Model Checking of Distributed Algorithm Implementations

THÈSE N^{O} 4858 (2011)

PRÉSENTÉE LE 18 MARS 2011

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES EN RÉSEAUX
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Maysam YABANDEH

acceptée sur proposition du jury:

Prof. R. Urbanke, président du jury
Prof. D. Kostic, Prof. R. Guerraoui, directeurs de thèse
Dr C. Cachin, rapporteur
Prof. V. Kuncak, rapporteur
Dr V. Quéma, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2011

Abstract

It is notoriously difficult to develop reliable, high-performance distributed systems that run over 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. 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.

Model checking or systematic state space exploration, which has been used for testing of centralized systems, is also not effective for testing of distributed applications. The aim of these techniques is to exhaustively explore all the reachable states and verify some user-specified invariants on them. Although effective for small software systems, for more complex systems such as distributed systems the exponential increase in number of explored states, manifests itself as a problem at the very early stages of search. This phenomenon, which is also known as exponential state space explosion problem, prevents the model checker from reaching the potentially erroneous states at deeper levels, in a realistic time frame.

This thesis proposes Dervish, a new approach in testing that makes use of a model checker in parallel with the running distributed system. Before the model checker performance gets hampered by the exponential explosion problem, the model checker restarts form the current live state of the system, instead of the initial state. The continuously running model checker at each node predicts the possible future inconsistencies, before they actually manifest. This approach, not only helps in testing by checking more relevant states that could occur in a real run, but also enables the application to steer the execution away from the predicted inconsistencies. We identified new bugs in mature Mace implementations of RandTree, Bullet′, Paxos, and Chord distributed systems. Furthermore, we show that if the bug is not corrected during system development, Dervish is effective in steering the execution away from the inconsistent states at runtime.

To be feasible in practice, the state exploration algorithm in Dervish should be efficient enough to explore some useful states in the period between each two restarts. Our default implementation of this approach benefits from a new search heuristic effective for distributed algorithms with short communications,

termed *consequence prediction*, which selectively explores future event chains of the system. For consensus algorithms, however, which are known to be one of the most complex of distributed algorithms, the exploration algorithms built upon principles of model checking centralized systems are not scalable enough to be installed in Dervish. Those approaches reduce the problem of model checking distributed systems to that of centralized systems, by using the global state, which also includes the network state, as the model checking state. This thesis introduces LMC, a novel model checking algorithm designed specifically for distributed algorithms. The key insight in LMC is to treat the local nodes' states separately, instead of keeping track of the global states.

We show how Dervish equipped with LMC enables us to find bugs in some complex consensus algorithms, including PaxosInside, the first consensus algorithm proposed and implemented for manycore environments. A modern manycore architecture can be viewed as a distributed system with explicit message passing to communicate between cores. Yet, doing this efficiently is very challenging given the non-uniform latency in inter-core communication and the unpredicted core response time. This thesis explores, for the first time, the feasibility of implementing a (non-blocking) *consensus* algorithm in a manycore system. We present PaxosInside, a new *consensus* algorithm that takes up the challenges of manycore environments, such as limited bandwidth of interconnect network as well as the consensus *leader*. A unique characteristic of PaxosInside is the use of a single *acceptor* role in steady state, which in our context, significantly reduces the number of exchanged messages between replicas.

Keywords Distributed systems, testing, execution steering, inconsistency prediction, consensus algorithms, manycore systems, consequence prediction, reliability, execution steering, enforcing invariants.

Résumé

Il est notoirement difficile de développer des systèmes distribués fiables, de haute performance, qui s'exécutent sur des réseaux asynchrones. Même si un système distribué est basé sur un algorithme bien compris, son implantation peut contenir des erreurs découlant de la complexité des environnements distribués réalistes ou tout simplement des erreurs de codage. Bon nombre de ces erreurs ne peut se manifester qu'après que le système ait fonctionné pendant longtemps, a mis au point une topologie complexe, et a connu une séquence particulière d'événements de faible probabilité.

Les techniques de model-checking qui ont été utilisées pour tester des systèmes centralisés ne sont pas efficaces pour des applications distribuées. Le but de ces techniques est d'explorer de manière exhaustive tous les états accessibles par l'algorithme et de vérifier certains invariants spécifiés par l'utilisateur. Bien que l'efficacité de ces techniques est avérée sur des systèmes de petits logiciels, l'augmentation exponentielle du nombre d'états explorés se manifeste comme un problème à un stade très précoce de la recherche dans un système distribués. Ce problème empêche le vérificateur deur d'atteindre des états potentiellement erronées à des niveaux plus profonds, dans un laps de temps réaliste.

Cette thèse propose Dervish, une nouvelle approche qui consiste à exécuter un vérificateur de modèle en parallèle avec le fonctionnement du système distribué. Avant que le vérificateur de la performance du modèle ne se heurte au problème d'explosion exponentielle, le redémarrage de modèle de formulaire de vérificateur se fait à partir de l'état actuel en direct du système, au lieu de l'état initial. Le fonctionnement continu du vérificateur de modèle à chaque noeud permet de prédire d'éventuelles incohérences, avant qu'elles ne manifestent finalement. Cette approche, permet non seulement dans les tests de vérifier les états les plus pertinents qui pourraient se produire dans une exécution réelle, mais permet également à l'application de diriger l'exécution loin des incohérences prédites.

Nous avons identifié de nouveaux bogues dans les implantations de protocoles connus tels que Mace de RandTree, Bullet′, Paxos, et Chord. En outre, nous montrons que si le bogue n'est pas corrigé au cours du développement du système, Dervish est efficace dans le pilotage de l'exécution au détriment des états incompatibles de l'exécution.

Pour être réalisable dans la pratique, l'algorithme d'exploration d'état dans

Dervish doit être suffisamment efficace pour explorer certains états utiles dans la période qui sépare deux redémarrages. Notre implantation par défaut de cette approche bénéficie d'une nouvelle recherche heuristique efficace pour les algorithmes distribués avec des communications brèves qui explorent de façon sélective les chaines s d'événements futurs du système.

Pour des algorithmes de consensus, cependant, qui sont connus pour être des plus complexes en algorithmique distribuée, l'exploration des algorithmes construits sur des principes de vérification de modèles de systèmes centralisés ne sont pas suffisamment adaptables pour être installés dans Dervish. Dans ce but, cette thèse introduit LMC, un nouveau modèle de contrôle algorithme conçu spécifiquement pour les algorithmes distribués. La clé de LMC est de traiter les états des noeuds locaux séparément, au lieu de garder la trace des états globaux.

Nous montrons comment Dervish équipé de LMC nous permet de trouver des bogues dans certains algorithmes de consensus complexes, y compris PaxosInside, un algorithme de consensus que nous avons proposé et mis en oeuvre pour les environnements multicoeurs.

Une architecture multicoeur moderne peut être considérée comme un système distribué avec des messages explicitement échangés entre les coeurs. Assurer la cohérence des données dupliquées sur les coeurs nn'est pas une tâche facile du fait de la latence non uniforme des communications.

Cette thèse explore, pour la première fois, la possibilité de mettre en ouvre un algorithme(non bloquant) de *consensus* dans un système multicoeur. Nous présentons PaxosInside, un nouvel algorithme de *consensus* qui relève les défis des environnements multicoeurs, comme la bande passante limitée du réseau d'interconnexion ainsi que le problème de l'élection dd'un *leader*. Une caractéristique unique de PaxosInside est l'utilisation d'un *acceptor* unique dans l'état d'équilibre, ce qui dans notre contexte, réduit considérablement le nombre de messages échangés entre les replicas.

Mots-clés Systèmes distribués, contrôle réparti, duplication, vérification, consensus, système multicoeur, invariants.

This thesis is dedicated to my parents.

Acknowledgments

First, I would like to thank my supervisor, Prof. Rachid Guerraoui, not only for all his endless supports and advices, but also, more importantly, for the freedom atmosphere in his laboratory, which made research be a joyful, pleasant experience.

I would like to say a special thank you to Prof. Viktor Kuncak, whose manner of life taught me modesty, generosity, honesty, and dignity. He had developed the notion of *execution steering* as well as the main idea of the presented tool in this thesis, Dervish, which was running a model checker in parallel with the live system, and I am very grateful for the opportunity of working on such an interesting idea.

I will never be able to properly thank Prof. Willy Zwaenepoel, for his selfless, generous, continued supports. He kindly helped me in the toughest period of my PhD, and if it was not because of his advices and generous helps, I would not have been able to finish this PhD.

I also thank all the lab members and friends for the many useful discussions.

Preface

This thesis includes the PhD work done under the supervision of Prof. Rachid Guerraoui at the Distributed Programming Laboratory, School of Computer and Communication Sciences, EPFL, from 2008 to 2010. The work presented in this thesis is centered on consensus algorithms and techniques for testing them. In addition to the presented material, I also worked on independent faults in the cloud [GY10], new lower bounds for abortable Byzantine fault tolerant protocols, model checking tools for software implementations [Yab10], new programming model for distributed systems [YVKK09], dynamic partial order reduction for distributed systems [YK09], and testing distributed systems by detecting almost-invariants [YACK09].

The presented materials in this thesis are published in transaction on computer systems (TOCS) [YKKK10], the NSDI conference [YKKK09], and a LPD technical report [YFG10].

[YKKK10] *Predicting and preventing inconsistencies in deployed distributed systems.* M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. In *ACM Transactions on Computer Systems (TOCS)*, 28(1):1–49, 2010.

[YKKK09] *CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems.* M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. In *NSDI*, 2009.

[YFG10] *One Acceptor is Enough.* M. Yabandeh, L. Franco, and R. Guerraoui. Technical Report LPD-REPORT-2010-01, EPFL, January 2010.

Table of Contents

Abstract iii

Résumé v

Acknowledgments ix

Preface xi

1 Introduction 1
- 1.1 Contributions . . . 3
- 1.2 Example . . . 4
- 1.3 Dervish Overview . . . 7
- 1.4 Thesis Organization . . . 10

2 Fundamental Concepts & Existing Approaches 11
- 2.1 System Model . . . 11
- 2.2 Model-Checking Distributed Systems . . . 12
- 2.3 Consistent Global Snapshots . . . 14
- 2.4 Related Work . . . 15
  - 2.4.1 Testing Distributed Systems . . . 16
  - 2.4.2 Runtime Mechanisms . . . 18

3 Dervish Design & Implementation 21
- 3.1 Consistent Neighborhood Snapshots . . . 22
  - 3.1.1 Discovering and Managing Snapshot Neighborhoods . . . 23
  - 3.1.2 Enforcing Snapshot Consistency . . . 23
  - 3.1.3 Checkpoint Content . . . 23
  - 3.1.4 Managing Checkpoint Storage . . . 24
  - 3.1.5 Managing Bandwidth Consumption . . . 24
- 3.2 Consequence Prediction Algorithm . . . 25
  - 3.2.1 Exploring Chains . . . 26
- 3.3 Execution Steering . . . 28
  - 3.3.1 Event Filters . . . 28
  - 3.3.2 Granularity of Filters . . . 29
  - 3.3.3 Point of Intervention . . . 29

3.3.4 Non-Disruptiveness of Execution Steering 30
3.3.5 Rechecking Previously Discovered Violations 30
3.3.6 Immediate Safety Check 31
3.3.7 Liveness Properties 31
3.4 Scope of Applicability 31
3.5 Implementation 32
3.5.1 Checkpoint Manager 33
3.5.2 Consequence Prediction 33
3.5.3 Immediate safety check 34
3.5.4 Replaying Past Erroneous Paths 35
3.5.5 Event Filtering for Execution Steering 35
3.5.6 Checking Safety of Event Filters 36
3.6 Evaluation 36
3.6.1 Experimental Setup 37
3.6.2 Deep Online testing Experience 37
3.6.3 Comparison with MaceMC 46
3.6.4 Execution Steering Experience 47
3.7 Summary 55

4 LMC: Local Model Checking **57**
4.1 Local Model Checking: A Primer 60
4.2 Design 62
4.2.1 LMC Algorithm 63
4.2.2 Implementation Details 67
4.2.3 Scope of Applicability 70
4.3 Evaluation 71
4.3.1 LMC Speedup 72
4.3.2 LMC Scalability Limits 74
4.3.3 LMC Memory Requirements 74
4.3.4 LMC Overheads 75
4.3.5 Testing Paxos 76
4.3.6 Testing PaxosInside 77
4.4 Related Work 78
4.5 Summary 80

5 PaxosInside **81**
5.1 PaxosInside: The Main Insight 84
5.2 Preliminaries 86
5.2.1 Manycore Systems 86
5.2.2 Blocking Agreement 87
5.2.3 Consensus 88
5.2.4 Failure Model 93
5.3 PaxosInside: The Algorithm 93
5.3.1 The Failure-free Case 93

5.3.2 Switching Acceptor . . . 94
5.3.3 Switching Leader . . . 95
5.3.4 Switching both Leader and Acceptor . . . 96
5.4 PaxosInside: A Multicore Implementation . . . 97
5.4.1 Message Queuing . . . 98
5.4.2 Message Delivery . . . 98
5.4.3 Agreement . . . 99
5.4.4 Description of C++ Code . . . 100
5.4.5 Persistent Storage in Multi-Paxos . . . 100
5.5 Evaluation . . . 100
5.5.1 Experimental Setup . . . 101
5.5.2 Workload . . . 101
5.5.3 Micro-benchmarks . . . 101
5.5.4 Scalability . . . 103
5.5.5 Throughput in Failure Scenarios . . . 103
5.6 Related Work . . . 105
5.7 Summary . . . 107

6 Conclusion **109**
6.1 Summary of Results . . . 110
6.2 Future Work . . . 111
6.2.1 On Symbolic Execution . . . 111
6.2.2 On Incremental Model Checking . . . 111
6.2.3 On Collaborative State Exploration . . . 111
6.2.4 On Collaborative Filter Installation . . . 111
6.2.5 On Model Checking Heuristics . . . 112
6.2.6 On Invariants . . . 112
6.2.7 On Model Checking Distributed Algorithms . . . 112
6.2.8 On PaxosInside . . . 112

A Example Run of Consequence Prediction on a Small Service **113**

B PaxosInside: Pseudo Code **117**

C PaxosInside: Proof of Correctness **121**

D LMC: Soundness Verification Proofs **125**

Bibliography **127**

List of Figures, Algorithms, and Tables **135**

About the Author **141**

After all, it could only cost you your life, and you got that for free!

Tortian Sailor, EarthBound Game

1 Introduction

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 [JKBK+08] to improve its routing reliability by further treating routing as a distributed consensus problem [Lam98]. 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 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 [LGW+08]. 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 the increases in computing power and bandwidth to make it easier to find errors in distributed systems, and to enhance 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 invariants. As Figure 1.1 illustrates, the ability to

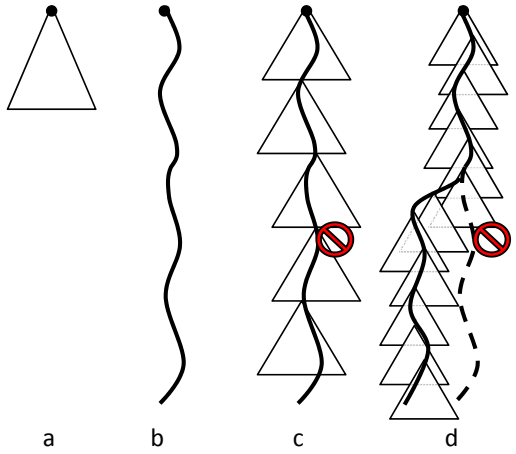

Figure 1.1: Execution path coverage by a) classic model checking, b) replay-based or live predicate checking, c) Dervish in deep online testing mode, and d) Dervish in execution steering mode. A triangle represents the state space searched by the model checker; a full line denotes an execution path of the system; a dashed line denotes an avoided execution path that would lead to an inconsistency.

detect future inconsistencies allows us to address the problem of reliability in distributed systems on two fronts: testing and resilience.

- Our technique enables deep online testing because it explores more states than live runs alone or more relevant states than model checking from the initial state. 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 testing 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: the model checker should search till a reasonable depth in the period between each two restarts. The exponential explosion in the state space size, however, does not allow the classic global model checking algorithms to explore further than a few steps in a limited time budget. To this aim, we propose *consequence prediction*, a heuristic for model checking distributed systems with short communications. Using this approach, we identified bugs in Mace implementations of a random overlay tree, and the Chord distributed hash table [SMLN$^+$03]. These implementations were previously manually tested as well as model-checked by exhaustive state exploration starting from the initial system state.

For more complex distributed systems, where lots of messages are triggered after an event, we propose LMC, a new local model checking algorithm that optimistically excludes the changes into the network state from the performed transitions. Using the proposed model checking algorithms in parallel with the running distributed system, 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 the predicted erroneous states, without significantly degrading the performance of the distributed service. We then instrument Dervish with LMC to test PaxosInside, a new *consensus* algorithm that takes up the challenges of manycore environments, such as limited bandwidth of interconnect network as well as the consensus *leader*. Thanks to quick exploration by LMC as well as deep exploration by Dervish, we managed to find bugs in PaxosInside, which is an example of complex distributed algorithms.

1.1 Contributions

We summarize the contributions of this thesis 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 present execution steering, a technique that enables the system to steer execution away from the predicted inconsistencies;
- We describe Dervish [YKKK09], the implementation of our approach on top of the Mace framework [KAB+07]. We evaluate Dervish on RandTree [KAJV07], Bullet′ [KBK+05], Paxos [Lam98], and Chord distributed system implementations. Dervish detected several previously unknown bugs that could cause system nodes to reach inconsistent states. Moreover, in the case of remaining bugs, Dervish's execution steering predicts them in the deployed system and steers execution away from them, all with an acceptable impact on the overall system performance.
- We introduce LMC, a novel approach in model checking distributed algorithms. LMC separates the network state from the global state and focuses on the remaining system states, which is the required part for invariant checking. We show how this approach enables us to find bugs in PaxosInside.
- We explore, for the first time, the feasibility of implementing a (non-blocking) *consensus* algorithm in a manycore system. We present PaxosInside, a new *consensus* algorithm that takes up the challenges of manycore

environments, such as limited bandwidth of interconnect network as well as the consensus *leader*.

1.2 Example

We next describe an example of an inconsistency exhibited by a distributed system, then later we show how Dervish predicts and avoids it. The inconsistency appears in the Mace [KAB+07] implementation of the RandTree overlay. RandTree implements a random, degree-constrained overlay tree designed to be resilient to node failures and network partitions. The 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 [KBK+05] and RanSub [KRA+03]. In general, trees are used in a variety of multicast scenarios [CDK+03, CRSZ02] and data collection/monitoring environments [JMK+08]. Inconsistencies in these environments translate into denial of service to the 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 [KAJV07] but, to our knowledge, this inconsistency has not been discovered before (see Section 3.6 for some additional inconsistencies that Dervish 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. The node with the numerically smallest IP address acts as the root of the tree. Each non-root node contains the address of its parent. Children of the root maintain a sibling list. Note that, for a given node, its parent, children, and siblings are all distinct nodes. The seemingly simple task of maintaining a consistent tree topology turns out to be complicated 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 message has not been forwarded by the root, then the receiver forwards the request back 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 request reaches a node n_p that its 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 1.2 shows a state of the system that we encountered by running RandTree in the ModelNet cluster [VYW+02] starting from the initial state. We examine the local states of

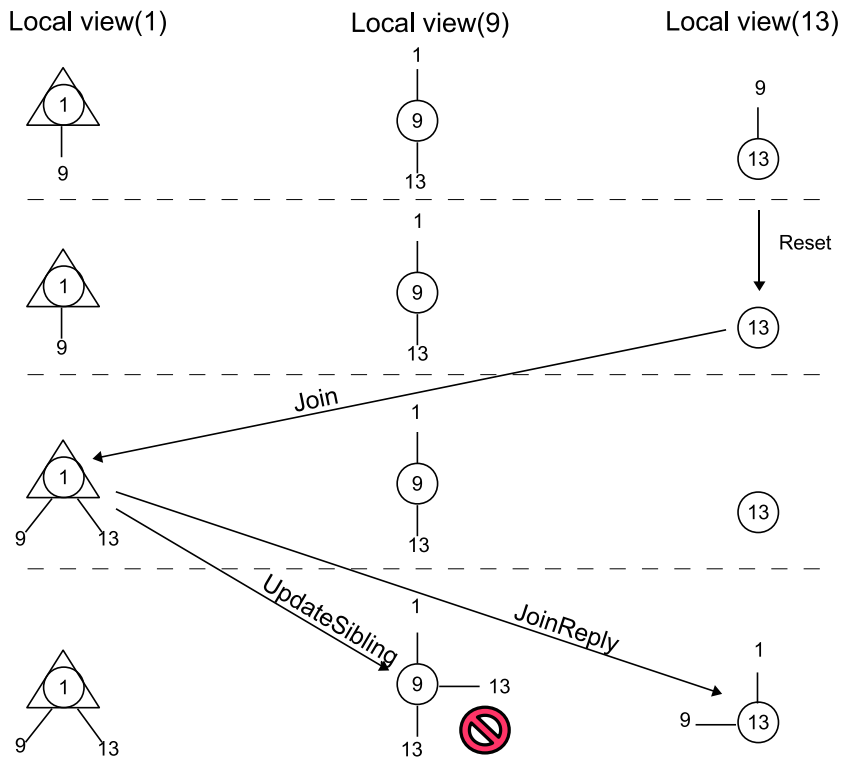

Figure 1.2: An inconsistency in a run of RandTree; Safety property: children and siblings are disjoint lists

nodes n_1, n_9, and n_{13}. For each node n_j, 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 1.2 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 1.2 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 1.2 describes a sequence of actions that leads to the 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, node n_1, accepts the join request and adds n_{13} as its child. Up to this point node n_9 has received no information on actions that followed the reset of n_{13}, thus 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 its sibling list. 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 1.2. 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 [KAJV07] and reported a number of safety and liveness bugs in Mace implementations. However, due to an exponential explosion of state space, 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 1.2. For example, when we applied the Mace model checker's [KAJV07] exhaustive search to the safety properties [1] of RandTree starting from the initial state, it failed to identify the inconsistency in Figure 1.2 even after running for 17 hours (on a 3.4-GHz Pentium-4 Xeon that we used for some of our experiments in Section 3.6). 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] In this thesis, we use the terms of "safety property" and "invariant" interchangeably.

1.3 Dervish 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 1.2, this state is fed to the model checker as initial state and the model checker will predict the state at the end of Figure 1.2 as a possible future inconsistency. In summary, instead of focusing only on 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 1.1 suggests, compared to the standard model checking approach, this approach identifies inconsistencies that can occur within much longer system executions. Compared to simply checking the live state of the running system, 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 1.2. 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 testing 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 is possible because the model checker can simulate packet transmission in time shorter than propagation latency, and because it can simulate timer events in time shorter than the actual time delays. This aspect of our approach opens an entirely new possibility: adapt the behavior of the running system on the fly and avoid a predicted 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.

