. WiDS-checker can also work as a simulator. In contrast to replay- and simulation-based systems, CrystalBall explores additional states and can steer execution away from erroneous states.

Online predicate checking. Singh *et al.* [40] have advocated debugging by online checking of distributed system state. Their approach involves launching queries across the distributed system that is described and deployed using the OverLog/P2 [40] declarative language/runtime combination. D3S [27] enables developers to specify global predicates which are then automatically checked in a deployed distributed system. By using binary instrumentation, D3S can work with legacy systems. Specialized *checkers* perform predicate-checking topology on snapshots of the nodes' states. To make the snapshot collection scalable, the checker's *snapshot neighborhood* can be manually configured by the developer. This work has shown that it is feasible to collect snapshots at runtime and check them against a set of user-specified properties. CrystalBall advances the state-of-the-art in online debugging in two main directions: 1) it employs an efficient algorithm for model checking from a live state to search for bugs "deeper" and "wider", and it 2) enables execution steering to automatically prevent previously unidentified bugs from manifesting themselves in a deployed system.

Model checking. Model checking techniques for finite state systems [16, 20] have proved successful in analysis of concurrent finite state systems, but require the developer to manually abstract the system into a finite-state model which is accepted as the input to the system. Early efforts on explicit-state model checking of C and C++ implementations [31, 30, 44] have primarily concentrated on a single-node view of the system.

MaceMC [22] represents the state-of-the-art in model checking distributed system implementations. MaceMC runs state machines for multiple nodes within the same process, and can determine safety and liveness violations spanning multiple nodes. MaceMC's exhaustive state exploration algorithm limits in practice the search depth and the number of nodes that can be checked. In contrast, CrystalBall's consequence prediction allows it to achieve significantly shorter running times for similar depths, thus enabling it to be deployed at runtime. In [22] the authors acknowledge the usefulness of prefix-based search, where the execution starts from a given supplied state. Our work addresses the question of obtaining prefixes for prefix-based search: we propose to directly feed into the model checker states as they are encountered in live system execution. Using CrystalBall we found bugs in code that was previously debugged in MaceMC and that we were not able to reproduce using MaceMC's search. Unlike CrystalBall, MaceMC attempts to identify (likely) violations of liveness properties. On the other hand, MaceMC does not support execution steering that enables CrystalBall to automatically prevent the system from entering an erroneous state.

Cartesian abstraction [1] is a technique for overapproximating state space that treats different state components independently. The independence idea is also present in our consequence prediction, but, unlike overapproximating analyses, bugs identified by consequence search are guaranteed to be real with respect to the model

explored. The idea of disabling certain transitions in state-space exploration appears in partial-order reduction (POR) [15],[12]. Our initial investigation suggests that a POR algorithm takes considerably longer than the consequence prediction algorithm. The advantage of POR is its completeness, but completeness is of second-order importance in our case because no complete search can terminate in a reasonable amount of time for state spaces of distributed system implementations.

Runtime Mechanisms. In the context of operating systems, researchers have proposed mechanisms that safely re-execute code in a changed environment to avoid errors [33]. Such mechanisms become difficult to deploy in the context of distributed systems. Distributed transactions are a possible alternative to execution steering, but involve several rounds of communication and are inapplicable in environments such as wide-area networks. A more lightweight solution involves forming a FUSE [11] failure group among all nodes involved in a join process. Making such approaches feasible would require collecting snapshots of the system state, as in CrystalBall. Our execution steering approach reduces the amount of work for the developer because it does not require code modifications. Moreover, our experimental results show an acceptable computation and communication overhead.

In Vigilante [9] and Bouncer [8], end hosts cooperate to detect and inform each other about worms that exploit even previously unknown security holes. These systems deploy detectors that use a combination of symbolic execution and path slicing to detect infection attempts. Upon detecting an intrusion, the detector generates a Self-Certifying Alert (SCA) and broadcasts it quickly over an overlay in an attempt to win the propagation race against the worm that spreads via random probing. There are no false positives, since each host verifies every SCA in sandbox (virtual machine), after receiving it. After verification, hosts protect themselves by generating filters that block bad inputs.

Researchers have explored modifying actions of concurrent programs to reduce data races [18] by inserting locks in an approach that does not employ running static analysis at runtime. Approaches that modify state of a program at runtime include [10, 36]; these approaches enforce program invariants or memory consistency without computing consequences of changes to the state.

7 Conclusions

We presented a new approach for improving the reliability of distributed systems, where nodes predict and avoid inconsistencies before they occur, even if they have not manifested in any previous run. We believe that our approach is the first to give running distributed system nodes access to such information about their future. To make our approach feasible, we designed and implemented consequence prediction, an algorithm for selectively exploring future states of the system, and developed a technique for obtaining consistent information about the neighborhood of distributed system nodes. Our experiments suggest that the resulting system, CrystalBall, is effective in finding bugs that are difficult to detect by other means, and can steer execution away from inconsistencies at runtime.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS*, 2001.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[3] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth Content Distribution in Cooperative Environments. In *SOSP*, October 2003.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[6] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*, June 2002.

[7] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *Selected Areas in Communications, IEEE Journal on*, 20(8):1456–1471, Oct 2002.

[8] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, 2007.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *SOSP*, October 2005.

[10] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, 2003.

[11] J. Dunagan, N. J. A. Harvey, M. B. Jones, D. Kostić, M. Theimer, and A. Wolman. FUSE: Lightweight Guaranteed Distributed Failure Notification. In *OSDI*, 2004.

[12] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[13] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[14] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, 2007.

[15] P. Godefroid and P. Wolper. A partial approach to model checking. *Inf. Comput.*, 110(2):305–326, 1994.

[16] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[17] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *OSDI*, December 2008.

[18] M. U. Janjua and A. Mycroft. Automatic correction to safety violations. In *Thread Verification (TV06)*, 2006.

[19] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus Routing: The Internet as a Distributed System. In *NSDI*, San Francisco, April 2008.

[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS*, 1990.

[21] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007.

[22] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

[23] D. Kostić, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining High Bandwidth under Dynamic Network Conditions. In *USENIX ATC*, 2005.

[24] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subsets to Build Scalable Network Services. In *USITS*, 2003.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Com. of the ACM*, 21(7):558–565, 1978.

[26] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[27] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^3S: Debugging deployed distributed systems. In *NSDI*, 2008.

[28] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS checker: Combating bugs in distributed systems. In *NSDI*, 2007.

[29] D. Manivannan and M. Singhal. Asynchronous recovery without using vector timestamps. *J. Parallel Distrib. Comput.*, 62(12):1695–1728, 2002.

[30] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *NSDI*, 2004.

[31] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[32] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP*, 2005.

[33] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP*, 2005.

[34] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI*, 2006.

[35] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *SIGCOMM*, 2005.

[36] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.

[37] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, 2004.

[38] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *SOSP*, 2001.

[39] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[40] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. *SIGOPS Oper. Syst. Rev.*, 40(4):389–402, 2006.

[41] S. M. Srinivasan, S. K, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX ATC*, 2004.

[42] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[43] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*, December 2002.

[44] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.