Example of Execution Steering. In our example, a model checking algorithm running in n_1 detects the violation at the end of Figure 1.2. 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 1.3 presents one scenario illustrating this alternate execution sequence. Effectively, execution steering has exploited

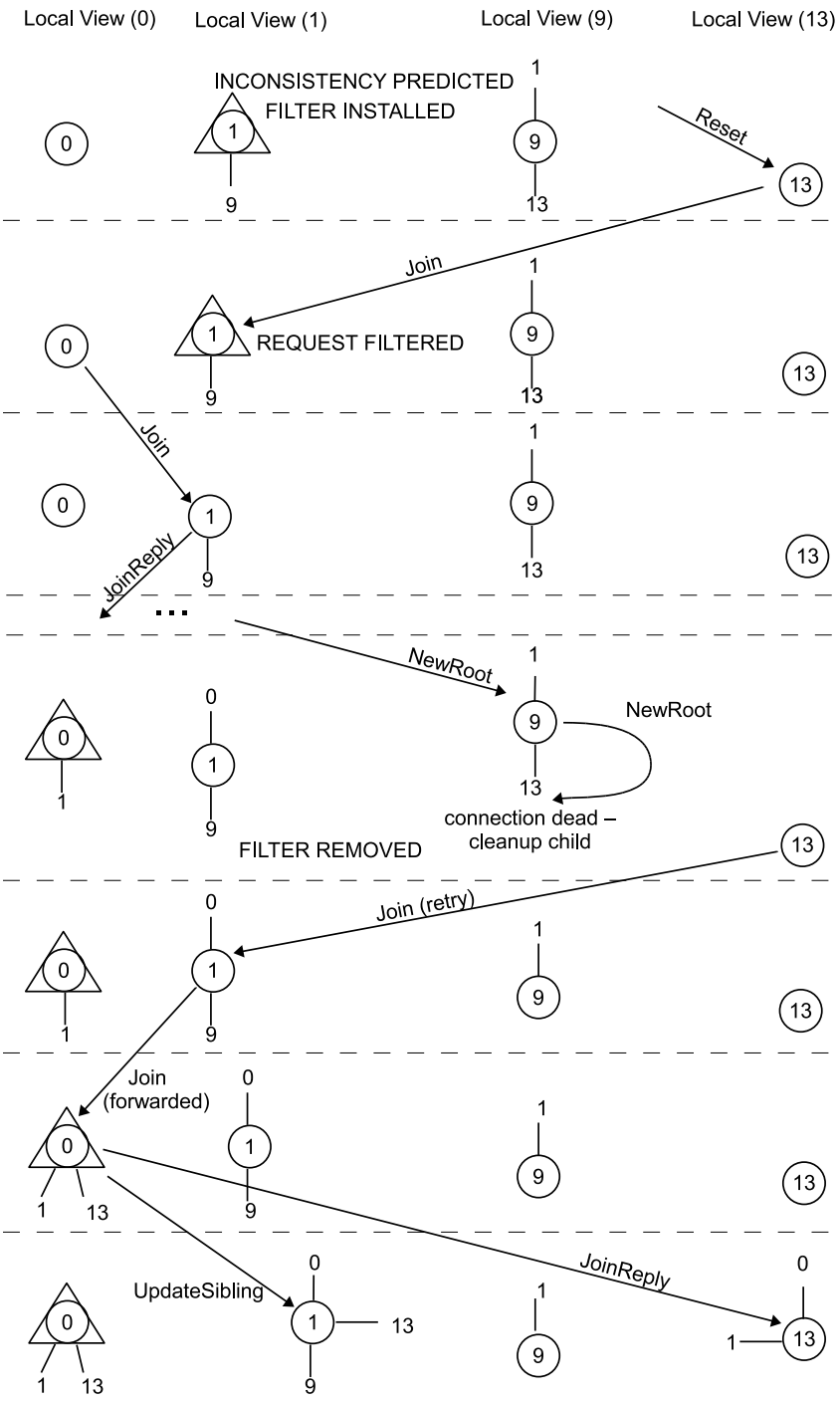

Figure 1.3: An example execution sequence that, thanks to execution steering, avoids the inconsistency from Figure 1.2.

the non-determinism and robustness of the system to choose an alternative execution path that does not contain the inconsistency.[2]

Fast Model Checking. 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. To make this approach feasible, it is essential to have a model checking technique capable of quickly discovering potential inconsistencies till significant depths in a very short amount of time. The classic global 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 faster than the time that it takes node interaction to cause the inconsistency in a distributed system. We present two such exploration techniques: *consequence prediction* and LMC.

Consequence Prediction. Consequence prediction heuristically filters some non-network events. Our system identifies the scenario in Figure 1.2 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.

LMC. Consequence prediction focuses on pruning non-network events, which turns out to be very efficient for algorithms with short communications. For example, each join request in a random overlay tree initiates a couple of network messages and then finishes by having the new node joined under its new parent node. This is not the case for complex distributed algorithms such as consensus algorithms. In these algorithms, each client request results into 10∼20 protocol messages, which makes heuristics such as consequence prediction less effective.

For complex algorithms with long event chains, the exploration algorithms built upon principles of model checking centralized systems are not scalable enough to be installed in Dervish. Those approaches reduce the problem of model checking distributed systems to that of centralized systems, by using the global state as the model checking state. The frequent changes into the

[2]Besides external events, such as additional node joining, a node typically has sufficient amount of choice to progress (e.g., multiple bootstrap nodes). In general however, it is possible that enforcing safety can reduce liveness.

network state are a major contributor to the exponential increase in number of global states. To tackle the challenges of testing complex distributed algorithms, we introduce LMC, a novel approach in model checking especially designed for distributed algorithms. LMC separates the network state from the global state and focuses on the remaining system states, which is the required part for invariant checking. Besides, instead of keeping track of the system states, LMC keeps the traversed local states by each node separately; the system states are created temporarily only to be verified against invariants.

PaxosInside. We then instrument Dervish with LMC to test PaxosInside, a new *consensus* algorithm that takes up the challenges of manycore environments, such as limited bandwidth of interconnect network as well as the consensus *leader*. A unique characteristic of PaxosInside is the use of a single *acceptor* role in steady state, which in our context, significantly reduces the number of exchanged messages between replicas. Thanks to quick exploration by LMC as well as deep exploration by Dervish, we managed to find bugs in PaxosInside, which is an example of complex distributed algorithms.

1.4 Thesis Organization

The rest of the thesis is split into 5 chapters, organized as follows.

Chapter 2 presents a simple model for distributed systems, which is the basis for all the model checking algorithms presented in this thesis. It then reviews model checking algorithms as well as techniques for taking consistent global snapshots. It then gives an overview of the previous, related works in testing distributed systems.

Chapter 3 presents the design and implementation details of Dervish. It covers the techniques for taking consistent neighborhood snapshots, consequence prediction algorithm, and execution steering. At the end, it presents the evaluation results of testing some distributed systems with Dervish.

Chapter 4 explains our novel model checking algorithm, LMC, which is specifically designed for complex distributed algorithms. The evaluation results at the end of this chapter analyze the scalability of LMC and verifies its ability in finding bugs. The design and implementation of PaxosInside, which is an example of complex distributed algorithms that will be used in benchmarking, is covered in Chapter 5.

Finally, Chapter 6 concludes the thesis, by giving a summary of the results as well as possible future works.

One can safely bet that all publicly received ideas, all established conventions are mere foolishness, because these are convenient to the majority.

Nicolas Chamfort

2

Fundamental Concepts & Existing Approaches

This chapter reviews some fundamental concepts of distributed systems and existing approaches that tackle testing them. 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 2.1 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 algorithms, consequence prediction and LMC.

System state. The *global state* of the entire distributed system encompasses (1) the system state, i.e., local states of all nodes, 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 $L^n \in S$. A local state encompasses node-local information, such as explicit state variables of the distributed node implementation, the status of timers, and the state that determines application calls. A network state corresponds to the set of 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 the system runs the same state-machine implementation. The state machine has two kinds of handlers: (i) a message handler executes in response to a network message; (ii) 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'. An 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 the handling of an internal node action $a \in A$. (In both handlers, the fact that c is the empty set 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 state* (L, I) to another global state (L', I'). We denote this transition by $(L, I) \rightsquigarrow (L', I')$ and describe it in Figure 2.1 in terms of handlers H_M and H_A. (L_0 and I_0 in Figure 2.1 are subsets of L and I, respectively.) 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.

Observations. The following observations can be derived from the definitions of H_M and H_A in Figure 2.1: (i) Except the node in which the event is executed, the state of other nodes, i.e., L_0, is untouched. This implies that to execute an event on node n, we require only the local state of node n. (ii) To execute H_M with message m on node n, the only required part from the network state is tuple (n, m): the rest of the network state, i.e., I_0, is untouched. These observations indicate that the entire global state of the system is not required to execute a handler in the model checker.

2.2 Model-Checking Distributed Systems

Figure 2.2 presents a standard depth-first search (DFS) for tracking invariant violations in the transition system captured by relation $\rightsquigarrow$ of Figure 2.1. In practice, model checkers usually use bounded depth-first search (B-DFS), in which the maximum depth of the search is bounded and this bound is iteratively increased. The search starts from a given global state `firstState` which, in the standard approach, is the initial state of the system. By executing enabled handlers (H_M and H_A) on the traversed global states, Procedure `exploreState`, the search systematically explores reachable global states at larger and larger depths and checks whether the states satisfy the given `invariant` condition. In practice, the number of reachable states is very large and the search needs to

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 – internal node actions (timers, application calls)

global state : $(L, I) \in G, \quad G = 2^{N \times S} \times 2^{N \times M}$
system state (local nodes' 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 2.1: A simple model of a distributed system

be terminated upon reaching some bound such as running time or search depth (`StopCriterion`).

To avoid search loops, the algorithm requires a duplicate state detection mechanism. This is done by keeping a history of the traversed global states, `explored`, and checking this history for duplicate states before exploring any state by Procedure `exploreState`. For the sake of space efficiency, instead of the whole state, a hash of the global states is stored in Variable `explored`. Note that although the presented algorithm is recursive, DFS can also be implemented non-recursively.

Soundness. The algorithm presented in Figure 2.2 is sound in the sense that all violation reports in `error` could also occur in a real run of the system. In other words, there is no *false positive* in the reported bugs. Moreover, all traversed states in `explored` are valid and could also be created in a real run.

```
1 proc findErrors(firstState : G, invariant : (G → boolean)) {
2   explored = emptySet(); errors = emptySet();
3   state = firstState;
4   exploreState(state);
5 }
6
7 proc exploreState(state : G, invariant : (G → boolean)) {
8   explored = explored ∪ hash(nextState);
9   if (!invariant(state))
10    errors.add(state);
11  foreach (nextState where (state ⇝ nextState))
12    if (!StopCriterion && (hash(nextState) ∉ explored))
13      exploreState(nextState);
14 }
```

Figure 2.2: The classic DFS-based (Depth First Search) algorithm to model checking distributed systems.

The sufficient part for soundness, however, is only the reported violations to the developer, i.e., `error`. We will show later that our local model checking is also sound, even though some system states created a priori might be invalid.

Completeness. An exploration algorithm is complete if, given enough time and space, it can explore all system states. In other words, completeness is satisfied if there is no *false negative*. Although DFS are both complete, due to an inherently limited time budget, in practice they can explore only a small fraction of the state space of complex algorithms.

2.3 Consistent Global Snapshots

Examining global state of a distributed system is useful in a variety of scenarios, such as checkpointing/recovery, testing, 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 [CL85]. We next describe one of the recent algorithms for obtaining consistent snapshots [MS02]. The general idea is to collect a set of checkpoints which do not violate the happens-before relationship [Lam78] 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 checkpoint number is similar to the Lamport's logical clock [Lam78].). 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

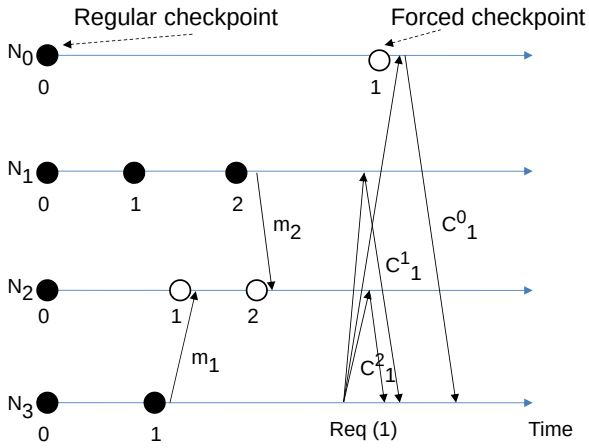

Figure 2.3: Example illustrating the consistent snapshot collection algorithm. Black ovals represent regular checkpoints. Messages m_1 and m_2 force checkpoints (white ovals) to be taken before messages are processed at nodes 2 and 1, respectively, and so does the checkpoint request from node 3 when it arrives at node 0.

$cn_j = M.cn$. 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 $C.cn \geq cr_i$. The example depicted in Figure 2.3, includes different scenarios that the algorithm forces taking checkpoints in order to maintain the checkpoints globally consistent.

2.4 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.

2.4.1 Testing Distributed Systems

Collecting and Analyzing Logs Several approaches (Magpie [BDIM04], X-trace [FPK+07], Pip [RKW+06]) have successfully used extensive logging and off-line analysis to identify performance problems and correctness issues in distributed systems. Unlike these approaches, Dervish works on deployed systems, and performs an online analysis of the system state.

Deterministic Replay with Predicate Checking Friday [GAM+07] 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 [LLPZ07] 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, Dervish explores additional states and can steer execution away from erroneous states.

Online Predicate Checking Singh *et al.* [SMRD06] have advocated testing 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 [SMRD06] declarative language/runtime combination. D^3S [LGW+08] and MaceODB [DAKV09] enable developers to specify global predicates which are then automatically checked in a deployed distributed system. By using binary instrumentation, D^3S 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. Dervish advances the state-of-the-art in online testing in two main directions: (1) it employs an efficient algorithm for model checking from a live state to search for bugs "deeper" and "wider" than in the live run, and (2) it 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 [Hol97, JEK+90] 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. Several systems have used sound abstraction techniques to verify sequential software implementation as opposed to its models [BR02, HJMS02, CCG+03]. Recently, techniques based on bounding the number of context switches in multi-threaded programs have been applied to single-node software systems [MQ07, MQB+08]. Bounding the number of context switches stands in contrast to consequence prediction, which follows chains of actions across any number of different nodes. Early efforts on explicit-state model checking of C and C++ implementations [MPC+02, ME04,

YTEM06] have primarily concentrated on a single-node view of the system. A continuation of this work, eXplode [YSE06], makes it easy to model-check complex, layered storage systems, in some cases involving a client and a server.

MODIST [YCW+09] and MaceMC [KAJV07] represent the state-of-the-art in model checking distributed system implementations. MODIST [YCW+09] is capable of model checking unmodified distributed systems; it orchestrates state space exploration across a cluster of machines. 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, Dervish's consequence prediction allows it to achieve significantly shorter running times for similar depths, thus enabling it to be deployed at runtime. In MaceMC [KAJV07] 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 Dervish we found bugs in code that was previously debugged in MaceMC and that we were not able to reproduce using MaceMC's search. In summary, Dervish differs from MODIST and MaceMC by being able to run state space exploration from live state. Further, Dervish supports execution steering that enables it to automatically prevent the system from entering an erroneous state.

Cartesian abstraction [BPR01] is a technique for over-approximating state space that treats different state components independently. The independence idea is also present in our consequence prediction but, unlike over-approximating analyses, bugs identified by consequence prediction 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) [GW94, FG05]. Our initial investigation suggests that a POR algorithm takes considerably longer than the consequence prediction algorithm.

To avoid loops created by exploring duplicate states, it is necessary to keep track of the visited states. Obtaining a hash of the system state requires touching the whole state once, which can be nontrivial in the case of large states. Although stateless approaches [God97] avoid this cost by not keeping track of traversed states, visiting duplicate states can make them very inefficient. Thanks to Mace [KAB+07] language, upon which we have implemented our tool, the relevant state of the protocol is specified by the developer and it is, hence, straightforward for MaceMC [KAJV07] to obtain its hash.

Partial Order Reduction Partial Order Reduction (POR) techniques [God96] prune the state space of a concurrent system to avoid unnecessary interleaving of events. If e_1 and e_2 are events on two separate nodes, executing $< s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 >$ and $< s_0 \xrightarrow{e_2} s'_1 \xrightarrow{e_1} s_2 >$ result in the same

global state, exploring only one of them is enough for checking the thread-local assert statements. In this case, e_1 and e_2 are called *independent*. Independence is not enough to prune e_2 from the state space graph because other events might be enabled at state s_1', leading the search towards states that are not reachable from s_1. To be able to prune e_2, one must first prove that e_1 and e_2 are in a *persistent* set at s_0.

Obtaining independent events and persistent sets requires static analysis of the source code. Static analysis tools might not be available in all programming languages; specifically if the operating system is included in the analysis, e.g., using communication objects in distributed algorithms. It is for instance not easy to establish the independence of two operations using different system calls. Furthermore, static analysis is not effective for dynamic data such as pointers [FG05]. This is because the value referenced by the pointer is not available at the time of analysis.

Dynamic Partial Order Reduction (DPOR) [FG05] was designed to address these limitations in multi-threaded programs. The DPOR algorithm computes the dependency during exploration, when the concrete values of the pointers are available. According to the observed dependencies, it adds appropriate branches to guarantee the completeness of the exploration. Although applicable to multi-threaded programs, DPOR is not applicable to distributed systems [YK09].

LMC and consequence prediction are designed for distributed systems, in which DPOR cannot be applied. To the best of our knowledge, the only recent adaptations of DPOR for distributed systems are DPOR-DS [YK09] and the work by Sen *et al.* [SA06a]. The performance of B-DFS that we used for benchmarking in this chapter, could potentially improve by implementing this techniques. However, we expect the improvement would be marginal because of frequent changes in the global state; transmitting any message would change the network state and consequently the global state. Moreover, lots of redundancies avoided by POR-based techniques are already avoided by duplicate state detection in LMC.

2.4.2 Runtime Mechanisms

In the context of operating systems, researchers have proposed mechanisms that safely re-execute code in a changed environment to avoid errors [QTZS07]. 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. Making such approaches feasible would require collecting snapshots of the system state, as in Dervish. Our execution steering approach reduces the amount of work for the developer because it does not re-

quire code modifications. Moreover, our experimental results show an acceptable computation and communication overhead.

In Vigilante [CCC+05] and Bouncer [CCZ+07], 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. Relative to these systems, Dervish deals with distributed system properties, and predicts inconsistencies before they occur.

Researchers have explored modifying actions of concurrent programs to reduce data races [JM06] by inserting locks in an approach that does not employ running static analysis at runtime. In another static approach to modifying program behavior [JGB05], the authors formalize the problem as a game. Approaches that modify state of a program at runtime include [DR03, RCD+04]; these approaches enforce program invariants or memory consistency without computing consequences of changes to the state.

A recent approach [WKK+08, WLK+09] first uses offline static analysis to construct a Petri net model of the multi-threaded application. Authors then apply Discrete Control Theory to identify potential deadlocks in the model, and synthesize control logic that enables the instrumented application to avoid deadlocks at runtime. This approach is applicable to multi-threaded applications and the particular property of avoiding deadlock. In contrast, Dervish is designed to avoid general safety properties in distributed systems.

Although from west to east is the army of darkness, Dervishes have time from the first moment of life till the eternity.

Hafez

3

Dervish Design & Implementation

We next sketch the design of Dervish. Figure 3.1 shows the high-level overview of a Dervish-enabled node. We concentrate on distributed systems implemented as state machines, as this is a widely-used approach [KAB+07, Lam78, Lam98, RKB+04, Sch90].

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 Dervish controller contains a checkpoint manager that periodically collects consistent snapshots of a node's neighborhood, including the local node. 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.

Dervish can operate in two modes. In the *deep online testing 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.

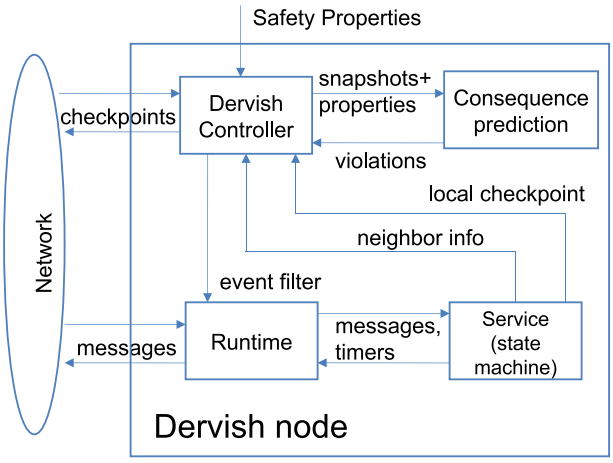

Figure 3.1: High-level overview of Dervish. This figure depicts the high-level architecture of Dervish and its main components.

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. 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 nodes 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 thus needs to perform model checking only on this neighborhood. In some distributed hash table implementations, a node keeps track of $O(\log n)$ other nodes; 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. Also in a random overlay tree, a node is typically aware of the root, its parent, its children, and its siblings. We therefore arrange for a node to distribute its state checkpoints to its neighbors, and we refer to received checkpoints as *snapshot neighborhood*. The *checkpoint manager* maintains checkpoints and snapshots. Other Dervish components can request an on-demand snapshot to be gathered by invoking an appropriate call on the checkpoint manager.

3.1.1 Discovering and Managing Snapshot Neighborhoods

To propagate checkpoints, the checkpoint manager needs to know the set of a node's neighbors. This set depends on the 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.

Because changing the service code might not always be possible, our second technique uses a heuristic to determine the snapshot neighborhood. Specifically, the heuristic periodically queries the runtime to obtain the list of open connections (for TCP), and recent message recipients (for UDP). It then clusters connection endpoints according to the communication times, and selects a sufficiently large cluster of recent connections.

3.1.2 Enforcing Snapshot Consistency

Dervish ensures that the neighborhood snapshot corresponds to a consistent view of the distributed system at some point of logical time. Our starting point is a technique similar to the one described in Section 2.3.

Instead of gathering a global snapshot, a node periodically sends a checkpoint request to the members of its snapshot neighborhood. Even though nodes receive checkpoints only from a subset of nodes, all distributed service and checkpointing messages are instrumented to carry the checkpoint number (logical clock) and each neighborhood snapshot is a fragment of a globally consistent snapshot. In particular, a node that receives a message with a logical timestamp greater than its own logical clock takes a forced checkpoint. The node then uses the forced checkpoint to contribute to the consistent snapshot when asked for it.

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 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.

3.1.3 Checkpoint Content

Although the total footprint of some services might be very large, this need not necessarily be reflected in checkpoint size. For example, the Bullet′ [KBK+05] file distribution application has non-negligible total footprint, but the actual file content transferred in Bullet′ does not play any role in consistency detection.

In general, the checkpoint content is given by a serialization routine. The developer can choose to omit certain parts of the state from serialized content and reconstruct them if needed at de-serialization time. As a result, checkpoints are smaller and the code compensates the lack of serialized state when a local state machine is being recreated from a remote node's checkpoint in the model checker.

3.1.4 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 the range of remaining 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.

3.1.5 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 bandwidth 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 transmitted, Dervish 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, Dervish 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

```
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 proc addNextState(state,n,s,s',c0,c)
23   nextState.L = (state.L \ {(n,s)}) ∪ {(n,s')};
24   nextState.I = (state.I \ c0) ∪ c;
25   if (!explored.contains(hash(nextState)))
26     frontier.addLast(nextState);
```

Figure 3.2: Consequence Prediction Algorithm

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 Dervish is our consequence prediction algorithm. The idea of the algorithm is to avoid exploring internal (local) actions of nodes whose local state was encountered previously at a smaller depth in the search tree. Recall that the state of the entire system contains the local states of each node in the neighborhood. A standard search avoids re-exploring actions of states whose particular combination of local states was encountered previously; it achieves this by storing hashes of the entire *global state*. We found this standard approach to be too slow for our purpose because of the high branching in the search tree (Figure 3.7 shows an example for RandTree.). Consequence prediction therefore additionally avoids exploring internal actions of a node if this node was already explored with the

same local state (regardless of the states of other nodes). Consequence prediction implements this policy by maintaining an additional hash table that stores hashes of visited *local states* for each node.

Figure 3.2 shows the consequence prediction pseudo code. In its overall structure, the algorithm is similar to the standard state-space search in Figure 2.2. (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 2.1.) In fact, if we omitted the test at Line 17,

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

the algorithm would reduce precisely to Figure 2.2.

In Line 8 of Figure 3.2, the algorithm checks whether the explored state satisfies the desired safety properties. To specify the properties, the developer can use a simple language [KAJV07] that supports universal and existential quantifiers, comparison operators, state variables, and function invocations.

3.2.1 Exploring Chains

To understand the intuition behind consequence search, we identify certain executions that consequence prediction does and does not explore. Consider the model of Figure 2.1 and the algorithm in Figure 3.2. We view local actions as triggering a sequence of message exchanges. Define an *event chain* as a sequence of steps $e_A e_{M1} \dots e_{Mp}$ (for $p \geq 0$) where the first element e_A is an internal node action (element of H_A in Figure 2.1, representing application or scheduler event), and where the subsequent elements e_{Mi} are network events (elements of H_M in Figure 2.1). Clearly, every finite execution sequence can be written as a concatenation of chains.

Executions explored: Consequence prediction explores all chains that start from the current state (i.e. the live state of the system). This is because (1) consequence search prunes only local actions, not network events, (2) a chain has only one local action, namely the first one, and (3) at the beginning of the search the `localExplored` hash tables are empty, so no pruning occurs.

Executions not necessarily explored: Consider a chain C of the form $e_A e_{M1} \dots e_{Mp}$ and another chain C' of the form $e'_A e'_{M1} \dots e'_{Mq}$, where e'_A is a local action applicable to a node n. Then consequence prediction will explore the concatenation CC' of these chains if there is no previously explored reachable state (at the depth less than p) at which the node n has the same state as after C. As a special case, suppose chains C and C' involve disjoint sets of nodes. Then consequence prediction will explore both C and C' in isolation.

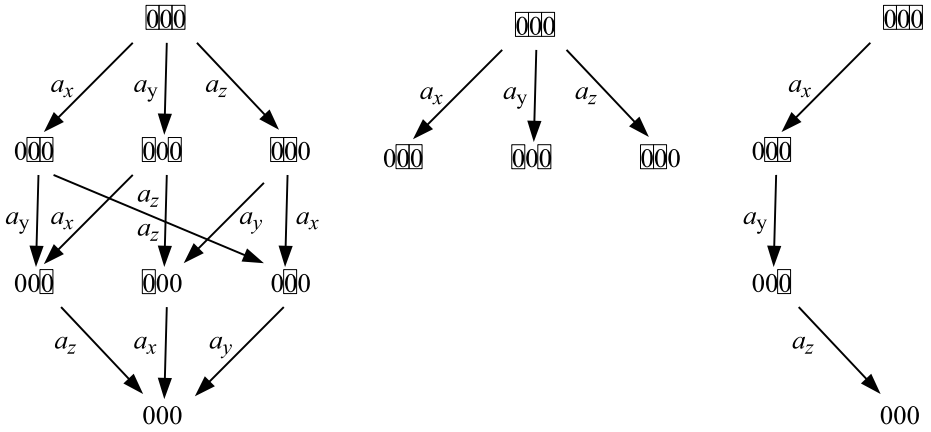

Figure 3.3: Full state space, consequence search, and partial order reduction in an example with internal actions of three distinct nodes

However, it will not explore the concatenation CC', nor a concatenation C_1C_1' where C_1 is a non-empty prefix of C and C_1' is non-empty prefix of C'.

The notion of avoiding interleavings is related to the techniques of partial order reduction [GW94] and dynamic partial order reduction [FG05, SA06b], but our algorithm uses it as a heuristic that does not take into account the user-defined properties being checked. Note that user-defined properties in Dervish can arbitrarily relate local states of different nodes. In the example of disjoint nodes above, a partial order reduction technique would explore CC' but, assuming perfect independence information, would not explore C'. Figure 3.3 illustrates this difference; the Appendix presents a larger example. The preference for a longer execution makes partial order reduction less appropriate for our scenario where we need to impose a bound on search depth.

Note that $hash(n, s)$ in Figure 3.2 implies that we have separate tables corresponding to each node for keeping hashed local states. If a state variable is not necessary to distinguish two separate states, the user can annotate the state variable that he or she does not want to be included in the hash function, improving the performance of Consequence Prediction (our experiments in Section 3.6 make use of this mechanism.). Instead of holding all encountered hashes, the hash table could be designed as a bounded cache of explored hash entries to fit into the L2 cache or main memory, favoring access speed while admitting the possibility of re-exploring previously seen states.

Note that consequence prediction algorithm does not prune any message processing events. For example, if a node resets, the other nodes in the system can detect this reset by trying to communicate with the node and receiving an RST signal.

Although simple, the idea of removing from the search the internal actions of nodes with previously seen states eliminates many (uninteresting) interleavings from the search and has a profound impact on the search depth that the model checker can reach with a limited time budget. This change was therefore the key to enabling the use of the model checker at runtime. 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 evaluation (Section 3.6).

3.3 Execution Steering

Dervish's execution steering mode enables the system to avoid entering an erroneous state by steering its execution path away from the 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. Moreover, in the new programming model [YVKK09] which has been suggested recently, the developer can explicitly define multiple handlers for a given event. At run-time, the run-time support module picks the one that satisfies the user-defined objective, such as consistency. In the present thesis, however, we focus on generic runtime mechanisms that do not require the developer to change the programming model or explicitly specify corrective actions.

3.3.1 Event Filters

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 could lead to an erroneous state, Dervish installs an *event filter*, which temporarily blocks the invocation of the state machine handler for the messages from the relevant sender.

The rationale for event filters 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 safety property. The joining node can later retry the procedure with an alternative potential parent and successfully join the tree. Similarly, if handling a message causes 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. Note that state machine handlers are atomic, so Dervish is unlikely to interfere with any existing recovery code.

3.3.2 Granularity of Filters

There is a trade-off in the granularity of the filter that we can install to steer away from the inconsistency. By using a coarse granularity filter, we can cover more cases that can possibly lead to the same inconsistency and hence reduce the false negative rate. However, doing so might increase the false positive rate by filtering other events which are not threatening the consistency of the system. On the other hand, using a fine granularity filter would decrease the false positive rate as it affects only the events that are more likely to reach the inconsistency point. Doing so could result in a higher false negative rate since there might be other erroneous paths which the model checker could not find in time.

There has been related work that tries to determine the right granularity for the filter by further processing around the erroneous path using symbolic execution and path slicing [CCZ+07]. Although these techniques can be effective in general, they are not directly applicable to execution steering, because the filter needs to be installed very quickly to catch the erroneous events in time.

The granularity we adopt in this thesis is at the level of a handler identifier. In particular, we filter on the handler whose execution in the model checker led to an inconsistency. In the case of handlers for network messages, we include the source address of the received message in the filter. It is worth noting that we can have multiple handlers for the same event. They differ in the guard condition (i.e., the boolean expression defined over the state variables) and the received message header that determines which handler is allowed to execute.

3.3.3 Point of Intervention

In general, execution steering can intervene at several points in the execution path. Given several choices with equal impact in terms of predicted errors, our policy is to steer the execution as early as possible (subject to non-disruptiveness criteria described below). 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 discovers the erroneous path, it can install the event filter.

We choose the earliest-point-of-intervention policy because it gives the system more choices (a longer execution) to adjust to the steering action. Moreover, we have a separate mechanism, the immediate safety check (described below), that enables a node to perform last-minute corrections to its behavior.

3.3.4 Non-Disruptiveness of Execution Steering

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. Dervish therefore makes execution steering safe in practice through two mechanisms:

1. **Sound Choice of Filters.** It is important that the chosen corrective action does not sacrifice the soundness of the state machine. A *sound filtering* is the one in which the observed finite executions *with* filtering are a subset of possible finite executions *without* filtering. Consequently, if a finite execution appeared in the presence of execution steering, it was also possible (even if less likely) for it to appear in the original system. The breaking of a TCP connection is common in a distributed system using TCP. This makes sending a TCP RST signal a good candidate for a sound event filter, and is the filter we choose to use in Dervish. In the case of communication over UDP, the filter simply drops the UDP packet, which could similarly happen in normal operation of the network. The sound filter for timer events is to avoid invoking the timer handler and to reschedule the timer firing. This is a sound filter because there is no real-time guarantees for invoking the scheduled timers on time, hence the timer execution could have been postponed during normal system operation.
2. **Exploration of Corrected Executions.** Before allowing the event filter to perform an execution steering action, Dervish runs the consequence prediction algorithm to check the effect of the event filter action on the system. If the consequence prediction algorithm does not suggest that the filter actions are safe, Dervish does not attempt execution steering and leaves the system to proceed as usual.

3.3.5 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. Dervish 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, Dervish immediately reinstalls the appropriate filter.

3.3.6 Immediate Safety Check

Dervish also supports *immediate safety check*, a mechanism that avoids inconsistencies which 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.

We have found that exclusively using immediate safety check would not be sufficient for avoiding inconsistencies. The advantages of installing event filters are: (i) performance benefits of avoiding the error sooner, e.g., reducing unnecessary message transmission, (ii) faster reaction to an error, which implies greater chance of avoiding a "point of no return" after which error avoidance would be impossible, and (iii) the node that is supposed to ultimately avoid the inconsistency by immediate safety check might not have all the checkpoints needed to notice the violation; this can result in false negatives (as shown in Figure 3.10).

3.3.7 Liveness Properties

The notion of safe filtering (presented above) ensures that no new *finite* executions are introduced into the system by execution steering. It is possible, in principle, that applying an event filter would affect liveness properties of a distributed system (i.e., introduce new infinite executions). In our experience, due to a large amount of non-determinism (e.g., the node is bootstrapped with a list of multiple nodes it can join), the system usually finds a way to make progress. We focus on enforcing safety properties; according to a negative result by Fischer, Lynch, and Paterson [FLP85], it is anyway impossible to have safety and liveness in an asynchronous system.

3.4 Scope of Applicability

Dervish does not aim to find all errors; it is rather designed to find and avoid important errors that can manifest in real runs of the system. Results in Section 3.6 demonstrate that Dervish works well in practice. Nonetheless, we next discuss the limitations of our approach and characterize the scenarios in which we believe Dervish would be effective.

Up-to-Date Snapshots. For Consequence Prediction to produce results relevant for execution steering and immediate safety check, it needs to receive

sufficiently many node checkpoints sufficiently often. (this is not a problem for deep online testing.) We expect the stale snapshots to be less of an issue with *stable properties*, e.g., those describing a deadlock condition [CL85]. Since the node's own checkpoint might be stale (because of enforcing consistent neighborhood snapshots for checking multi-node properties), immediate safety check is perhaps more applicable to node-local properties.

Higher frequency of changes in state variables requires higher frequency of snapshot exchanges. High-frequency snapshot exchanges in principle lead to: (1) more frequent model checker restarts (given the difficulty in building incremental model checking algorithms), and (2) high bandwidth consumption. Among the examples for which our techniques is appropriate are overlays in which state changes are infrequent.

Consequence Prediction as a Heuristic. Consequence Prediction is a heuristic that explores a subset of the search space. This is an expected limitation of explicit-state model checking approaches applied to concrete implementations of large software systems. The key question in these approaches is directing the search towards most interesting states. Consequence Prediction uses information about the nature of the distributed system to guide the search. The experimental results in Section 3.6 show that it works well in practice, but we expect that further enhancements are possible.

Applicability to Less Structured Systems. Dervish uses the Mace framework [KAB+07] and presents further evidence that higher-level models make the development of reliable distributed systems easier [DKK09,KAJV07]. Nevertheless, consequence prediction algorithm and the idea of execution steering are also applicable to systems specified in a less structured way. While doing so is beyond the scope of this thesis, recently proposed tools such as MODIST [YCW+09] could be combined with our approach to bring the benefits of execution steering to a wider range of distributed system implementations. Given the need for incomplete techniques in systems with large state spaces, we believe that consequence prediction remains a useful heuristics for these scenarios.

3.5 Implementation

Our Dervish prototype is built on top of Mace [KAB+07]. Mace allows distributed systems to be specified succinctly, and it outputs high-performance C++ code. We run the model checker as a separate process 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 the notion of logical time. It also includes an implementation of the consequence prediction algorithm, with the ability to replay the previously discovered paths which led to

some inconsistencies. Finally, it contains an implementation of the execution steering mechanism.

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

3.5.2 Consequence Prediction

Our starting point for the consequence prediction algorithm was the publicly available MaceMC implementation [KAJV07]. 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. Operating systems already have techniques to balance the load of processes among the available CPU cores. Furthermore, some kernels (e.g., FreeBSD, Windows Vista, and Linux) already have interfaces support for pinning down applications to CPU cores.

3.5.3 Immediate safety check

Our current implementation of the immediate safety check executes the handler in a copy of the state machine (using `fork()`), and avoids the transmission of the messages. In general, delaying message transmission can be done by adding an extra check in the messaging layer: if the code is running in the child process, it ignores the messages which are waiting in queue for transmission. However, because: (1) message transmission in the Mace framework is done by a separate thread and (2) by running `fork()` we only duplicate the active thread, adding the extra check would be redundant.

Our implementation must be careful about the resources that are shared between the parent (primary state machine) and the child process (copy). For example, the file descriptors shared between two processes might cause a problem for the incoming packets: they might be received either by the parent or the child process. The approach we took is to close all the file descriptors in the child process (the one doing immediate safety check) after calling `fork()`. The `fork()` happens after the system has reacted to a message and the execution of the handler does not read further messages. In principle, file descriptors can be used for arbitrary I/O, including sending messages and file system operations. In our implementation however, messaging is under Mace's control and it is straightforward to hold message transmission. In our experiments, only Bullet′ [KBK+05] was performing file system I/O and we manually implemented buffering for its file system operations. A more flexible solution could be implemented on top of a transactional operating system [PHR+09].

The other shared resources that we need to consider are locks. After forking, the locks could be shared between parent and child process and waiting on them in the child process might cause undesirable effects in both the parent and the child process. We address this issue by adding an extra check in the Mace library files which operate on locks; if the code is running as child process, the library does not invoke locking/unlocking functions. This choice does not affect the system operation because the locks are used for synchronization between threads. In the child process, there is only one thread, which eliminates the need for using locks.

Since modern operating system implementations of `fork()` use the copy-on-write scheme, the overhead of performing the immediate safety check is relatively low (and it did not affect our applications). In case of applications with high messaging/state change rates where the performance of immediate safety check is critical, one could obtain a state checkpoint [SKAZ04] before running the handler and rollback to it only in case of an encountered inconsistency. Another option would be to employ operating system-level speculation [NCF05].

Upon encountering an inconsistency in the copy, the runtime does not execute

the handler in the primary state machine. Instead, it employs a sound event filter (Section 3.3).

3.5.4 Replaying Past Erroneous Paths

To check whether an inconsistency that was reported in the last run of model checker can still occur when starting from the current snapshot, we replay past erroneous paths. Strictly replaying a sequence of events and messages that form the erroneous path, on the new neighborhood snapshot might be incorrect. For example, some messages could have only been generated by the old state checkpoints and would thus be different from those the new state can generate. Our replay technique therefore replays only timer and application events, and relies on the code of the distributed service to generate any messages. We then follow the causality of the newly generated messages throughout the system.

The high fidelity replay of erroneous paths necessitates a deterministic replay of pseudo-random numbers. Recall that the model checker systematically explores all possible return values of the pseudo-random number generator. [1] This function is called in three cases: (i) in the protocol implementation by the developer to get a random value, (ii) in the simulator to decide whether to simulate a fault injection in the current event, and (iii) in the model checker to pick one of the enabled events to be simulated in the next round. Because the list of enabled events changes based on the recent received checkpoints, we only aim to cover the first two cases for deterministically replaying pseudo-random number generation.

Toward this end, we instrumented the pseudo-random number generator function. During simulation of events, we record the values returned by the pseudo-random number generator; these values are then appended to the recorded information corresponding to the erroneous path. Later on, while replaying the erroneous path, we supply these values instead of calling the pseudo-random number generator.

3.5.5 Event Filtering for Execution Steering

Execution steering is driven by the reports received from the model checker; reports are sequences of events which could lead to inconsistencies.

The Dervish controller then picks a subset of these events and installs the corresponding filters. In general, the nodes could collaborate to install disjoint sets of filters corresponding to each reported erroneous path. We have used

[1] Note that the pseudo-random number generator should be carefully invokes to pick a member of a small set. Otherwise, large number of possible outcomes will quickly lead to an exponential explosion in state space size.

a simple but effective approach for picking the event to filter on: the node which discovers the erroneous path looks for its first contribution to this path and installs the filter for that event. Recall that filters are designed for protocol-specific events, hence the events like node reset or message loss which are beyond the control of the protocol, will be ignored. Upon checking the existence and the potential impact of a corrective action, the Dervish controller installs an event filter into the runtime.

3.5.6 Checking Safety of Event Filters

To check the safety of event filters, we modified our baseline execution steering library. When the execution steering module wants to check the safety of an event filter, it checks that if the code is running inside model checker, then it randomly selects both safe and unsafe choices. Since a call to the pseudo-random number generator causes a branch in the model checker, the model checker explores both cases: (i) where handling the event is safe and its corresponding handler will be called, and (ii) where handling the event is detected to be unsafe and the filter will be installed.

Each handler execution then adds two branches into the state space: (1) normal execution branch and (2) filtered branch. Although important for checking the safety of event filters, this technique increases the number of branches that the model checker needs to check and hence has a negative impact on the model checker efficiency. Assuming rare manifestation of erroneous paths, we can optimize this technique by enabling the filtered branch only after an inconsistency is reached in the exploration of the normal execution branch. The challenge however is that the B-DFS algorithm is implemented based on the assumption that the paths will be re-explored deterministically, and exploration of new states in the enabled branch would change the duplicate state detection behavior. We address this issue by isolating the exploration of filtered branches. For example, the states that are explored in this branch would not be reflected in the model checker data structures.

3.6 Evaluation

Our experimental evaluation addresses the following questions:

1. Is Dervish effective in finding inconsistencies in live runs?
2. Can any of the inconsistencies found by Dervish also be identified by the MaceMC model checker alone?
3. Is execution steering capable of avoiding inconsistencies in deployed distributed systems?

4. Are the Dervish-induced overheads within acceptable levels?

3.6.1 Experimental Setup

We conducted our live experiments using ModelNet [VYW+02]. 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 hyperthreading, 8 machines with dual 2.33 GHz dual-core Xeon 5140s, and 3 machines with 2.83 GHz Xeon X3360s (for Paxos experiments). Older machines have 2 GB of RAM, while the newer ones have 4 GB and 8 GB, respectively. 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 testing, 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 testing experiments, one per machine.

We use a 5,000-node INET [CGJ+02] 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 130 ms. 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.

3.6.2 Deep Online testing Experience

We have used Dervish 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 [SMLN+03]). These implementation were not only manually debugged in both local- and wide-area settings, but were also model checked using MaceMC [KAJV07]. We have also used our tool to find inconsistencies in Bullet′, a file distribution system that was originally implemented in MACEDON [RKB+04], and then ported to Mace. We found 13 new subtle bugs in these three systems that caused violation of safety properties. In 7 of inconsistencies, the violations were beyond the scope of exhaustive search by the 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 3.1: Summary of inconsistencies found for each system using Dervish. 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 succinctly. The C++ numbers do not include the line counts for libraries and low-level services that services use from the Mace framework.

Table 3.1 summarizes the inconsistencies that Dervish 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 tested to go through complex realistic scenarios.[2] Dervish 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.

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. The first identified scenario by Dervish that violates this property is the scenario from Figure 1.2 in Section 1.2. The problem can be corrected by removing the stale information about children in the handler for the UpdateSibling message. Dervish also identified variations of this bug that require changes in other handlers.

Scenario #2 exhibiting inconsistency. During live execution, node n_a is initially the root of the tree and parent of node n_b. Node n_r tries to join the tree by sending a join request to n_a. n_a accepts the request and decides that n_r should be the root of the tree, because it has a smaller identifier. Therefore, n_a joins under n_r. After receiving the JoinReply message, n_r is the root of the tree and n_a's parent. At this point, consequence prediction detects the following

[2] During this time, the model checker ran concurrently with a normally running system. We therefore do not consider this time to be wasted by the model checker before deployment; rather, it is the time consumed by a running system.

scenario. Node n_b resets, but its TCP RST packet to n_a is lost. Then, n_b sends a Join request to n_r, which is accepted by n_r because it has a free space in its children list. Accordingly, n_r sends a sibling update message to its child, n_a. Upon receipt of this message, n_a updates its sibling list by adding n_b to it, while the children list still includes n_b.

Scenario #3 exhibiting inconsistency. During live execution, node n_r is the root of the tree and parent of nodes n_a and n_c. Then, node n_b joins the tree under n_a. At this point, consequence prediction detects the following scenario. n_b experiences a node reset, but its TCP RST packet to n_a is lost. Then, n_b sends a Join request to n_r and n_r forwards the request to n_c. After that, n_r experiences a node reset and resets the TCP connections with its children n_a and n_c. Upon receiving the error signal, each of them removes n_r from its parent pointer and promotes itself to be the root. In addition, each of n_a and n_c, sets its join timer to find the real root. Join timer of n_a expires and sends a Join message to n_c, which is accepted by n_c because it has a free space in its children list. Upon receipt of JoinReply message, n_a adds n_b to its sibling list, while n_b is in its children list as well.

Scenario #4 exhibiting inconsistency. During live execution, node n_r is the root of the tree and the parent of nodes n_a and n_c. At this point, consequence prediction detects the following scenario. Upon receiving the error signal, both nodes remove n_r from its parent pointer and promote themselves to be the root. In addition, both n_a and n_c start their join timer to find the real root. The join timer of n_a expires and n_a sends a Join message to n_c, which is accepted by n_c because it has a free space in its children list. Recall that n_a is still a sibling of n_c.

Possible corrections. In the message handler, check the children list before adding a new sibling. In the case of a conflict, we can either simply trust the new message and remove the conflicting node from the children list or query the offending node to confirm its state.

Root is Not a Child nor Sibling. Dervish found violation of the property that the root node should not appear as a child, identifying a node n_{69} that considers node n_9 both as its root and its child.

Scenario exhibiting inconsistency. During live execution, node n_{61} is initially the root of the tree and parent of nodes n_5, n_{65}, and n_{69}; n_{69} is also parent of n_9. At this point, consequence prediction detects the following scenario. Node n_9 resets, but its TCP RST packet to its parent (n_{69}) is lost. n_9 sends a Join request to n_{61}. Based on n_9's identifier, n_{61} considers n_9 more eligible and selects it as the new root and indicates it in the JoinReply message. Besides, it also sends a Join message to the new root which would be n_9. After receiving a JoinReply from n_9, n_{61} informs its children about the new root (n_9) by sending

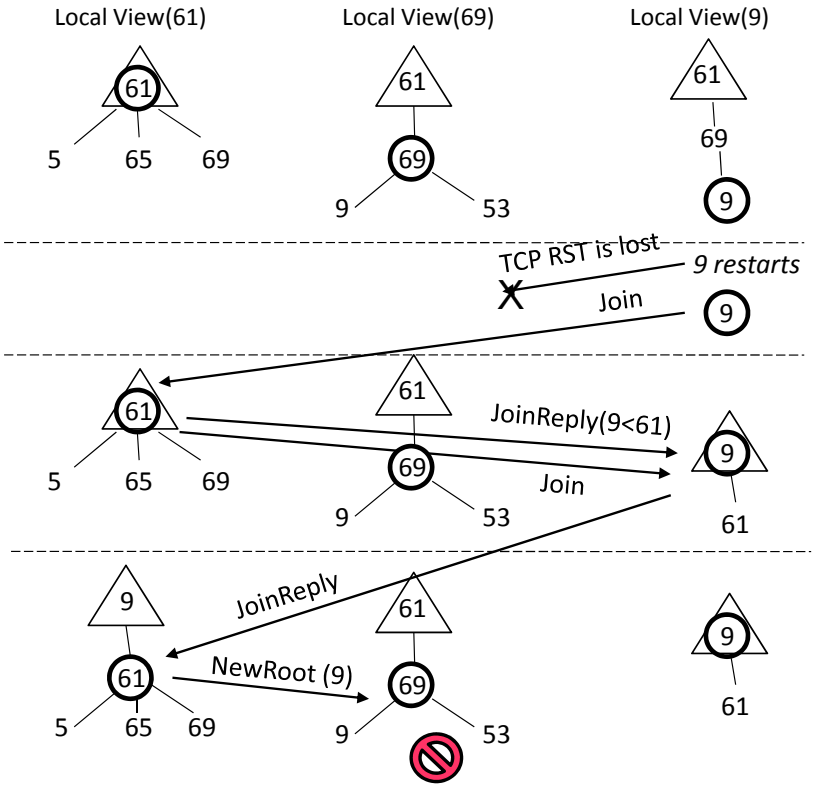

Figure 3.4: An inconsistency in a run of RandTree. Root (n_9) appears as a child.

NewRoot messages to them. However, n_{69} still thinks n_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. Dervish found violation of the property that the root node should contain no sibling pointers, identifying a node n_a that considers itself a root but at the same time has an address of another node n_b in its sibling list.

Scenario exhibiting inconsistency. During live execution, node n_a is initially the root of the tree and parent of nodes n_b and n_c. Node n_r sends a Join request to n_a. Based on n_r's identifier, n_a considers n_r more eligible to be the root and thus informs it of its new role. n_a notifies its children about the new root by sending them the NewRoot messages. At this point, consequence prediction detects the following scenario. n_a experiences a node reset and resets the TCP connections with its children n_b and n_c. Upon receiving the error signal, n_b

removes n_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 each node to send Probe messages to the members of its peer list with which it does not have an open connection. It is vital for the tree's consistency to keep the nodes up-to-date about the global structure of the tree. The property that checks whether the recovery timer is always running was written by the authors of MaceMC [KAJV07] 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. Dervish found a violation of the property in a state where node n_a joins itself, and changes its state to "joined" but does not schedule any timers. Although this does not cause problems immediately, the inconsistency occurs when another node n_b with a smaller identifier tries to join, at which point node n_a gives up the root position, selects n_b as the root, and adds n_b it to its peer list. At this point n_a has a non-empty peer list but no running timer.

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

Example Chord Bugs Found

We next describe violations of consistency properties in Chord [SMLN$^+$03], a distributed hash table that provides key-based routing functionality. Chord and other related distributed hash tables form a backbone of a large number of proposed and deployed distributed systems [JMK$^+$08, RGK$^+$05, RD01].

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 the list of its successors. 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 n_a first identifies its potential predecessor by querying with its Id. This request is routed to the appropriate node n_p, which in turn replies to n_a. Upon receiving the reply, n_a inserts itself between n_p and n_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.

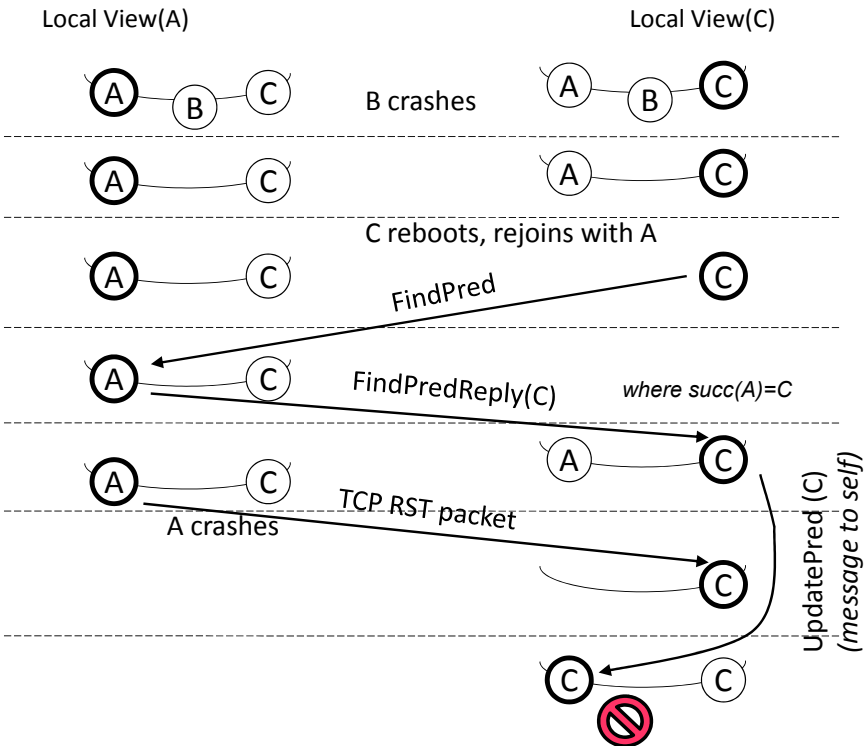

Figure 3.5: An inconsistency in a run of Chord. Node n_c has its predecessor pointing to itself while its successor list includes other nodes.

If Successor is Self, So Is Predecessor. If a predecessor of a node n_a equals n_a, then its successor must also be n_A (because then n_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.

Scenario exhibiting inconsistency: Dervish found a state where node n_a has n_a as its predecessor but has another node n_b as its successor. This violation happens at depths that are beyond those reachable by exhaustive search from the initial state. Figure 3.5 shows the scenario. During live execution, several nodes join the ring and all have a consistent view of the ring. Three nodes n_a, n_b, and n_c are placed consecutively on the ring, i.e., n_a is predecessor of n_b and n_b is predecessor of n_c. Then n_b experiences a node reset and other nodes which have established TCP connection with n_b receive a TCP RST. Upon receiving this error, node n_a removes n_b from its internal data structures. As a consequence, node n_a considers n_c as its immediate successor.

Starting from this state, consequence prediction detects the following scenario that leads to the violation. n_c experiences a node reset, losing all its state. n_c

then tries to rejoin the ring and sends a FindPred message to n_a. Because nodes n_a and n_c did not have an established TCP connection, n_a does not observe the reset of n_c. Node n_a replies to n_c by a FindPredReply message that shows n_a's successor to be n_c. Upon receiving this message, node n_c (i) sets its predecessor to n_a; (ii) stores the successor list included in the message as its successor list; and (iii) sends an UpdatePred message to n_a's successor which, in this case, is n_c itself. After sending this message, n_c receives a transport error from n_a and removes n_a from all of its internal structures including the predecessor pointer. In other words, n_c's predecessor would be unset. Upon receiving the (loopback) message to itself, n_c observes that the predecessor is unset and then sets it to the sender of the UpdatePred message which is n_c. Consequently, n_c 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.). 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 the Chord specification, a node's predecessor pointer contains the Chord identifier of the immediate predecessor of that node. Therefore, if a node n_a has a predecessor n_p and one of its successor is S, then the Id of S should *not* be between the Id of n_p and the Id of n_a.

Scenario exhibiting inconsistency. Dervish found a safety violation where node n_a adds a new successor n_b to its successor list while its predecessor pointer is set to n_c and the Id of n_b is between the Id of n_a and n_c. The scenario discovered is as follows (Figure 3.6). The Id of n_b is less than the Id of n_a and the Id of n_a is less than the Id of n_c. During live execution, first, node n_c joins the ring. Then, nodes n_a and n_b both try to join with n_c by sending FindPred messages to it. Node n_c sends FindPredReply back to n_a and n_b with exactly the same information. Upon receipt of this message, nodes n_a and n_b set their predecessor and successor to n_c and send UpdatePred message back to n_c. Finally, node n_c sets its predecessor to n_a and successor to n_b.

In this state, consequence prediction discovers the following actions. Stabilizer timer of n_a fires and this node queries n_c by sending it a GetPred message. Node n_c replies back to n_a with a GetPredReply message that shows the n_c's predecessor to be n_a and its successor list to contain n_b. Upon receiving this message, n_a adds n_b to its successor list while its predecessor pointer still points to n_c.

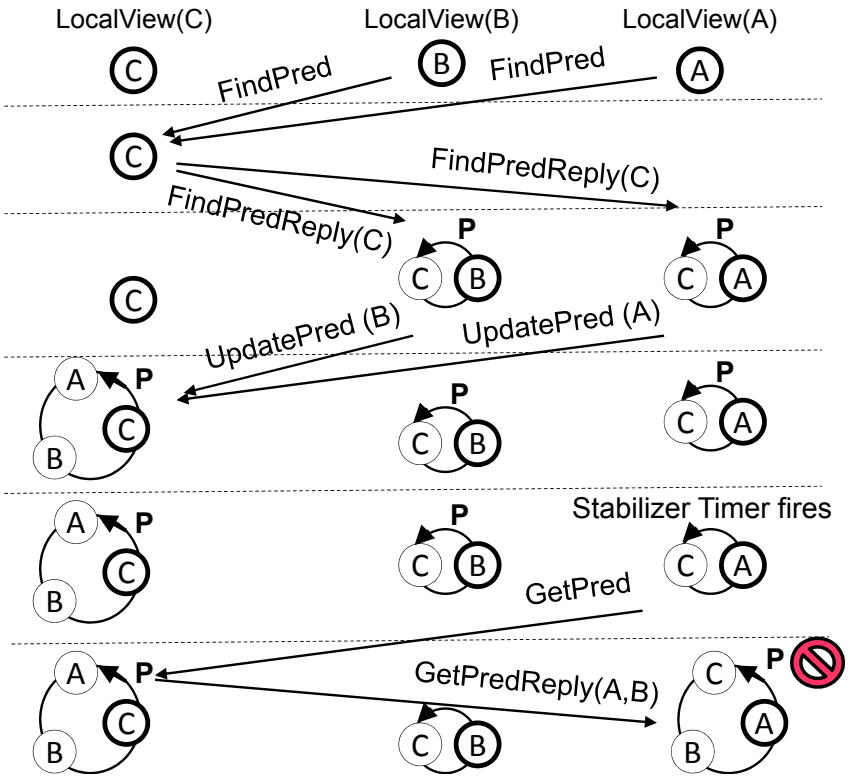

Figure 3.6: An inconsistency in a run of Chord. For node n_a, its successor and predecessor do not obey in ordering constraint.

Possible correction. The bug occurs because node n_a 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.

Bound on data structure size. An implicit property of the data structure of a protocol is that there should be a reasonable limit on its size. Usually, this limit is internally enforced by the data structure. Chord keeps a queue called findQueue to store the received join requests while the node has not joined the ring. The node feeds requests as loop back messages to itself to respond to them. We set a property that puts a limit on the size of this queue.

Scenario exhibiting inconsistency. Dervish found a safety violation where the size of this queue gets unreasonably large. This causes both (i) a performance problem for processing all of the messages, and (ii) increase in the size of the node checkpoint from a few KB to 1 MB. The bug occurs when the potential root of the tree (i.e. n_r) has not yet joined and other nodes of the tree stubbornly send join requests to it. When n_r finally joins the ring, it has a very long list of request to process.

Possible correction. The bug exists because the Mace Chord implementation does not avoid inserting duplicate messages in this queue. This bug was not observable using MaceMC because it does not model check more than several nodes (because of the state explosion problem). Our live 100-node run using Dervish uncovered the bug.

Example Bullet′ Bug Found

Next, we describe our experience of applying Dervish to the Bullet′ [KBK+05] 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 three bugs we found were results of inconsistencies involving the variants of property (1). Since the conditions that led to the inconsistencies are similar, we describe a bug involving the following property:

Sender's File Map and Receivers View of it Should Be Identical. Every sender keeps a "shadow" file map for each receiver informing 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, Dervish 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. Interestingly 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.

3.6.3 Comparison with MaceMC

To establish the baseline for model checking performance and effectiveness, we installed our safety properties in the original version of MaceMC [KAJV07]. We then ran it for the three distributed services for which we had identified safety violations. After 17 hours, exhaustive search did not identify any of the violations caught by Dervish. 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 3.7 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. This illustrates the limitations of exhaustive search from the initial state.

In another experiment, we additionally employed random walk feature of MaceMC. Using this setup, MaceMC identified some of the bugs found by Dervish, but it still failed to identify 2 Randtree, 2 Chord, and 3 Bullet′ bugs found by Dervish. 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, Dervish identified violations that are close to live executions and therefore more likely to occur in the immediate future.

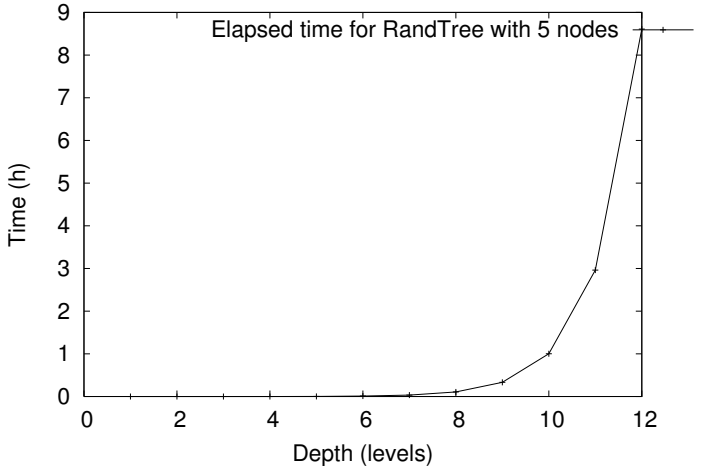

Figure 3.7: MaceMC performance: the elapsed time for exhaustively searching in RandTree state space.

3.6.4 Execution Steering Experience

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

RandTree Execution Steering

To estimate the impact of execution steering on deployed systems, we instructed the Dervish controller to check for violations of RandTree safety properties (including the one described in Section 3.6.2). 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 Dervish 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

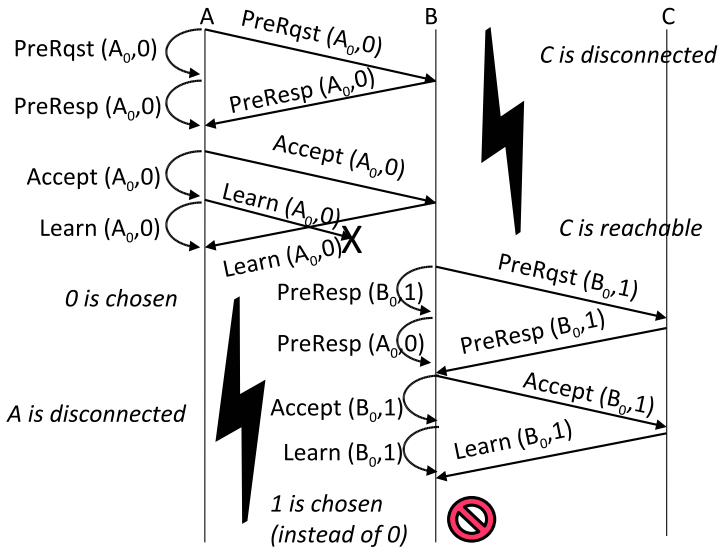

Figure 3.8: Scenario that exposes a previously reported violation of a Paxos safety property (two different values are chosen in the same instance.).

fallback engages 160 times. Through a combined action of execution steering and immediate safety check, Dervish avoided all inconsistencies, so there were no uncaught violations (false negatives) in this experiment.

To understand the impact of Dervish 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 joining time between 0.8 and 0.9 seconds across different experiments, with variance exceeding any difference between the runs with and without Dervish. In summary, Dervish changed system actions 415 times (2.77% of the total of 14,956 actions executed), avoided all specified inconsistencies, and did not degrade system performance.

Paxos Execution Steering

Paxos [Lam98] is a well-known fault-tolerant protocol for achieving consensus in distributed systems. Recently, it has been successfully integrated in a number of deployed [CGR07, LLPZ07] and proposed [JKBK+08] 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 previously deployed systems [CGR07, LLPZ07]. The Paxos protocol includes five steps:

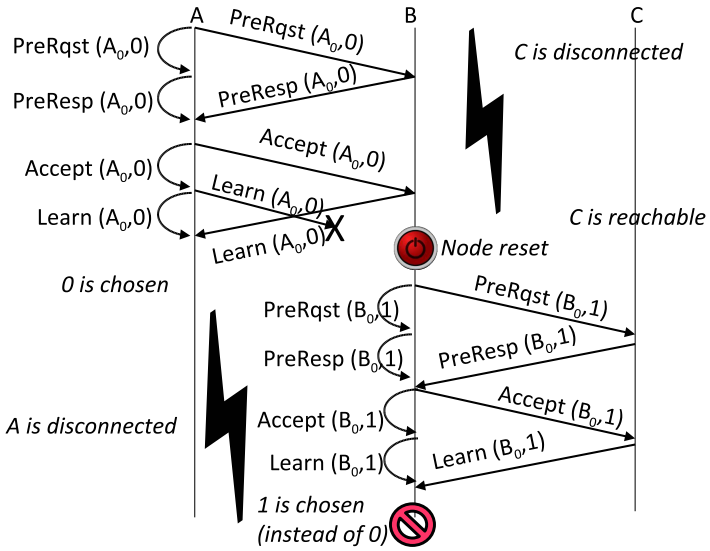

Figure 3.9: Scenario that includes *bug2*, where node n_b resets but after reset forgets its previously promised and accepted values. This leads to violation of the main Paxos safety property (two different values are chosen in the same instance.).

1. A leader tries to take the leadership position by sending prepare_request messages to acceptors, and it includes a unique round number in the message.
2. Upon receiving a prepare_request 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 prepare_response message that contains the last accepted value if there is any.
3. Once the leader receives a prepare_response message from a majority of acceptors, it broadcasts an accept request to all acceptors. This message contains the value of the prepare_response message with the highest round number, or is any value if the responses reported no proposals.
4. Upon the receipt of the accept message, 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 a majority of the nodes, a learner considers the reported value as chosen.

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 (leader, acceptor, learner) in the Paxos algorithm; 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. To speed up Consequence Prediction in the Paxos experiments, we annotated the unnecessary state variables to exclude them from the state hash.

The first bug we injected [LLPZ07] is related to an implementation error in step 3, and we refer to it as *bug1*: once the leader receives the prepare_response message from a majority of nodes, it creates the accept request by using the submitted value from the last prepare_response message instead of the prepare_response message with highest round number. Because the rate at which the violation (due to the injected error) occurs was low, we had to schedule some events to lead the live run toward the violation in a repeatable way.

The setup we use comprises three nodes and two rounds, without any artificial packet delays (other than those introduced by the network). As illustrated in Figure 3.8, in the first round the communication between node n_c and the other nodes is broken. Also, a learn packet is dropped from n_a to n_b. At the end of this round, n_a chooses the value proposed by itself (0). In the second round, the communication between n_a and other nodes is broken. Node n_b proposes a new value (1) but its messages are received by only nodes n_b and n_c. Node n_b responds by a promise message containing value 0, because this value was accepted by node n_b in the previous round. However, node n_c was disconnected in previous round and responds back by the same value proposed by node n_b (1). Here bug *bug1* shows up and node n_b upon receipt of the prepare_response of node n_c with value 1, broadcasts the accept message with this value (1). At the end of this round, the value proposed by n_b (1) is chosen by n_b itself. In summary, this scenario shows how a buggy Paxos implementation can choose two different values in the same instance of consensus.

The second bug we injected (inspired by [CGR07]) involves keeping a promise made by an acceptor, even after crashes and reboots. As pointed out in [CGR07], it is often difficult to implement this aspect correctly, especially under various hardware failures. Hence, we inject an error in the way an accepted value is not written 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 n_b. This scenario is depicted in Figure 3.9. The first round is similar to the first round in the scenario of *bug1*. At the end of this round, n_a chooses the value proposed by itself (0). Then, node n_b resets, but because of the *bug2* explained above, it forgets the values that were promised and accepted before the reset. In the second round, communication between n_a and the other nodes is broken. Node n_b proposes a new value (1) but its messages are only received by nodes n_b and n_c. Thus, both nodes n_b and n_c accept the default value proposed by n_b because: (1) node n_c was disconnected and did not know about the chosen value and (2) node n_b has forgotten its accepted value (0) after the reset (due to the injected *bug2*). At the end of this round, the value proposed by n_b (1) is chosen by n_b

Figure 3.10: In 200 runs that expose Paxos safety violations due to two injected errors, Dervish successfully avoided the inconsistencies in all but 1 and 4 cases, respectively.

itself. Consequently, two different values have been chosen in the same Paxos instance.

To stress test Dervish's ability to avoid inconsistencies at runtime, we repeat the live scenarios 200 times in the cluster (100 times for each bug) while varying the time between rounds uniformly at random between 0 and 20 seconds. As we can see in Figure 3.10, Dervish's execution steering is successful in avoiding the inconsistency at runtime 74% and 89% of the time for *bug1* and *bug2*, respectively. In these cases, Dervish starts model checking after node n_c reconnects and receives checkpoints from other participants. After running the model checker for 3.3 seconds, n_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 n_c rejects the prepare_request message sent by n_b. Execution steering is more effective for *bug2* than for *bug1*, as the former involves resetting n_b. This in turn leaves more time for the model checker to rediscover the problem by: (i) consequence prediction, or (ii) replaying a previously identified erroneous scenario. Immediate safety check engages 25% and 7% of the time, respectively (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 n_c at node n_b. Dervish could not prevent the violation for only 1% and 4% of the runs, respectively. The cause for these false negatives was the incompleteness of the set of checkpoints.

Figure 3.11: In this experiment we run Paxos across an emulated wide area network using ModelNet. The experiment contains 200 runs in which the same two errors as in Figure 3.10 were injected. Dervish successfully avoided the inconsistencies in all but 6 and 3 cases, respectively.

To evaluate the performance of Dervish's execution steering over latencies and packet drops typical of wide area networks, we run the same Paxos experiment, but this time across the wide area network emulated by ModelNet. The results are depicted in Figure 3.11. Overall, execution steering works well over wide area networks as well. However, the number of cases where execution steering and immediate safety check fail to detect the inconsistencies and prevent them from occurring are slightly higher. This occurs because higher latency and loss rate of wide area networks increases the chance that the model checker does not receive the consistent neighborhood snapshot in time. In execution steering, missing the updated checkpoint makes the model checker not to be able to predict the actions which are the consequences of the updated state. Similarly, the immediate safety check cannot detect the violation because of the stale snapshots.

Dervish causes a two-fold increase in CPU utilization. One CPU core becomes almost 100% utilized to run consequence prediction (up from zero utilization). On the core running the application itself, the CPU utilization is 8% peak burst compared to 6% without Dervish (in a one-second window). Given the availability of additional CPU cores and the fact that the model checking process is decoupled from the application, we consider this increase acceptable. Our approach is therefore in line with related efforts to improve reliability by leveraging increasing hardware resources (e.g. [BM05, AVY08]).

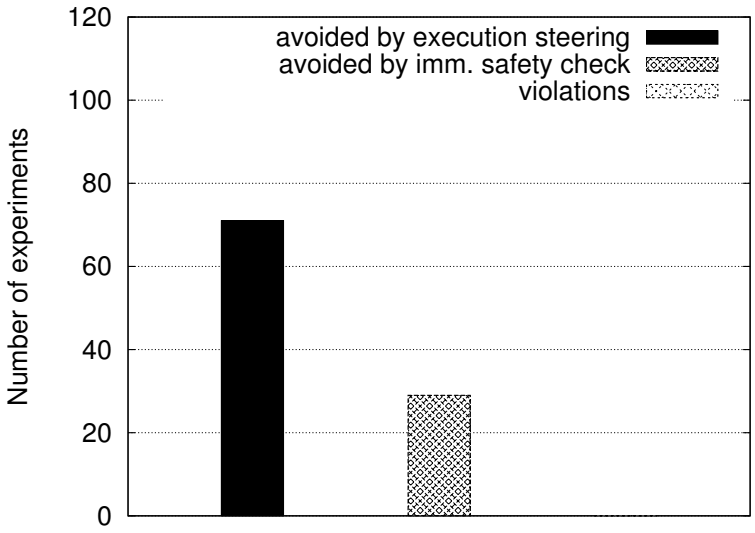

Figure 3.12: In 100 runs that expose a Chord safety violation we identified, Dervish successfully avoided the inconsistencies in all cases.

Chord

In the final set of our execution steering experiments, we stress test Dervish's ability to avoid violations of Chord safety properties. We use a scenario that repeatedly exposes a violation described in Section 3.6.2. Figure 3.12 demonstrates that Dervish's execution steering is successful in avoiding this inconsistency at runtime 100% of the time. Execution steering avoids 71% of the cases, while immediate safety check avoids the rest.

Performance Impact of Dervish

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 3.13 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, because the effective Dervish's search depth is less than seven or eight, the consumed memory by the search tree is less than 1 MB and can thus easily fit into the

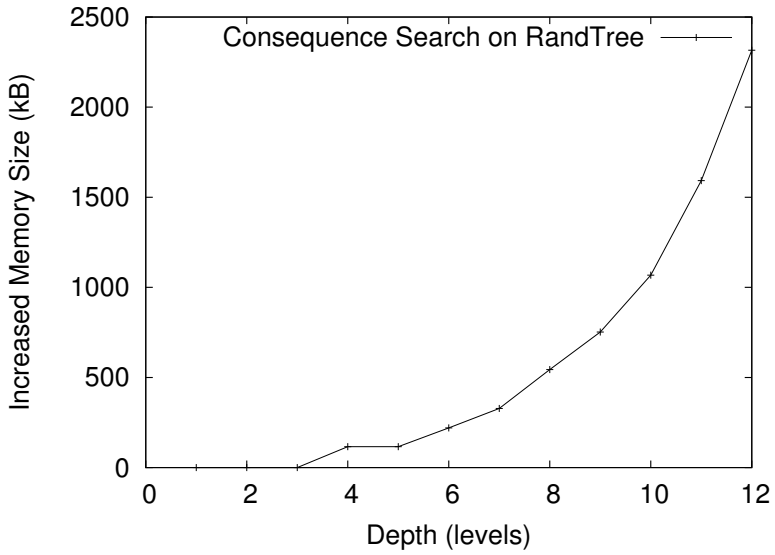

Figure 3.13: The memory consumed by consequence prediction (RandTree, depths seven to eight) fits in an L2 CPU cache.

L2 or L3 (most recently) cache of the state of the art processors. Having the entire search tree in-cache reduces the access rate to main 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 3.14, the per-state memory gets stable at about 150 bytes as we take more states into consideration and amortize the fixed amount of space used by the model checker.

In the deep online testing mode, the model checker 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 also 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 numbers show that overheads introduced by Dervish are low. Hence, we did not need to enforce any bandwidth limits in these cases.

Overhead from Checking Safety Properties. In practice we did not find the overhead of checking safety properties to be a problem because: (i) the number of nodes in a neighborhood snapshot is small, (ii) the most complex of our properties have $O(n^2)$ complexity, where n is the number of nodes, and (iii) the state variables fit into L2 cache.

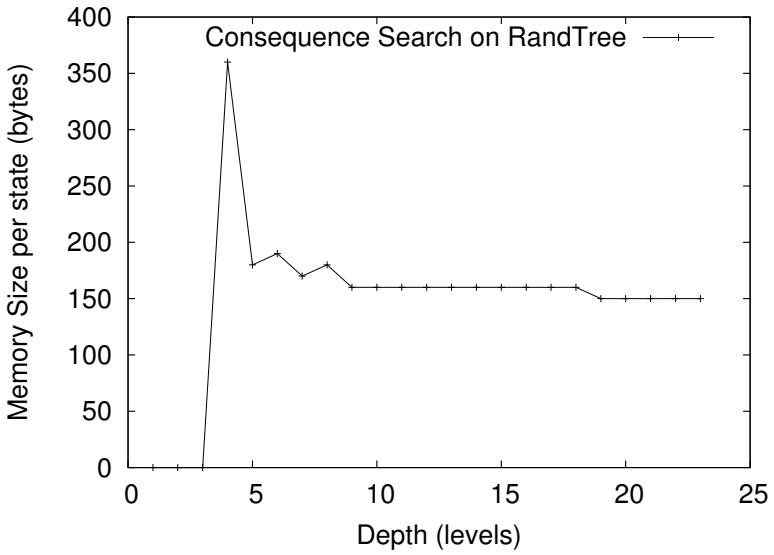

Figure 3.14: The average amount of memory consumed by each explored state.

Overall Impact. Finally, we demonstrate that having Dervish 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 Xeon CPU with hyper-threading. Figure 3.15 shows that in this case, using Dervish had a negative impact on 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 the bandwidth consumed by checkpoints.

3.7 Summary

In this chapter, 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, a heuristic for selectively exploring future states of the system, and developed a technique for obtaining consistent infor-

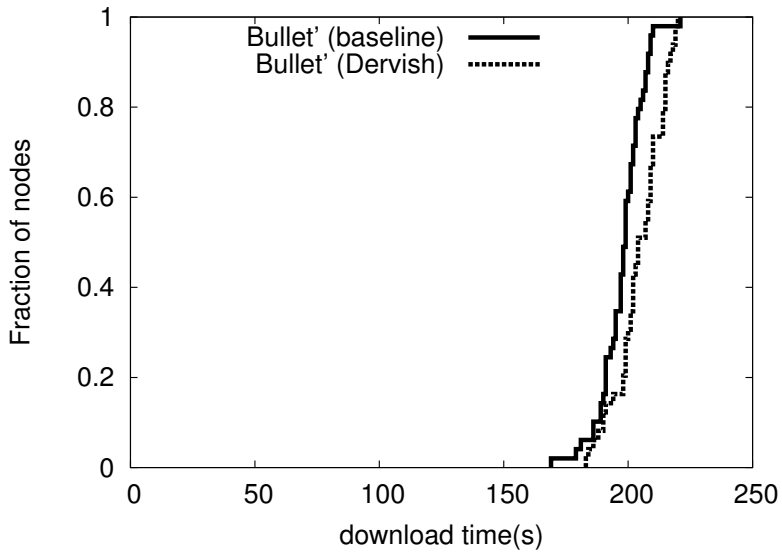

Figure 3.15: Dervish slows down Bullet′ by less than 5% for a 20 MB file download.

mation about the neighborhood of distributed system nodes. Our experiments suggest that the resulting system, Dervish, is effective in finding bugs that are difficult to detect by other means,

Society exists only as a mental concept; in the real world there are only individuals.

Oscar Wilde

4

LMC: Local Model Checking

At each step of model checking a centralized system, (i) one of the traversed states is selected, (ii) an enabled event is executed on that state, and (iii) the resulting state is added to the list of traversed states. The user-specified invariants are checked against the traversed states after each step and the set of these states grows exponentially with the *depth* of the exploration, i.e., the length of the sequence of enabled events considered. Current approaches to model checking distributed systems [KAB+07,KAJV07,YKKK10,YCW+09,MPC+02] reduce the problem to that of model checking a centralized system (Figure 4.1). The sets explored are *global* states comprising the *local* states of the nodes involved in the distributed system, i.e., the *system* state, as well as the *network* state involving the exchange of messages.

The exponential state space explosion problem manifests itself very quickly in this *global* approach, which makes the model checking of distributed systems practically ineffective. This is because the global state changes following any small change into a node local state or the network state. Consider for instance the celebrated Paxos protocol [Lam98], in the simple setting with three nodes where exactly one proposes at the start, i.e., no contention: it takes the global model checking approach 1514s (running on a 3.00 GHz Intel(R) Pentium(R) 4 CPU with 1 MB of L2 cache) to explore the interleaving of messages. In this very small state space, ∼180,000 transitions are executed by the model checker which, after eliminating duplicate global states, results into exploring ∼160,000 global states.

The starting point of this chapter is a couple of simple, complementary observations: (1) in the global model checking approach, the invariants are checked

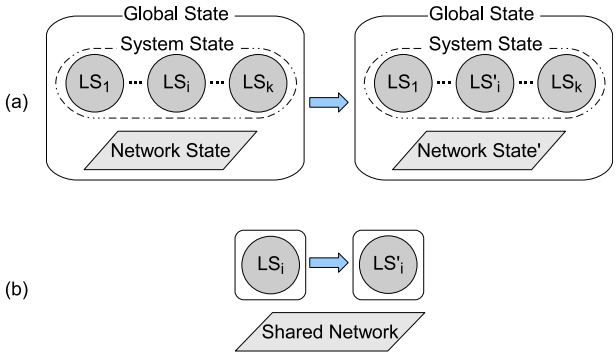

Figure 4.1: State transition in model checking distributed systems. In (a) the classic global approach, the model checker creates the entire state space of the global states, whereas in (b) our proposed local approach, the network element is eliminated from the stored states and the model checker keeps track of only local states.

on each traversed global state, although these invariants are typically specified only on the system states, i.e., the invariants do not involve the network states [KAJV07, YKKK10, YCW+09, MPC+02]; (2) for checking invariants that are defined on system variables, visiting the system part is a priori sufficient. Focusing on these states only, and ignoring the network states, significantly reduces the exploration space in comparison to the classic approach where each system state is typically repeated in multiple global states that differ only in the network part.

We present in this chapter a *local* model checking approach, which essentially consists in keeping track of the traversed local nodes' states separately by ignoring the network, a priori. Combined, these states are sufficient for invariant checking. As we explain in Section 4.4, our local model checking approach can be viewed as the combination of two approximation techniques: (i) a Cartesian approximation [BPR01, MPR06] of the set of system states out of independently explored local states, and (ii) a monotonic approximation of the network [DY83, Mit02] which always keeps the delivered messages in a shared state. We obtain by projection an overapproximation of the system state space, which enables a *complete*, efficient search with, however, potential false positives. We eliminate these with a complementary soundness verification technique.

Figure 4.1 contrasts our local approach with the global one. For the Paxos example state space with one proposal, our approach explores the entire system state in a few seconds by dropping the number of transitions to only ∼1000, which, after eliminating duplicate local states, results into only 180 local states: when compared to the classic approach, eliminating the network element induces an order of magnitude drop in the number of transitions. The total number of

recreated system states by our local approach is reduced to ∼90,000 (in the Paxos example). In contrast with the global model checking approach, in which visiting the system states is part of the exploration process, our local approach separates the exploration of transitions from the creation of system states. This makes it possible to ignore all system states on which the user-specified invariants can inherently not be violated: for instance, the Paxos invariant stipulates that no two decisions should be different and all undecided states can systematically be eliminated.

We show that our approach is *complete* in the sense that any violation of a system state invariant that could be detected by the global approach could be detected by our local approach. Two important remarks are however in order.

First, the combination of local states does not induce system states that are all *valid*: the fact that we ignore the network element, a priori, means that some combinations of local states might not occur in a real run. In other words, although complete, checking invariants on the retrieved system states is *unsound* since it could report a violation on an invalid system state. We address this problem by, a posteriori, verifying every preliminary violation report to make sure the sequence of events leading to the corresponding system state could also happen in a real run. An invariant violation is then reported to the user only if passes this test. If the number of preliminary violations is low enough, which turns out to be the case in our experiments, the performance penalty of verifying them becomes negligible.

Second, although our local approach is several orders of magnitude faster than the classic model checking approach, the state explosion problem is not eliminated. (The cost of invalid states created by our approach, although low at the start, will anyway eventually dominate in the general case.) Yet, we believe this can, to a large extent, be addressed by *online* model checking tools where the model checker is run for just a short period (a few seconds): in this case, our approach is efficient enough to search till depths of 20∼30 for the Paxos example state space.

To summarize, the contribution of this chapter is a new, local approach to model checking distributed systems. Instead of keeping track of global states, we eliminate the network element from the model checking states and keep only track of local states. We thus eliminate the overhead of ensuring soundness of every visited state (openning the door for effective invariant specific optimizations) and instead we verify soundness only on the states that violate the invariants. We present an efficient implementation of our approach and we show how this approach tracks bugs in two variants of Paxos, known to be one of the most complex distributed algorithms.

The rest of the chapter is organized as follows. Section 4.1 illustrates our approach through a simple example. Section 4.2 presents our approach. After

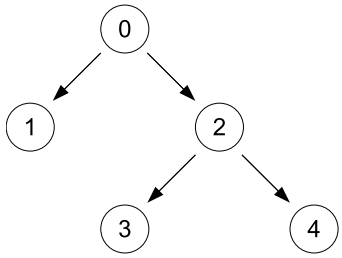

Figure 4.2: A simple distributed tree algorithm. Node 0 sends a message to all its children. Each node forwards the message to its children.

presenting the evaluation results in Section 4.3, we summarize the chapter in Section 4.5. We prove the correctness of our soundness verification procedure in Appendix D.

4.1 Local Model Checking: A Primer

Here we use a simple example to highlight the difference between classic (global) model checking and our proposed local approach. The example we consider here does not attempt to illustrate the performance improvements obtained by our approach but aims at explaining the main idea. The example system is a simple distributed tree structure, depicted in Figure 4.2. Node 0 initiates a message for node 4 and changes its state to `sent`. Each node, upon receiving a message, forwards it to its children. Node 4 changes its state to `received` upon receiving the message.

At each step of global model checking, the model checker transitions from a global state to another by running an enabled event, such as handling a message. The global state contains the network state besides the local state of all the nodes, i.e., the system state. The global state space of the example system is depicted in Figure 4.3. Each change into the network element causes the creation of a new global state. As one can observe, the number of system states covered by this global state space is much less than the global state space size.

Figure 4.4 illustrates our local approach on the same example system. Here, the network element, i.e., the non-essential parts for invariant checking, is separated form the model checking state. Instead, we keep a shared network component that receives the generated messages by all the transitions in the model checking. Observe that the messages added to the network are not removed by the executed transitions. In other words, the content of the shared network component is always increasing. As we will explain in Section 4.2, this is necessary for the completeness of the search.

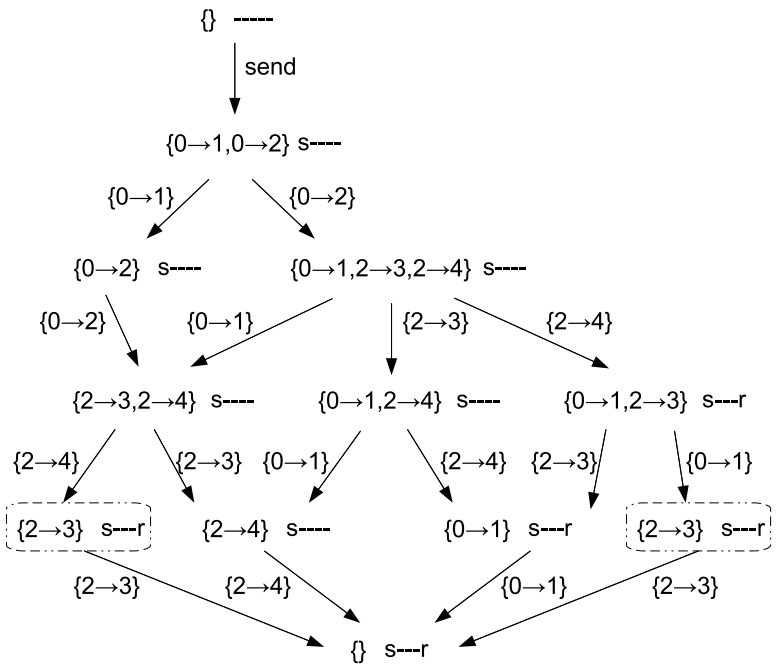

Figure 4.3: The global state space of the example tree in Figure 4.2 as explored by a global model checking approach. The initial local state of each node is denoted "-". The state of node 0 and 4 is changed to "s" and "r" after the send and receive of the message, respectively. The network element of the global state is represented by the set of in-flight messages. Each arrow depicts a transition in the model checker from one global state to another. The label besides each arrow indicates the event that triggers the transition. Although the global states inside the rectangles are duplicates, they are not joined into one state, for simplicity of presentation.

The last column of the figure depicts the new system states created after each step. The system states are created temporarily for the sake of being checked against the user-specified invariants. Observe that, in total, only 4 system states are created in contrast with the 12 global states of Figure 4.3. Moreover, the last system state, i.e., "----r" is invalid since node 4 could not receive the message before it is sent by node 0. Invalid states do not hurt the *completeness* of the exploration. After an invariant is violated on a system state, we run a *soundness* verification phase to ensure the validity of the system state.

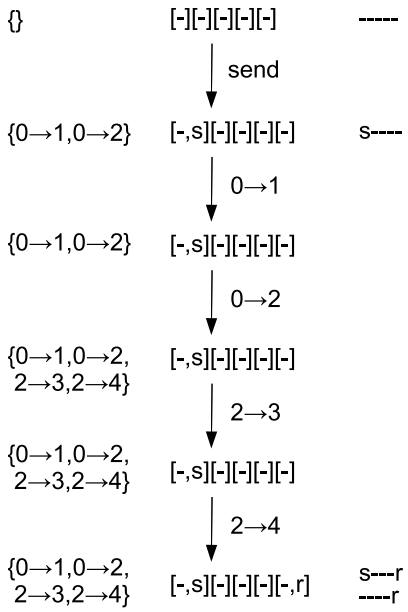

Figure 4.4: Local model checking approach of the example tree in Figure 4.2. The first column indicates the changes into the shared network element. The middle column shows the set of local states of node 0 to 4. The initial local state of each node is denoted "-". The state of node 0 and 4 is changed to "s" and "r" after the send and receive of the message, respectively. The first event is the local event of node 0 that generates the message. The generated message is then added to the shared network element. At each step, an event is selected and is executed on all local states of the destination node. The resultant states are added to the list of visited local states if they have not been visited before. The last column shows the new system states created after each step.

4.2 Design

The classic approach to model checking distributed systems keeps track of the traversed global states, e.g., Variable `explored` in Figure 2.2. (A *global state* consists of nodes' *local states* as well as the network state.)

The architecture of our local model checking approach is depicted in Figure 4.5. In this approach, the model checker keeps track of nodes' local states separately: set LS^i contains all the traversed states of node N_i. This is enough to recreate the *system states* upon which the invariants are checked. After a preliminary violation report on a system state, the validity of the system state is checked by a soundness verification module. If the system state is confirmed to be valid, the error is then (and only then) reported to the developer.

As shown in Figure 4.5, the handler execution module receives input only

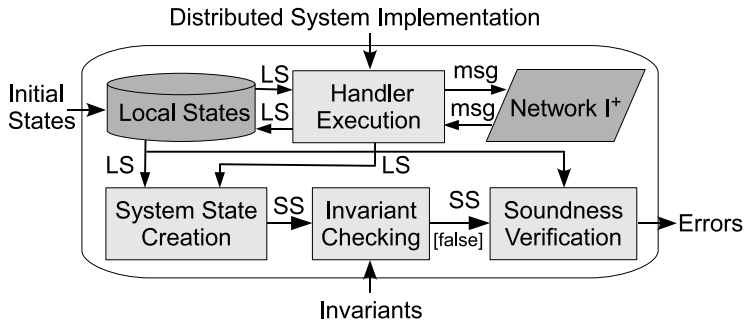

Figure 4.5: In our local approach, the handler execution works only on local states and produces new local states. Local and system states are denoted "LS" and "SS", respectively. The messages are not removed from the shared network component after execution. The new system states are created after a new local state is produced. The soundness verification checks the validity of a system state, only after an invariant violation is reported.

form local states and the shared network module. Recall from Section 2 that, to execute a handler on node N_i, the only required state is the local state of node N_i, i.e., LS^i. Therefore, the stored local states are enough to execute the handlers and we do not need to recreate the system state for that. To execute network handlers, however, we require also message (i, m) from the network (we do not need the whole network state.).

Instead of keeping a separate network state for each global state, we keep one single network state I^+ that contains all generated messages during the model checking (Figure 4.5). The execution of handlers must change to work with the shared network state I^+ (Figure 4.6). In the new handlers, H'_M and H'_A, the network state of the input global state is replaced with the new shared network state, I^+. Furthermore, the received message, (n, m), is not removed from I^+ after the execution of handler H'_M. In other words, the content of I^+ is always increasing.

It is not hard to see that the altered handlers preserve the completeness of the search: for each Transition $(L_p, I_p) \rightsquigarrow (L_q, I_q)$ in H_M, there exist a corresponding Transition $(L_p, I^+) \rightsquigarrow (L_q, I_q^+)$ in H'_M. We discuss soundness later in this section and we prove it Appendix D.

4.2.1 LMC Algorithm

Figure 4.7 presents our algorithm. Variable LS in Figure 4.7 refers to the set of all visited local states, i.e., (n,s), where n is the node index and s is the local state. Procedure `findBugs` takes the live state of the system as input,

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H'_M}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I^+ \uplus \{(n, m)\}) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I^+ \uplus \{(n, m)\} \uplus c)\end{array}}$$

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 4.6: The altered handlers in local model checking.

to initialize Variable LS at Lines 3-4. As in classic (global) model checking (Figure 2.2), the search terminates upon exceeding some bounds, such as running time or search depth (Line 5).

Handler execution. At each step of the model checking, an enabled handler, either network or local, is executed. For network handlers, the algorithm at each step checks all network messages in Variable I^+. To obtain the enabled network events, for each message e of node n in network I^+, all the currently visited states of node n are considered (Line 6). The corresponding network handler is then executed (Line 8) and Procedure `addNextState` is called on the resultant state, s', and the set of new network messages, c. Note that the messages that are added to network I^+ in this round of the loop (i.e., c in Figure 4.6) will be considered on the local states in the next round.

As in the classic global model checking approach, the node local events, such as timers and application calls, are defined based on the node local states. In other words, the value of local state LS_i^n determines which of the local events are enabled. To obtain the enabled local events, we look at all visited local states and retrieve their local events (Lines 7).

In Procedure `addNextState`, the set of new network messages is added to the shared network, I^+ (Line 12). If the state of node n has changed, it is added to set LS (Line 13). Variable *predecessors* keeps track of all the last immediate local states as well as the executed events on them that led to the current local state (Line 21). We need more than one pointer in Variable *predecessors*, since the same local state might be reached by executing different sequences of events.

Creating system states. The invariants are defined on system states. Since we do not store the system states, they must be temporarily created for the sake of invariant checking, which is performed by Procedure `checkSystemInvariant`. The procedure is called after each change to LS. Each system state ss is created by combining the local states of different nodes in LS. (We will explain in Section 4.2.2 an optimization that prevents revisiting system states.)

```
1  proc findBugs(liveState, invariant)
2    LS = emptySet(); I+ = emptySet();
3    foreach n ∈ N
4      LS^n = LS^n ∪ {liveState^n};
5    while ( ! StopCriterion )
6      if ( ∃((s, e), (s′, c)) ∈ H′_M where LS^n_s ∈ LS^n, (n, e) ∈ I+ ||
7           ∃((s, e), (s′, c)) ∈ H′_A where LS^n_s ∈ LS^n)
8        addNextState(n, s, s′, e, c, LS);
9        checkSystemInvariant(n, s′, liveState, LS, invariant);
10
11 proc addNextState(n, s, s′, e, c, LS)
12   I+ = I+ ∪ c;
13   LS^n = LS^n ∪ s′;
14   LS^n_s′.predecessors.add(s, e);
15
16 proc checkSystemInvariant(n, s′, liveState, LS, invariant)
17   foreach ss : system state
18        where ∀n_k. ss^{n_k} ∈ LS^{n_k}
19     if ( ! invariant(ss) )
20       if ( isStateSound(liveState, ss) )
21         reportBug(ss); // a bug found
22
23 proc isStateSound(liveState, state)
24   //obtain all sequences following predecessor pointers
25   foreach h : list of event sequences where
26     h^n ∈ (state^n.predecessors)* // * is closure operator
27     if ( isSequenceValid(liveState, h) )
28       return true;
29   return false;
30
31 proc isSequenceValid(liveState, h)
32   state = liveState;
33   while (∃n, nextState where state ⇝^{h^n.first()} nextState)
34     state = nextState;
35     h^n.popFirst();
36   return h == ∅;
```

Figure 4.7: Local model checker algorithm.

The only purpose of system state creation is to verify the invariant on them. Therefore, we can design invariant-specific system state creation to bypass the system states that could not possibly violate the invariant. For example, the Paxos invariant specifies that no two nodes should choose different values. In system state creation, therefore, we can ignore the local states in which no value is chosen yet. If the invariant is defined on local states separately, the invariant-specific system state creation can also bypass the system states in which none of local states have violated the invariant. For example, in RandTree distributed tree structure, one invariant specifies that in all local states the children and siblings must be disjoint sets.

Soundness verification. Since taking all combinations of local states could result into some invalid system states, the preliminary violation of an invariant could be unsound. Procedure `isStateSound`, therefore, verifies validity of the system state upon which an invariant is violated. Variable *predecessors* in each local state s' contains all the last immediate local states that led to local state s'. Following these pointers, we obtain the set of event sequences that could lead to local state s'. If a system state is valid, then there exist at least one valid combination of its local states' event sequences.[1] Line 25-26 loops on all these combinations and invokes Procedure `isSequenceValid` on each to verify them. The number of paths could exponentially increase with sequence size, which is the major cost in soundness verification.

Procedure `isSequenceValid` receives n event sequences (h^i, $i \in N$) corresponding to n nodes in the system. The procedure then looks for a valid total order for execution of the events, in which an event is executed only after it is enabled. For example, to execute a network handler that receives message m from node s, the message must first be generated by an event in s. At each step, the procedure verifies whether any of the events on top of the h^i stacks are enabled (Line 33). The first enabled event is greedily selected for execution based on definition of handlers in Figure 2.1 (the events are executed similar to a real run of the distributed system.). The loop continues until there is no enabled event on top the h^i stacks. Afterward, the fact that h is empty (Line 36) indicates that the set of sequenced events in h was possible to run and hence its corresponding system state is valid.

Procedure `isSequenceValid` returns true if and only if the corresponding input system state is valid. Appendix D provides formal arguments for the above statement. Intuitively, since an event in not popped out from h unless it is a valid, enabled event, the feasibility of executing all events implies that the corresponding system state is valid. It actually does not matter which enabled

[1] Each event sequence must deterministically lead to the same local state. If the event handler implementation is dependent on some non-deterministic values, those values must be recorded as part of the event, to be replayed deterministically on a re-execution of the event.

event is selected for the next step, since the demanded order by the sequences will be eventually enforced by receiving only the messages that are already generated.

4.2.2 Implementation Details

Our prototype implementation of the local model checking approach, denoted LMC, uses MaceMC [KAJV07], a model checker for distributed system implementations in the Mace language [KAB+07]. Mace programs are basically structured C++ implementations, in which the boundary of handlers and the protocol messages need to be specified. This helps Mace automatically generate the code for serialization and deserialization of the protocol state, and simplifies the definition of events in the model checker.

We use Dervish for online running of the model checker, in parallel with a live distributed system. The model checker is then periodically restarted from the taken snapshot.

We changed MaceMC to work only on one global object of the network simulator, i.e., I^+. To change the network handler implementations from H_M to H'_M (Figure 4.6), we changed the network simulator not to remove a message after its delivery.

MaceMC automatically generates specific functions for (de)serializing a module state in the service. We added specific functions to save and restore the whole service stack. This is required for multi-layer services such as PaxosInside [YFG10] (one of the protocols we check), which uses Paxos as its lower layer module. To efficiently check for duplicate states, we use the hash of the serialized states. For each node n, the hash of the traversed states are kept in a `set` structure. The serialized state itself is stored in a `deque` structure to benefit from its efficiency in random access.

Each message keeps track of the number of local states on which it has been executed. Therefore, in each round, each message is checked only on the newly added states, by jumping over the old states. Instead of the actual event, its hash is added into the predecessor pointers. These hash values will be checked against the hash value of the enabled events, later when we verify the soundness of the system state.

Test driver. The test in model checking a service is generally driven by an application sending requests to the service. In Paxos for example, an application sending propose requests to the service is the test driver of the model checker. The more complex the test driver, the larger the generated state space is. A careful design of the test driver could greatly impact the efficiency of model checking. In our Paxos experiments, the test driver proposes values for a particular index. The index is selected from recent chosen proposals, where not all

the nodes have learned the proposal yet. Otherwise, a new index is used for the proposal.

System states. To avoid revisiting system states, checking invariants on system states is performed only after visiting a new local state, which implies the possibility for creating new system states. For each new local state (n,s), the system states are created by iterating over the local states of all the nodes except node n and loading them. This is because the combinations of the previously visited states of node n and the local states of the other nodes have already been verified in previous rounds. It is worth noting that this optimization could make the model checking incomplete because the handler execution that has not produced a new local state, could still change the pointers in *predecessors*, which means the possibility of a valid event sequence for a previously rejected system state. To address this issue we could cache the system states in which an invariant is violated and reverify them after the changes into LS that affect them.

Beside the general approach for system state creation, we also implemented an invariant-specific variation, denoted LMC-OPT, optimized for the Paxos main invariant. In this variation, we map the local states to the values that are chosen in them. Because most of the local states have not chosen any value, lots of them will not be included in this mapping. When creating system states, we thus select only the local states that at least two of them are mapped to different values. This optimization helps avoid the creation of lots of redundant system states and consequently omits their corresponding invariant checking and soundness verification steps.

Soundness verification. Procedure `isStateSound` uses pointers in Variable *predecessor* to find event sequences that could lead to the input local states. For the sake of simplicity in implementation, we ignore the *self-references* in following the pointers in Variable *predecessor*. Although in theory this could make the exploration incomplete, in practice the search in the limited time budget is incomplete anyway and benefiting from the simplicity is, hence, preferable. Moreover, after the soundness verification on a system state is finished, some more pointers could be added into Variable *predecessor* by the process of local model checking. Therefore, a complete exploration should invoke soundness verification after each change into a *predecessor*. However, an efficient implementation of that would be complex since it should check only for the newly added pointers. For the sake of simplicity in implementation, we invoke soundness verification only after a new local state is visited.

Procedure `isSequenceValid`. The validity of a set of sequenced events could in general be checked by executing them in a simulator (the same way the global model checking approach transitions from one global state to another). If no event from the sequences is enabled in the simulator, it indicates that sequence of events is not valid. Although using the simulator simplifies the implementation,

initializing the simulator at each run of the soundness verification is expensive since it involves loading the test driver.

For efficient implementation of soundness verification module, we take advantage of the following observation. The role of the simulator in executing event e on node n is to (i) updates the local state of node n, (ii) remove the message m from the network if e is a network event for delivery of message m, and (iii) add the set c of messages, resulting from the execution of e, to the network.

The consumed message by a network event is specified by its corresponding hash in the node event sequence, which was given as a part of the input to the procedure. The set of the generated messages by an event execution can also be remembered by keeping the hashes of the generated messages in Variable *predecessor*. In this manner, the input to Procedure `isSequenceValid` is the set of sequenced events as well as the set of generated messages by each event. The execution of event e in Procedure `isSequenceValid` can then be simplified as follows:

1. A local event e is always enabled. A network event e is enabled if the hash of the required message is found in the set of generated message hashes, *net*.
2. If event e is enabled, then pop it out from the sequence. If event e is a network event, remove the hash of the corresponding message from set *net*.
3. After popping out event e, add its generated message hashes to set *net*.

The above implementation simplifies Procedure `isStateSound` to some integer comparison operations and therefore makes checking the validity of a set of sequenced events very efficient.

Local assertions. LMC checks for the system invariants defined on the system state. The source code could still be instrumented by some local assertions by which the developers have benefited in earlier stages of testing. The violation of the local assert statements in the process of local model checking could imply that either (i) the local state is invalid, perhaps because of delivering an unexpected message, or (ii) there is a bug in the system under test. Checking the latter case necessitates (i) creating all the system states by combining the local state with all local states from other nodes, and (ii) checking the validity of those states by invoking soundness verification. This approach is very expensive since it involves lots of invocation of soundness verification.

In general we could ignore violation of a local assert since a protocol bug will eventually manifest itself by violating a system invariant. Alternatively, we can discard the local state on which the assertion is violated assuming that the assert violation implies the invalidity of the local state. In the applications we tested, the assert statements were mostly used to exclude the receipt of

unexpected messages, i.e., the case that could be caused by conservative message delivery policy of LMC which delivers the message to all the local states of the destination. We, therefore, benefited from the local assert violations by discarding the corresponding local state.

Local events. The presented algorithm in Figure 4.7 is complete in the sense that, given enough time and space, it explores all possible states. In practice, however, we have a short time budget to check the reachable states from a given current state. Therefore, the developers might be interested to favor some events to be explored first in the search. Hence, in each round we put a bound on the number of local events that each node can execute; after finishing the round, the bounds are increased and the model checking is started from scratch. This approach is in spirit similar to B-DFS search, where the search depth is increased at each step.

Duplicate messages. In general, a node could infinitely issue duplicates of the same message. For example, in the verified Paxos implementations, the same Chosen message will be sent over and over to the proposer that insists for an already chosen value. To favor the main protocol messages in the limited time of search, we have put a limit on the number of duplicate messages sent from a source to a destination node. This limit is set to zero for the results reported in this chapter. Note that the duplicate messages can be postponed to be processed later, after processing some main protocol messages.

As we explained, to ensure completeness, the messages are never erased from the network object, I^+. However, if local state $s \overset{m}{\rightsquigarrow} s'$ where m is a network event, execution of m on s' is redundant since m is already executed in the sequence. To avoid redundant executions, we keep the history of the messages that has been executed to obtain the state: a network event is considered on a state only if it is not in the history of the state. After executing message m on local state s that results into local state s', we apply the two following rules to maintain the history: (i) $s'.history = s.history$, (ii) $s'.history.addLast(m)$. Thus, message m will never be executed on local state s' as well as its descendants. Maintaining history gets complicated if state s' already exists since we need to maintain separate histories for different sequences that lead to s'. We have simplified the implementation by applying rule (i) only if the state does not exist. Since the run of LMC in the limited time budget is not complete anyway, we decided to favor simplicity over completeness here.

4.2.3 Scope of Applicability

In contrast with global model checking that validity of each traversed state is ensured, local model checking optimistically allows visiting invalid states and verifies the validity of a state only after it violates an invariant. If we have a few preliminary violations, the optimistic approach of local model checking performs

better since it does not pay for ensuring validity of every single visited state. Otherwise, the cost of soundness verification dominates. For example, in online model checking, if a run of the model checker is revealing a bug in the protocol, it is likely to see lots of violation reports caused by both valid and invalid event sequences. Perhaps, one solution could be running both local and global model checker in parallel and use the result of the one that finishes sooner.

By eliminating the network element from the model checking state, local model checking reduces the explored state space since each system state is repeated in multiple global states that are different only in the network part. The larger the network state space is, the more space and time is saved by eliminating it. Local model checking is, therefore, most effective for the protocols that are chatty, i.e., exchange lots of messages to service a request. Otherwise, if the nodes rarely communicate, the changes into the network is rare and therefore there is not much to be saved by local model checking.

The current implementation of LMC assumes a best-effort, lossy network, i.e., IP. The protocols that use UDP can, therefore, be directly model checked with LMC. Although, TCP could be considered as part of the protocol stack, in practice this is not efficient and TCP is usually simulated in the model checker. To do so, LMC implementation should be also augmented to benefit from the fact that reordered messages in a connection will eventually be rejected by TCP and could, hence, be ignored, saving some unnecessary handler executions in the model checker.

4.3 Evaluation

We evaluate in this section the performance of our local model checking approach compared to a classic global one. We also illustrate the ability of our tool, LMC, in finding bugs in Paxos and its variant, PaxosInside.

We use Paxos as a complex distributed testbed to evaluate the performance of the proposed local model checking approach. In usual implementations of Paxos, each node implements three roles: proposer, acceptor, and learner. Multiple proposers can concurrently propose values for the same index. The Paxos invariant (also known as the Paxos safety property) stipulates that no two nodes will choose different values for the same index. A proposition (i.e., proposing a value for an index) starts by broadcasting prepare_request messages to the acceptors. The acceptors respond by a prepare_response message. After receiving it from a majority of acceptors, the proposer broadcasts an accept message to the acceptors. The value in the accept message is the value returned by the prepare_response message with the highest proposal number, which reflects the accepted values from previous proposals, if there is any. Each acceptor then

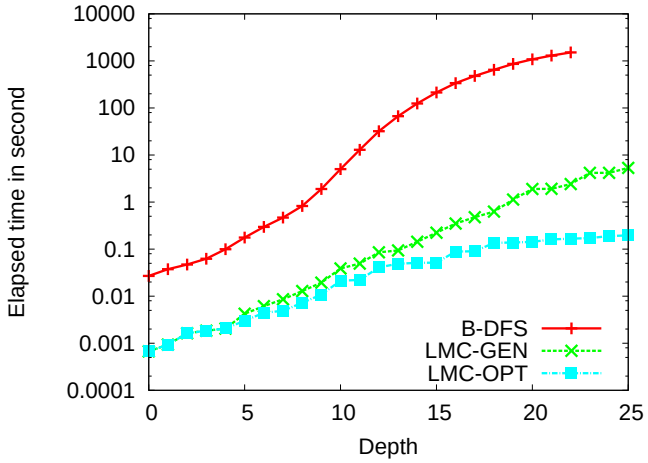

Figure 4.8: The elapsed time in model checking Paxos where only one out of three nodes proposes a value.

broadcasts a learn message to the learners. A value is chosen by the learners after receiving the learn message from a majority of acceptors.

For benchmarking purposes, we use a state space of Paxos running between three nodes, in which one node proposes a value once and the others react to this proposal by communicating using Paxos messages. The long chain of messages following each proposal could be received in a variety of orders, which all must be considered by a model checker. For each experiment, we report on evaluation of 3 algorithms: (i) B-DFS (explained in Section 2), (ii) LMC-GEN, which is the non-optimized, general version of our local model checker (LMC), and (iii) LMC-OPT, which is a version of our local model checker optimized for the Paxos main invariant according to Section 4.2.2. The experiments are run on a 3.00 GHz Intel(R) Pentium(R) 4 CPU with 1 MB of L2 cache.

4.3.1 LMC Speedup

Here we evaluate the speedup in model checking that we can get by our tool, LMC. Figure 4.8 presents the results for the example state space, in which only one node proposes a value. This state space is relatively small and yet effective in finding bugs when it is explored through an online model checker. The depth of the state space is 22 events (three initialization, one propose local event, three prepare_request messages, three prepare_response messages, three accept messages, and nine learn messages). LMC explores also longer sequences of events (up to 25) since it does not initially reject invalid sequences. The elapsed time is depicted in a logarithmic scale to illustrate exponential state space explosion problem. In B-DFS, the exponential explosion starts from the

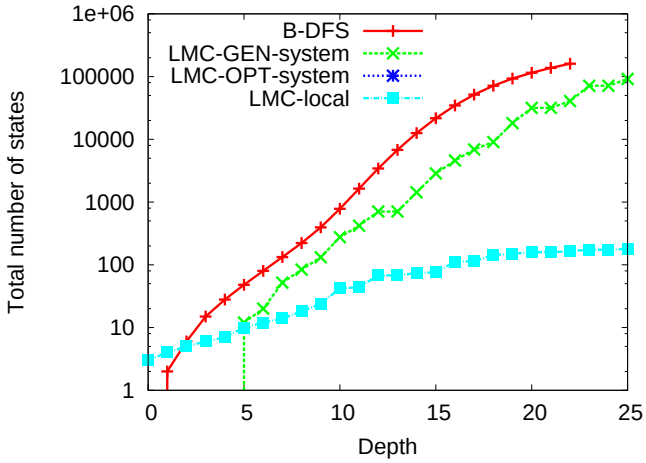

Figure 4.9: The number of explored states.

very early steps, which makes the exploration take 1514 s. The growth in LMC-OPT is much less steep, which allows it to finish the model checking in just 189 ms (∼8,000 times faster than B-DFS).

The growth in LMC-GEN, although still much more gentle than B-DFS, is steeper than LMC-OPT. The exploration finishes in 5.16 seconds which is still ∼300 times faster than B-DFS. The extra delay is due to the creation of the system states out of the explored local states, which in LMC-OPT is optimized to be performed only after a different value is chosen. Figure 4.9 depicts the number of explored states. The number of created system states in LMC-GEN, although is much less than B-DFS, is much more than the total number of local states, denoted LMC-local in the figure. LMC-OPT on the other hand drops the number of created system states to zero since there is no bug in the Paxos implementation to lead to any preliminary violations. (LMC-OPT creates a system state only if it is likely to invalidate the invariants.)

The total number of performed transitions in B-DFS is 157,332. LMC drops this to 1,186, which is ∼132 times less. This is because a LMC transition from local state LS_i^n to local state LS_j^n in node n, is redundantly executed several times in global model checking approach (once for each global state that encompasses LS_i^n and its network event is enabled).

This state space of Paxos is very useful in online model checking, where we expect the model checker to seek for a bug in the time budget of less than a minute. Both LMC-OPT and LMC-GEN can finish this state space in this duration and LMC-OPT can continue for more complicated state spaces where there is some time left (as we explained in Section 4.2.2, the model checker, in favor of time, starts with small state spaces by gradually increasing the number

of allowed local events.). This is in contrast to B-DFS that will not go further than depth 12 within a minute.

4.3.2 LMC Scalability Limits

We showed that LMC manages to finish a valuable state space in less than a few seconds. This is already good enough for practical applications such as online model checking that restarts the model checker every few seconds. From the theoretical point of view at least, it is interesting to find the scalability limits of LMC, i.e., the point where the postponed exponential explosion problem eventually manifests and makes LMC ineffective for the rest of the exploration. To this aim, we choose a much bigger state space, where two separate nodes propose two values. The depth of the state space is 41 events, which is two times the events in one error-free proposal. (LMC explores also longer sequences of events, up to 68, since it does not initially reject invalid sequences.)

Due to exponential explosion problem, neither B-DFS nor LMC could finish the state space, even after hours of running. Within this duration, B-DFS explores till 20 steps (out of maximum depth of 41) and LMC searches till 39 steps (out of maximum depth 65). The major contributer to the slow down of LMC is the expensive task of soundness verification. The number of different event histories that must be considered for checking validity of a system state exponentially increases by the search depth. In the above example that the search depth of LMC is 39, each invocation of soundness verification induces $\sim$10 seconds into the algorithm.

Invocation of soundness verification is much lower in the smaller state space in which only one node proposes a value. Running LMC-OPT on the buggy Paxos implementation (explained in Section 4.3.5) triggers the soundness verification for 773 times, and each time call takes 45 ms in average. Overall, 35,723 different event histories were checked by the soundness verification module.

4.3.3 LMC Memory Requirements

Figure 4.9 depicts the very fact that the number of local states explored by LMC is much less than the total number of system or global states. Because LMC keeps track only of local states, and the system states are created only temporarily, LMC is expected to require very low memory footprint. Figure 4.10 verifies this expectation by depicting the memory footprints of different algorithms. LMC-no-system-state denotes the run of LMC in which the creation of system states is disabled. The difference between the LMC-no-system-state and LMC-OPT (resp. LMC-GEN) indicates the memory overhead of system state creation as well as soundness verification. Although there is always a marginal

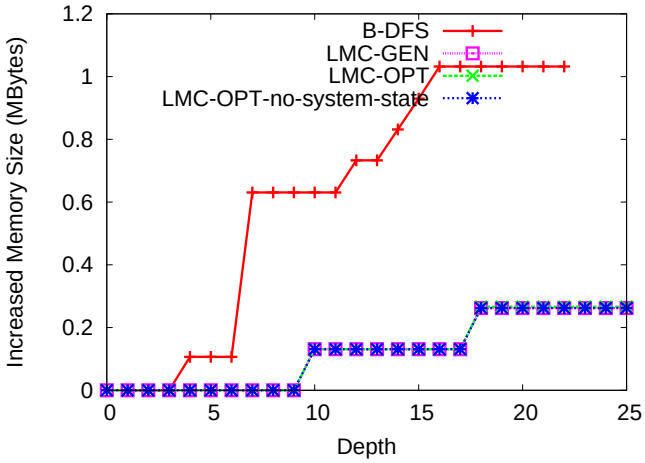

Figure 4.10: The consumed memory.

overhead for system states, the memory eventually returns to the system by reusing the deleted objects.

The consumed additional memory by all algorithms is less than 1 MB which can totally fit into the L2 cache. However, the exponential trend in memory consumption of B-DFS, promises the ineffectiveness of B-DFS for deeper searches. LMC in contrast uses the memory very efficiently (∼200 KB in total) and this amount grows linearly by increase in search depth.

4.3.4 LMC Overheads

Here we break down the overheads that limit the scalability of LMC. LMC has two major overheads: (1) creation of system states out of traversed local states, and (2) verifying soundness of the preliminary violations. To measure the overhead of each, we run LMC-OPT on the buggy implementation of Paxos, for which the corresponding bug is reported in Section 4.3.5. Figure 4.11 illustrates the overheads of LMC-OPT. In LMC-no-sound-check the soundness verification phase is disabled and in LMC-no-system-state the creation of system states is eliminated.

The difference between LMC-no-sound-check and LMC-no-system-state captures the overhead of creating the system states and checking the invariant on them. The overhead is zero until 21 steps since the unnecessary system states are bypassed by the optimization in LMC-OPT. Afterwards, the overhead increases with the depth search, because as the exploration moves forward, more local states are explored and hence more combinations of them must be considered for system state creation. The difference between LMC-OPT and LMC-no-sound-check reveals the overhead of soundness verification. (LMC-OPT did not

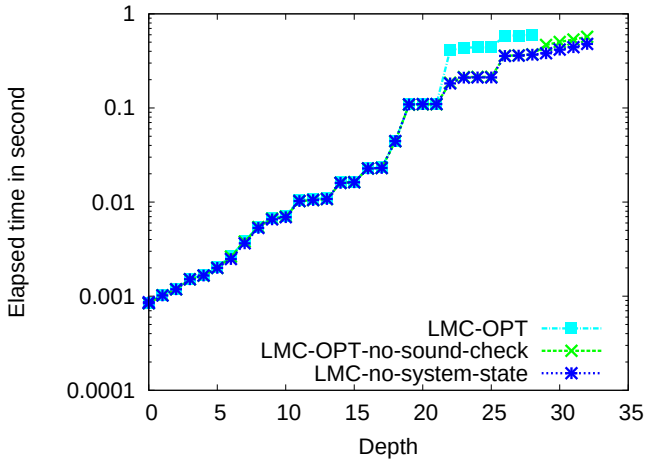

Figure 4.11: The overheads of LMC in model checking Paxos in which a bug is injected.

go further than 28 steps, the level at which the injected bug is rediscovered.) This overhead is the major contributor to the exponential increase in model checking time. The reason is that not all combinations of local states are valid, and the more local states are traversed, the more invalid system states will be checked. The violation of an invariant on each invalid state induces the cost of a soundness verification run on LMC. On the other hand, since the injected bug is close to manifest in this run of the model checker, the number of invalid combinations of node states that violate the invariant increases. LMC-OPT triggers the soundness verification for 773 times, and each call takes 45 ms in average. Overall, 427731 different event sequences were checked by the soundness verification module.

4.3.5 Testing Paxos

In this section, we report on our experiments in injecting a bug into a Paxos implementation and then running our prototype to verify its ability to detect the bug. The bug we injected was reported in a previous implementation of Paxos [LLPZ07]: once the leader receives the prepare_response message from a majority of nodes, it creates the accept request by using the submitted value from the last prepare_response message instead of the prepare_response message with highest round number. The installed invariant is the original Paxos invariant: no two nodes can choose different values.

Every one minute, the snapshot manager of Dervish takes the live system state of a running Paxos application and use that to initialize the next run of LMC. The application encompasses three nodes, each node proposes its Id

for a new index and then sleeps for a random time between 0 and 60 s. The nodes communicate using UDP and 30% of non-loopback messages are randomly dropped to allow rare states to be also created.

The bug was detected after 1150 seconds. The run of the LMC that detected the bug was initialized with the following live state: for index k_i, node N_1 has proposed value v_1, nodes N_1 and N_2 have accepted this proposal, but due to message losses only N_1 has learned it. Starting from this system state, LMC detected in 11 seconds a violation of the Paxos invariant in the following scenario: N_2 proposes a new value v_2 but its prepare_request messages is not received by N_1. N_2 responds by a prepare_response message containing value v_1, because this value was accepted by N_2 in the previous round. However N_3, since had not accepted any value for index k_i, responds back by the same value proposed by N_2, v_2. Receipt of prepare_response of N_3 triggers the bug, and N_2 broadcasts an accept message for v_2 instead of v_1. Eventually this leads to choosing value v_2 in N_2, which is different from the value chosen by N_1, i.e., v_1.

4.3.6 Testing PaxosInside

In this section, we report on running our prototype to find bugs on a variant of Paxos, denoted PaxosInside [YFG10]: this is an efficient variation of Multi-Paxos [CGR07] that uses only one acceptor. Upon failure, the active acceptor is replaced with a backup acceptor by the global leader. To uniquely identify the global leader and the active acceptor of the system, PaxosInside uses a separate consensus protocol referred to as PaxosUtility [YFG10]. The global leader and the active acceptor are identified by the last LeaderChange and AcceptorChange entries in the PaxosUtility, respectively. In this experiment, we have implemented PaxosUtility using Paxos itself. PaxosInside is more complex than Paxos for it comprises more logic. Here we use the same setup that was used for testing Paxos, with the difference that the application instead of proposing a value triggers the fault detector with the probability of 0.1 to stress the fault tolerance mechanisms of PaxosInside. In 225 seconds, the tool found one new bug in PaxosInside that we report in the following.

The bug was created because of the wrong usage of the "++" operator; if the operator is used after the operand, the returned value is the original value and not the increased one. The developer had made this mistake in the initialization function, where the leader is set to the first node of the members and the acceptor is set to the second. The used command was

```
acceptor = *(members.begin()++);
```

which makes the acceptor be the same node as the leader. The bug is of course fixed by putting the "++" operator before the operand, i.e.,

```
acceptor = *(++members.begin());
```

During the live run, node N_3 attempts to be the leader by inserting a LeaderChange entry into the PaxosUtility. At this moment, it obtains from the PaxosUtility the correct value of the active acceptor, which is N_2. After N_3 becomes leader, it proposes value v_3 for index k_i, which is accepted by the acceptor, i.e., N_2. N_2 then broadcast a learn message, which is received by N_3 as well as itself. At this point the live system state, in which all nodes except N_1 have chosen value v_3 for the index k_i, is taken to be used by LMC.

Starting from the above system state, LMC highlights the following scenario that violates the Paxos invariant: N_1, which still assumes it is the leader, proposes value v_1 for index k_i to the acceptor. Since N_1 considers itself to be the leader, according to the protocol, it does not refer to PaxosUtility to get the acceptor Id. Therefore, N_1 uses its current value, which is set to N_1, i.e., its own id, due to the initialization bug described above. N_1 accepts the proposal and sends a learn message to N_1. Upon receiving the loopback message, N_1 assumes value v_1 as chosen for index k_i. This violates the Paxos invariant since other nodes have chosen a different value, i.e., v_3.

4.4 Related Work

Cartesian abstraction. This [BPR01] is an abstraction-based verification technique where an overapproximated variant of the program is model checked, instead of the original one. Due to overapproximation, the reported bugs are not sound, which makes the technique mainly useful for correctness proving, benefiting from the completeness of the search. Malkis et al. [MPR06] achieved thread-modular model checking [FQ03,MPR10] using a Cartesian abstract interpretation of multi-threaded programs. Each thread state consists of the thread local variables plus the global variables. For each thread, the model checker separately explores possible valuations of the thread local variables as well as the global variables. The approximation comes from the fact that the valuations of the global variables by a thread is also used by other threads, ignoring the causal order for obtaining them. Again, the unsoundness, stemmed from the approximation, makes the technique inappropriate for testing purposes. In contrast, our reported bugs are sound and this is ensured by keeping track of the events executed for obtaining a local state and checking the validity of the combination of these histories after a preliminary invariant violation report.

We make use of the Cartesian product of independently explored local states to obtain the system states. Cartesian abstraction is essential here in our approach in order to create the system states and check (system-wide) invariants against them. In contrast, previous works benefited from the Cartesian abstraction by not creating system states; skipping the system states is possible since

the invariants in multi-threaded programs are just thread-local assert statements and could be verified on a local state of a thread without having the rest of the system state. Our local model checking approach employs the Cartesian abstraction in a different way: namely, to explore the system state space without exploring the global state space.

In [HJMQ03], Cartesian Abstraction is used on top of boolean abstraction of threads to find race conditions in multi-threaded programs. After boolean abstraction, each thread is represented by a long boolean expression over global and local variables including an artificially added variable for line number. A race condition is also represented by a boolean expression over the line numbers in which the threads read and write the global variables. Race conditions are detected by taking conjunction of the thread boolean expressions with race conditions. Therefore, there is no need for system state creation. This approach cannot be applied on general system invariants that would express a relation between local variables of multiple threads. The approach applies a heuristic on the detected races to eliminate some of the false positives.

One could indeed generalize the Cartesian abstract interpretation presented in [MPR06] to distributed systems, by using the network as the global object. However, the network would still be part of the model checking states, concatenated to the local states. In our approach, we exclude the network element from the model checking state and use only a shared network element.

Monotonic abstraction. Monotonic abstraction [Mit02] of the network has been used in verification of security protocols since it accounts for the maximal knowledge learned by attacker. Dolev-Yao's model [DY83] is one such model in which the attacker remembers all messages that have been intercepted or overheard. The shared network object in our local model checking approach is essentially an application of a monotonic abstraction since the delivered messages are not removed from the network. The shared monotonic network is key to ensuring the completeness of the search by applying the generated messages also on future generated local states.

Stateful vs stateless search. To avoid loops created by exploring duplicate states, it is necessary to keep track of the states visited by a model checker. Obtaining a hash of the system state requires touching the whole state once, which can be nontrivial for large states. (Although stateless approaches [God97] avoid this cost by not keeping track of traversed states, visiting duplicate states can make them very inefficient.) Thanks to Mace [KAB+07], a language upon which we implemented our tool, the relevant state of the protocol is specified by the developer and it is, hence, straightforward for MaceMC [KAJV07] to obtain its hash.

Partial order reduction. Since stateless approaches are not able to avoid loops, specific techniques are required to tackle the exponential explosion prob-

lem. Partial Order Reduction (POR) techniques [God96] prune the state space of a concurrent system to avoid unnecessary interleaving of events. The performance of B-DFS (used in our benchmarks) could potentially improve by implementing such technique. However, we expect the improvement would be marginal because of frequent changes into the global state; transmitting any message would change the network state and consequently the global state. Moreover, lots of redundancies avoided by POR-based techniques are already avoided by duplicate state detection in B-DFS. To the best of our knowledge, no study has compared the performance of stateful searches with POR-based techniques in distributed systems.

Furthermore, any application of a POR-based technique to model checking distributed systems would be incomplete since it would not account for system-wide invariants. For a set of local states, POR explores only one valid combination of them among all possible valid Cartesian products. It is useful in multi-threaded programs since the assert statements are defined on thread-local states and visiting a local state once in a combination with any other local states is enough. In contrast, we test the system against system-wide invariants such as Paxos main invariant, which could be held in one combination of local states and violated in another.

4.5 Summary

In this chapter, we have introduced a novel, *local* approach to model checking distributed systems. Essentially, the underlying idea is to remove the network state from the global state when model checking, and focus on the remaining system state, which is the usual required part for invariant checking. The system state is itself built temporarily out of local states, and these are maintained separately. Although complete, the approach is not sound in the sense that some system states could be invalid, i.e., could not have been produced by an actual run of the system. We check the soundness of the system state, a posteriori, only if an invariant is violated.

By removing the network from the global states, our local model checking approach creates much less system states than in the global approach. In addition, and in contrast with the latter approach, in which visiting the system states is an inherent part of the exploration process, our local approach separates the exploration of transitions from the actual creation of system states. This makes it possible to exploit the specificities of the user-specified invariants and a priori eliminate all system states on which these invariants cannot be violated.

We cannot solve problems by using the same kind of thinking we used when we created them.

Albert Einstein

5 PaxosInside

The consistency of cached data in manycore systems is usually guaranteed by the hardware. Although this approach simplifies the software design, recent studies show that it does not scale to a large number of cores [BBD+09]. An alternative approach has been recently proposed, where the cores are viewed as nodes of a distributed system [BBD+09] on which critical information is *explicitly* replicated. Both the applications (at the user level) and the kernels (at the operating system level [BBD+09]) ensure the consistency of the replicated data, by explicitly exchanging messages to implement an *agreement* algorithm.

Barrelfish pioneered this approach by implementing a multikernel model where the capability service is replicated on several cores [BBD+09]. These cores exchange messages to execute a 2PC (two phase commit) agreement algorithm [LS79], which ensure the consistency of the replicated state among the kernels. The advantage of a 2PC is its simplicity. The drawback however is its fragility: 2PC is *blocking* in the sense that, to progress, it requires responses from all the nodes. As a consequence, the whole system slows down if even a single core lags behind, which can easily be caused in a manycore system by an unpredicted load or consecutive cache misses. Upon a cache miss, loading the data from the memory takes around 100 ns [1], i.e., ∼10 times longer than loading data from cache. If the data is swapped out to the hard disk by the virtual memory manager, the core has to wait till the corresponding memory page is swapped into the memory, which takes around 8 ms, i.e., ∼800,000 slower than a cache access. The process context switch latency is between 10 and 20 μs in

[1] The memory access time is highly dependent on the memory architecture and can range from 50 ns to 150 ns.

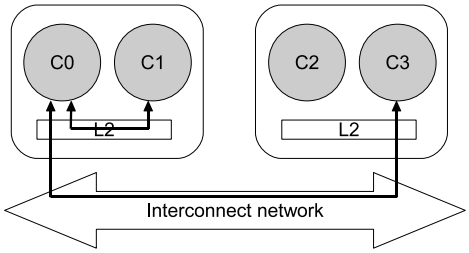

Figure 5.1: Non-uniform latency in inter-core communication; Cores C_0 and C_1 share the same L2 cache and communicate much faster than Cores C_0 and C_3 that have to go through the interconnect network.

average and can take much longer because of page faults. Moreover, the inter-core latency is sometimes non-uniform. For example, as depicted in Figure 5.1, the cores located on the same CPU share the same L2 cache and hence can communicate much faster than the cores located on different CPUs. In short, a protocol that waits only for the first response is more desirable than one that waits for all.

Non-blocking agreement protocols, also called *consensus* algorithms, on the other hand, can progress with responses from only a majority of replicas [Lyn96, GR06]. The model underlying message passing consensus usually considers the *crash* failure of a minority of nodes, as well as arbitrary long delays in the communication between the nodes. Such *asynchrony* makes it impossible to distinguish crashed nodes from delayed responses, and force consensus algorithms to progress as long as a majority of nodes are responsive. The combination of the very notions of crashes and asynchrony models pretty well the communication scheme underlying manycore systems with non-uniform communication latency and unpredictable slow cores.

A family of practical consensus protocols has been recently developed. These include Paxos [Lam98], Multi-Paxos [Lam01a, CGR07], Cheap-Paxos [ML04], Fast-Paxos [Lam06] and Mencius [MJM08]. Multi-Paxos [Lam98] is considered one of the most efficient such protocols; it has been implemented in a wide variety of IP network settings [CGR07, LLPZ07, Bur06, JKBK+08]. Although initially designed to be effective in distributed systems, none of these algorithms meet the new requirements of a manycore setting. This is basically because, although it looks alike, a manycore is not a genuine distributed system in the classical sense. The major challenges are the bandwidth of the interconnect network being a scarce resource on the one hand, and the large number of messages typically exchanged in consensus algorithms on the other hand. Besides, many such algorithms, e.g., Multi-Paxos, require that a request goes through a specific node that leads the consensus execution: this *leader* is reportedly a bottleneck [Bur06] that can significantly hamper scalability in a manycore

setting. Similarly, since the messages transmitted in a manycore setting are eventually placed in each core's cache, a high load on a core will make it run out of space in cache, inducing frequent cache misses, which in turn negatively impact performance. This is crucial in manycore systems where the lower latency of inter-core communication, compared to genuine distributed systems, promises a higher rate of requests to be received by the cores.

In this chapter, we explore, for the first time, the feasibility of implementing a consensus algorithm in a manycore system. We present PaxosInside, a consensus algorithm that takes up the challenges of the manycore environments. Very intuitively, PaxosInside was designed with the specific aim to reduce the consensus-related traffic in general, and more specifically that of a leader. A key insight underlying PaxosInside is the observation that the role of *acceptor* in consensus, i.e., to resolve conflicts among possibly multiple leaders, can be played by a single node. [2] Making use of a single acceptor introduces some technical challenges that we address in the thesis. This leads to much less traffic, yet with jeopardizing neither the consistency nor the general availability of the system. By using three cores, the system can progress even with one slow core, just like in the Paxos-family of protocols. The trade-off with a higher replication degree is that, to progress, PaxosInside requires at least one of the leader or the active acceptor to be responding.

We report on the implementation and evaluation of PaxosInside on four 2.4 GHz Dual-Core AMD Opteron(tm) processors (8 cores in total). This part was itself technically challenging because of the lack of any experience on implementing a message passing consensus algorithm in a manycore setting. For instance, the latency of context switching after receiving a message, which is negligible in classical distributed systems, becomes a serious overhead in manycore systems. In our implementation of PaxosInside, we eliminate the cost of system calls by avoiding lock-based synchronizations as well as delivering the messages via user-level threads.

We convey the efficiency of PaxosInside by measuring (1) the scalability of PaxosInside with the number of cores; and (2) the performance of PaxosInside compared to 2PC when a core becomes slow. PaxosInside is scalable to a maximum number of clients in our setting, five [3], whereas Multi-Paxos and 2PC are scalable to only two and one clients, respectively. PaxosInside can progress with slow cores and in worst case scenario where the leader is slow, PaxosInside replaces the leader and continues with the same rate, whereas 2PC blocks as long as even one node is not responding.

Section 5.2.3 has already recalled the design of Paxos and dissects the role

[2] The presence of multiple leaders can be caused by asynchrony: a new leader might be elected if the former leader is non-responsive, even only temporarily.

[3] Our experiments make use of a machine with eight cores, from which three cores are allocated to the replicas and the other five are free for clients.

of each Paxos participant. The rest of this chapter is organized as follows. Section 5.1 gives the key insight underlying PaxosInside and illustrates the differences between PaxosInside and the Paxos family of protocols. The detailed design of PaxosInside is presented in Section 5.3, and its implementation in a manycore system in Section 5.4. We present our experimental results in Section 5.5. Section 5.7 summarizes the chapter with some final remarks. Appendix B presents the pseudo code of PaxosInside which is followed by the correctness proofs of PaxosInside in Appendix C.

5.1 PaxosInside: The Main Insight

Blocking agreement algorithms that have been used so far to ensure the consistency of replicated data among multiple cores [BBD+09] suffer from the variant response latency of cores, which is unpredictable in manycore systems. Consensus algorithms, which are originally designed to tolerate crash failures, can also be employed to efficiently tolerate cores that do not respond in time. In the manycore fault model, a faulty core is one from which we have not received a response in time.

We present a consensus algorithm that meets the requirements of a manycore system, namely limited bandwidth in the interconnect network and limited cache size of the consensus leader. A major specificity of our algorithm, PaxosInside, is the use of only one active acceptor at a time. In the following, by comparing PaxosInside with Multi-Paxos, which is the most efficient variation of Paxos used in practical settings [CGR07], we show how this design decision reveals appropriate in a manycore system.

Figure 5.2 depicts message transmission in a collapsed Multi-Paxos setup that consists of three nodes. The messages that cross the node boundary must be included in the total number of messages. The following equation captures $\text{Msg}_{multi-paxos}$, the total number of exchanged messages between nodes in a normal Multi-Paxos instance:

$$\text{Msg}_{multi-paxos} = (A-1).(A+1) \tag{5.1}$$

Here, A is the number of acceptors. Then, for the usual setup of three nodes, this value would be equal to eight in Multi-Paxos as opposed to four in PaxosInside.

The total number of messages affects the overall consumed bandwidth between nodes. A crucial parameter is the number of sent/received messages by the leader node, $\text{Msg}^{leader}_{multi-paxos}$. The leader exchanges more messages compared to the other nodes and hence, when it gets saturated, the system cannot process more client commands. This is reportedly a problem for the scalability of Multi-Paxos [Bur06]. Typically, each node plays all the Multi-Paxos roles and hence

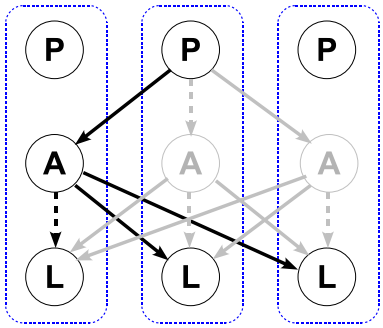

Figure 5.2: The reduced number of messages in PaxosInside compared to collapsed Multi-Paxos deployed on three nodes. The dotted box represents the node boundary. The dashed messages, which do not cross the node boundary, do not consume the node bandwidth. P, A, and L represent the proposer, acceptor, and learner roles, respectively. The grayed acceptors and consequently the communications to/from them are eliminated in PaxosInside.

the leader node is also a learner as well as an acceptor. Thus, the total number of messages exchanged between the leader node and the other nodes is:

$$\mathrm{Msg}^{leader}_{multi-paxos} = 3.(A-1) \tag{5.2}$$

Again, for a setup that includes three nodes, this number is equal to six. PaxosInside reduces this number to three by using only one acceptor.

One interesting variation of collapsed Multi-Paxos that we could have considered uses fewer acceptors. In such a case, fewer messages would be exchanged since some of the acceptors would not be active. For example, in the common setup with three nodes, if Multi-Paxos uses only two of them as acceptors, the number of exchanged messages by the leader would be four per command, as opposed to six, which is less than the improvement by our algorithm. Using fewer proposers and learners will reduce the availability and reliability/scalability of the system, respectively. Therefore, we do not compare PaxosInside with such variations.

As we explained in Section 5.2.3, the availability of the acceptor role can be provided in different ways. One approach, which is taken by Multi-Paxos, is the replication of the acceptor. A side-effect of this approach is the increase in the number of exchanged messages between acceptors and other roles. An alternative approach is to rely on *backup acceptors*, and replace the failed (or suspected to be failed) acceptor with a new fresh one from them. The backup acceptors do not participate in the normal execution of the algorithm and do not, hence, increase the message complexity of the algorithm. This idea is the main insight underlying PaxosInside, which reduces the number of exchanged messages between nodes by a factor of two. Although the use of backup acceptors addresses

the problem of the acceptor availability and yet provides better performance, poses the non-trivial problem of reliability of acceptor's data, which we discuss now.

Recall that the acceptors also keep a few data, which is necessary during the short-term period of a single Paxos instance to address the possible contention between multiple proposers. Missing this data, by switching from the active acceptor to a fresh backup acceptor in the middle of a Paxos instance, can violate system consistency. For instance, if the active acceptor promises not to take any proposal number less than pn, then a fresh new acceptor would not be aware of this promise and might accept proposal numbers less than pn. Nevertheless, if the proposers get properly notified of this data loss, they can safely restart the Paxos instance without risking the algorithm integrity. For example, upon receipt of the failure notification of the active acceptor, the proposers know that the promised sequence number by the previous acceptor is no longer held.

We will explain in Section 5.3 that, if we assume that the leader and the active acceptor nodes do not fail at the same time, then there exists a process in which the leader can safely notify the other proposers of the active acceptor switch. This is the same assumption that is already made by Paxos in the common setup that consists of three nodes implementing three proposer, three learner, and one acceptor roles. By carefully placing the proposer and acceptor roles among the nodes, in a way that the leader and the active acceptor are placed in two separate nodes, we can make the assumption that the leader and the active acceptor do not fail at the same time. The violation of this assumption cannot occur unless two of the three physical nodes fail. In this case, we would be left with one node which is less than the minimum required nodes for Multi-Paxos to progress ($min > total/2$).

5.2 Preliminaries

We discuss in this section the role of message passing agreement in manycore systems. We also give a brief overview of 2PC [LS79], as an example of a blocking agreement protocol, and Paxos, as an example of a non-blocking one.

5.2.1 Manycore Systems

A major scalability bottleneck in manycore systems is induced by the need to keep the cached data consistent among multiple cores. The developers expect to have the same view of data, independently of which core the processes are running on. However, two cores might have loaded the same data into their caches or local memory, and changes into one of them, hence, is not by default observable by the others. This gap, between the centralized view of the pro-

cessor and the distributed implementation inside manycore systems, is typically filled with hardware techniques, known as cache coherence protocols. There are different kinds of such protocols but the bottom line is that, after a change into a memory address by a core, all cores that have loaded the same address are notified about the change, before doing any computation on that data. For a large number of cores, this implies long delays for change propagation and/or a large number of synchronous inter-core message transmissions. This is because the hardware has to consider the possibility that any core might have already loaded the data, while it cannot make any assumption about the actual time each of the cores might actually need the updated version of the data. In other words, the hardware does not know precisely *where* and *when* the updates must be sent and must be, therefore, general enough to cover all the possible cases.

An alternative software-based approach has been recently proposed [BBD+09]. According to this approach, the software handles the consistency of its own data by viewing the entire machine as a large distributed system of which nodes represent the actual cores. If the software assigns two separate cores to process the same data, each core assumes its own copy of the data, i.e., replica. It is then the software's responsibility to maintain the consistency of the data by exchanging messages to run an agreement algorithm among the cores running the replicas. The advantage of this approach is that messages are transmitted between the cores only when it is necessary for software consistency, and the software, in contrary to the hardware, has full knowledge about when and to where these messages are necessary to be sent. This approach can be applied to both the operating system and the application layers. Recent work [BBD+09] has applied the approach to the kernel layer and showed good scalability.

To ensure the consistency of the state replicated among the cores, an agreement algorithm needs to be executed among the cores. In Barrelfish [BBD+09], the capability system is replicated on the cores and a 2PC (two phase commit) algorithm keeps the replicated state consistent among the kernels. The algorithm does not make any synchrony assumption about message transmission (they can take arbitrarily long) and it guarantees safety even if cores are arbitrarily slow.

5.2.2 Blocking Agreement

Similar to common practices in distributed systems, the user can replicate its data over multiple cores to increase its availability and scalability. The replicated data can be available through other replicas if some of the cores are slow. Moreover, the computation load and access bandwidth can be split among the replicas, making the system more scalable. To ensure the consistency of the state replicated among the cores, these need to execute an agreement protocol.

In Barrelfish [BBD+09], the capability system is replicated on the cores and a 2PC (two phase commit) algorithm [LS79] keeps the replicated state consistent among the kernels.

The 2PC algorithm, as its name suggests, has two phases. In the first phase, the coordinator (the leader) sends a prepare message to the replicas. Each replica locks its local copy of data and responds with an ack message if it is not already locked by another coordinator. The coordinator starts the second phase by broadcasting a commit message to the replicas, only if it receives an ack from all of them. In this case, each replica executes the command of the commit message and releases its lock, which is followed by a commit_ack message back to the coordinator. Otherwise, the coordinator broadcasts a rollback message to the replicas. Upon receiving a rollback message, each replica releases its lock if it is already acquired by the corresponding ack message.

The 2PC algorithm, however, is blocking: it requires responses from all the nodes to progress. As a result, the whole system slows down if even a single node lags behind, which is a likely scenario in manycore systems since a core could be slowed down by an unpredicted load or consecutive cache misses. The computing power of a core is shared by the operating system among the processes. This is necessary to make effective use of the computing unit, especially when the running process does not use the computing unit after a blocking I/O. An unpredicted load of processes on a core causes any other replica process running on that core to receive less shares of the core and consequently to respond later to the messages. Furthermore, the data of a process could be cached by the core, be located in the memory, or paged out to the hard disk by the virtual memory manager. The access time to these locations varies from 10 ns up to 8 ms, i.e. ∼800,000 times difference. This time difference can affect both replica processing time as well as the operating system context switch time.

All the above factors contribute to the unpredictability of a replica processing time. Besides, because of the non-uniform latency of communication between the cores, it is more desirable to progress after receiving the response from the closer cores. Figure 5.1 depicts one scenario where a core can communicate faster with the core with which it is sharing the L2 cache.

5.2.3 Consensus

Unlike blocking agreement algorithms, (asynchronous crash-resilient) consensus algorithms require responses from only a majority of the nodes to progress. Whereas crashes are considered common in classical distributed systems, in a manycore environment, these model slow cores. Asynchrony, on the other hand, models the tolerance to delayed messages. We recall below the celebrated Paxos protocol [Lam98] and its Multi-Paxos optimization [CGR07].

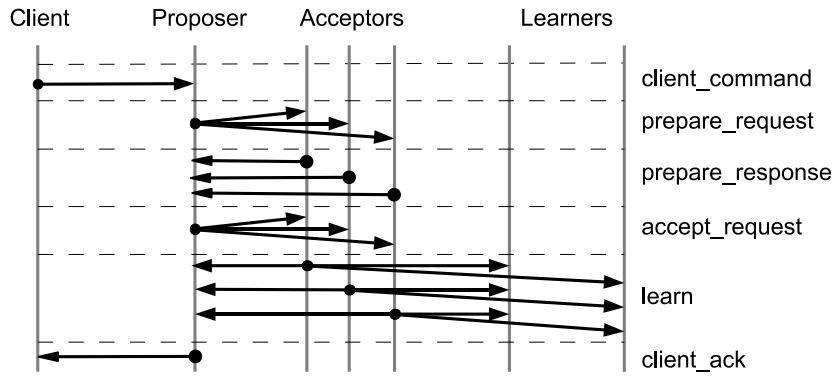

Figure 5.3: The interaction between nodes in Basic-Paxos. This example consists of one proposer, three acceptors, and two learners. In Multi-Paxos, the leader skips the first phase, i.e., prepare_request and prepare_response.

The challenge addressed by these consensus algorithms is that of ensuring the consistency of the replicas, while assuming that nodes hosting the replicas can crash (can be arbitrarily slow) and without making strong assumptions about the synchrony of the network. For instance, two issued commands by the clients could reach two nodes in the inverse order and that would cause inconsistency between the system states in those two nodes. Paxos was proposed by Lamport [Lam98] to address such challenges. It assumes a shared service (data and its associated operations) to be implemented as a state machine, replicated on multiple nodes. It gives an order to the issued commands by the clients and guarantees that all nodes execute the commands in the same order. Note that the agreed order is not necessarily according to the time the commands have been issued by the clients. In other words, if a client issues a command C_1 before another client a command C_2, the algorithm guarantees that they will be applied in the same order on all nodes: either as C_1-C_2 or as C_2-C_1.

Each node in Paxos writes some data into a persistent storage such as hard disk. After a crash, a node cannot participate in consensus until it recovers the data that has been written into its persistent storage. The delay of writing into persistent storage could become a bottleneck in some settings.

Basic-Paxos

We now give a brief description of the original Paxos algorithm [Lam98], which we call Basic-Paxos hereafter in this thesis. Basic-Paxos was first presented in [Lam98] and was further explained in [Lam01b]. The participant nodes in Basic-Paxos implement three different roles: *proposer*, *acceptor*, and *learner*. The proposers advocate the client commands, the acceptors resolve the con-

tention between multiple proposers, and the learners learn the chosen values. The *leader* orchestrating the consensus is chosen among the proposers.

The ultimate goal of Basic-Paxos is to assign orders to client commands. The order of a client command, which is called a value in the Paxos terminology, is specified by an instance number. To assign values to instance numbers, Basic-Paxos requires two phases. In the first phase, a proposer attempts to become leader for a particular instance number by broadcasting a prepare_request message to the acceptors. Upon receipt of a prepare_response message from a majority of acceptors, the proposer becomes the leader of that instance number. In the second phase, the leader proposes a value to the acceptors and the acceptors broadcast the corresponding message to all the learners. A learner learns the proposal after receiving the message from a majority of acceptors. All message transmissions related to a particular order constitute a separate *instance* of Basic-Paxos. The interaction between nodes is depicted in Figure 5.3.

We now explain one instance of Basic-Paxos, step by step:

1. client_command: The client_command message, which comes from a client to proposers, contains a command from the client. The proposer then advocates the client command.
2. prepare_request: The proposer first picks an order for the client command, which is called Paxos instance number. Then, it tries to take the leadership position and asks the acceptors to recognize it as such by broadcasting a prepare_request message, which contains a proposal number pn. The proposal number distinguishes different attempts of the proposer for the same instance number.
3. prepare_response: Upon receipt of a prepare_request message, each acceptor checks the proposal number. If the proposal number, pn, is greater than the proposal number of the previous accepted proposals, the acceptor sends a prepare_response message back to the proposer. By that it promises not to accept any proposal number smaller than pn. The highest proposal number must be stored in a persistent storage. If the acceptor has already accepted a value, the value will be included in the prepare_response message.
4. accept_request: After receiving the prepare_response messages from a majority of the acceptors, the proposer assumes itself as the leader. It first decides on a value; one selected from values received from a majority of the acceptors if they have already accepted a value, or any value otherwise. It then sends to all the acceptors an accept_request message with the proposal number pn and the proposed value.
5. learn: When an acceptor receives the accept_request message corresponding to the promise it has made, it accepts the proposal and broadcasts a

learn message to all the learners as well as the proposer. The accepted value must be stored in a persistent storage.

6. client_ack: When a learner receives the learn message from a majority of the acceptors, it recognizes the proposed value as chosen and can inform the clients with a client_ack message. Alternatively, this can be done by the proposer that advocates the client command.

Although each role can be implemented by a separate node, usually a single node implements all the three roles, which is then called collapsed Paxos. The advantage is that the transferred messages between two roles that are located on the same node do not cross the node boundary and thus consumes less bandwidth. According to the liveness property of Basic-Paxos [Lam06] a value will be eventually chosen, given that enough nodes are running. For example, in collapsed Paxos deployed on three nodes, the liveness property holds as long as two of the three nodes are running. Basic-Paxos guarantees the following two safety properties [Lam98]: (i) non-triviality: only the proposed values can be learned; and (ii) consistency: two different learners cannot learn two different values.

Multi-Paxos

After a proposer takes the leadership position for one instance in, it could be more efficient if it assumes this position for the next Paxos instances in' ($in' > in$) as well. The other proposers can still try to become leader, when they suspect that the last leader has failed. Multi-Paxos [Lam01b] is the version of Paxos which implements the mentioned optimization.

The first round is similar to Basic-Paxos. When a proposer P becomes leader, it uses the same proposal number pn for the next Paxos instances. Hence it can skip the first phase of Basic-Paxos, i.e. prepare_request, and start directly with the accept_request message. If in the meanwhile, another proposer P' tries to become the leader with a higher proposal number pn', then the proposal number of P will not be the maximum proposal number any longer, and its accept_request messages will be rejected. Proposer P can then either relinquishes the leadership position to proposer P' or try to become the leader again by sending a prepare_request message with a new proposal number.

The Roles in Paxos

To understand the idea underlying our PaxosInside algorithm, it is important to take a closer look at the different roles in Paxos. This is essential to understand the rationale behind the design of the proposed protocol, PaxosInside. As men-

tioned before, there are three major roles in Paxos: (i) proposer, (ii) acceptor, and (iii) learner.

The proposer role is to advocate the client command. This is essential for the scalability of the system. If the clients have to be involved in the consensus execution, e.g., by advocating their own request, the system could not scale with the number of clients. By relinquishing this task to the proposers, the consensus is required among only a few nodes and thus it is more scalable with the number of clients.

The learner is the actual long-term memory of the system. When a Paxos instance is finished successfully and its value is learned, this value is kept in the multiple available learners. The clients then can read this value from each of the learners.

The acceptor is the main role in Paxos that the safety property of consensus. If multiple proposers want to propose values for the same Paxos instance, the acceptor is key role to resolve the contention between the competing proposers. Suppose some acceptors accept value v_0 from proposer P_0 and, for some reasons, the Paxos instance does not complete successfully. Now, to finish the instance, proposer P_1 must first read the *accepted value* by the acceptors (i.e. v_0) and propose the *same value*. It implies that the acceptors play the role of the short-term memory for the system; they must remember a few values during the short period of one Paxos instance.

Replication in Paxos

At the heart of the efficiency of PaxosInside algorithm, lies the observation that replication is used for different purposes. In general, we have two types of replication: (i) replication of service and (ii) replication of data. Replication of service increases the availability of the system. In other words, when a client requests for the service, we want to make sure that there is at least one responding node, ready to receive the client commands. The replication of data, however, is for increasing the reliability of the system. In other words, it decreases the chance of data loss by missing some nodes (after permanent failures).

The roles in Paxos are replicated, but each one for a different purpose. The replication of the proposers is to increase availability, as the proposers provide a service to the clients, i.e., advocating their request. In contrast, the learners store the data of the system, and the purpose of their replication is to enhance reliability. [4]

[4] From performance perspective, one can take advantage of replication to increase scalability as well. For example, Mencius [MJM08] uses proposer replication to enhance the scalability. Moreover, if the application does not demand the very last state of the system, its read traffic can be directly serviced from either of the replicated learners.

The acceptor replication is partly for service availability and partly for data reliability. The proposers start the consensus process by contacting the acceptors. Thus, they require the provided service by the acceptor role to be available. In addition, as mentioned before, there are a few data kept by the acceptors such as the accepted value and the promised proposal number, which should be kept during the Paxos instance. However this data is required only for the active Paxos instance, and in the case of failure, we can think of some workaround solutions.

The main insight in design of PaxosInside, which will be explained later in Section 5.1, comes from the following observation: the replication of the acceptor role is mainly for availability, and if its availability is provided via other mechanisms, then the replication of acceptor is no longer necessary.

5.2.4 Failure Model

Our failure model assumes that the nodes, i.e., cores, could be non-responsive for a period of time. This models cores that are running slow because of contention in using shared resources, e.g., the CPU cycles, the cache, and the hard disk (after a page fault). The consensus protocol makes progress as long as a majority of cores are not slow. We also assume that the process running on the core could crash. However, to be able to benefit from consensus in tolerating the process crash, the process crashes should be made independent, e.g., via using n-versioning. This chapter, therefore, focuses only on slow cores as the target failures.

5.3 PaxosInside: The Algorithm

We explain in this section the PaxosInside algorithm in detail. As mentioned in Section 5.1, the idea is to use only one active acceptor and ensure availability via some backup acceptors. Care must be taken to also provide reliability for acceptor's data when the active acceptor is replaced. We first start this section by describing the communication scheme underlying PaxosInside in the failure-free case where messages are received in a timely fashion. Then we discuss the backup cases executed when the cores are faulty. In manycore systems, a faulty core is slow and does not respond in time. For example, a failed leader does not responding to client requests in a timely manner.

5.3.1 The Failure-free Case

The roles in PaxosInside and the interaction between them is depicted in Figure 5.2.

1. Proposer P decides to take the position of the leader. It first obtains the Id of the active acceptor, A (we will explain the process of obtaining this Id in the next subsection), and sends a prepare_request message including a proposal number, pn, to acceptor A. By doing so, the proposer asks the acceptor to recognize it as the leader.
2. If the proposal number, pn, is greater than all the previous proposal numbers received by the acceptor, it sends a prepare_response message back to proposer P. By doing so, the acceptor promises not to accept any proposal number smaller than pn.

   Notice that these two steps are necessary only the first time a proposer contacts the acceptor. After that, the proposer becomes leader and skips these two steps.
3. Proposer P then sends an accept_request message including the proposal number pn as well as a proposed value, to acceptor A.
4. When acceptor A receives the accept_request message corresponding to the proposal number, to which it has given its promise, it accepts the proposal and broadcasts a learn message to all the learners.

5.3.2 Switching Acceptor

Here we consider the scenario in which active acceptor A fails (does not respond in a timely manner) and the leader replaces it with another backup acceptor A'. It is worth noting that although PaxosInside is correct with crash faults, in the fault model of our manycore setting, a faulty core, although still working, does not respond in time.

When the active acceptor fails, the leader is the only node that is allowed to replace it with another backup acceptor. This change, however, must be confirmed by a majority of nodes. This is necessary to avoid having multiple instances of active acceptors running in the system, and consequently compromising consistency. The scenario is illustrated in Figure 5.4.

Obtaining the confirmation of a majority of the proposers is a separate consensus problem which can be solved by any Paxos-like algorithm. Although it is possible to merge this consensus into the main operation of PaxosInside, for the sake of simplicity of presentation, we assume that the consensus over the new active acceptor is achieved by a separate basic implementation of Paxos, which hereafter is called PaxosUtility. Notice that PaxosUtility instance which handles consensus over the new active acceptor is totally separate and independent from PaxosInside algorithm that we are explaining here. Moreover, running PaxosUtility does not require any extra nodes; it runs on the same nodes as PaxosInside.

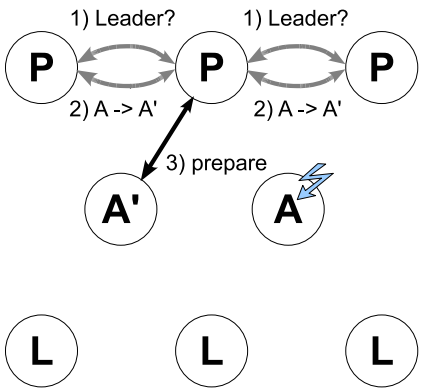

Figure 5.4: The interaction between nodes in PaxosInside to replace failed acceptor A with another backup acceptor A'. In Step 1, the leader makes sure that it is still known as the leader by a majority of the nodes. Then in Step 2, it announces the change of the active acceptor. Finally in Step 3, it sends a prepare_request message to the new active acceptor A'.

Beside the Id of acceptor A', the leader also includes the uncommitted proposed values into the message sent to the PaxosUtility. This is to cover the cases where acceptor A has received an accept_request message with value v_{in} for instance number in, but the corresponding issued learn message is not received by the other nodes yet. In this way, it guarantees that the next leader will try to propose the same value as v_{in} for instance number in.

The leader after finishing the consensus over the active acceptor, switches from acceptor A to acceptor A', i.e., the new active acceptor. Because the acceptor node has changed, the leader must start over with a prepare_request message to take the leadership of the new acceptor.

5.3.3 Switching Leader

In principle, every proposer could spontaneously try to take the leadership position by sending a prepare_request message to the acceptors. In practice, this usually happens when the current leader is non-responsive (i.e., fails). In PaxosInside also, when the leader fails, any proposer can try to take its position by sending a prepare_request message to the active acceptor. Assume that proposer P' suspects the failure of leader P and decides to become the leader. The active acceptor Id, A, can be obtained by inquiring a majority of the nodes. This is due to the fact that the last leader does not use the new active acceptor unless it obtains agreement from a majority of nodes. The sequence of messages is demonstrated in Figure 5.5.

Care must be taken to ensure that, in the meanwhile, the active acceptor A

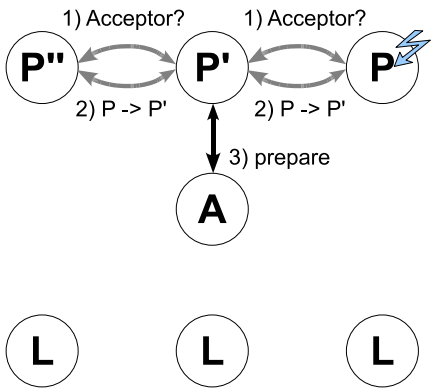

Figure 5.5: The interaction between nodes in PaxosInside when proposer P' takes the leadership position from leader P. In Step 1, proposer P' inquires for the active acceptor Id. It then announces itself as leader in Step 2. Finally in Step 3, it sends a prepare_request message to the active acceptor.

is not replaced by the last leader. Otherwise, we end up with two leaders which use two different active acceptors. To this aim, proposer P' uses PaxosUtility to start a consensus instance in which proposer P' announces that it is going to take the leadership position by assuming A as the active acceptor. Accordingly, every leader must always check for this announcement before switching the active acceptor. If the leader observes this announcement, it must consider its position as relinquished. This step is marked as Step 1 in Figure 5.4.

5.3.4 Switching both Leader and Acceptor

If the active acceptor fails, the leader is in charge of replacing it with a fresh backup acceptor. On the other hand, if the leader fails, then any proposer can safely take its position, given that the active acceptor is still running. The only remaining case to handle is when both the leader and the active acceptor fail together.

As mentioned in Section 5.1, to handle this scenario we carefully assign the PaxosInside roles to the nodes in a way that the leader and the active acceptor are located in two separate nodes. Assume that we have N nodes available and each node implements all the roles: proposer, acceptor, and learner. In PaxosInside that there is only one active acceptor, we have the option to pick the node that will also play the active acceptor role. This deployment is demonstrated in Figure 5.2. The idea is to assign the active acceptor and leader roles to two separate nodes. In this way, the failure of the leader and the active acceptor cannot occur together, unless two of N nodes fail at the same time.

Notice that, in the usual setup of consensus, which consists of three nodes,

this failure scenario implies that two of the three nodes are failed. On the other hand, consensus algorithms, including Paxos family, cannot progress with just one running node out of three. Consequently, we can assume that if the failures of the leader and the active acceptor occur at the same time, there is only one node left. In this situation, neither Paxos family of protocols nor PaxosInside can progress.

It is worth noting that, for $N > 3$, the failure of the leader and the active acceptor at the same time does not jeopardize the consistency of the system. It only slows down the progress of consensus, as these two cores are slow and hence respond slowly to the recovery messages. Nevertheless, failure probability of two particular nodes, i.e. the leader and the acceptor, is much less than failure probability of two arbitrary nodes, which makes this failure scenario very rare. For example, if the failure probability of a core is s, then the failure probability of two particular nodes is s^2, and the failure probability of two arbitrary cores is $\binom{N}{2}.s^2$. Then for $N = 7$, this failure scenario is 21 times less probable than the failure of two arbitrary nodes.

The detailed pseudo code of PaxosInside is presented in Appendix B. Furthermore, Appendix C provides the correctness proofs of PaxosInside.

5.4 PaxosInside: A Multicore Implementation

We present in this section our implementation of PaxosInside among multiple cores. This is based on a message passing framework, which we implemented on top of an inter-process shared memory communication. Our framework is implemented efficiently at the user level using standard C++ libraries, and hence is portable to all the operating systems, including Barrelfish. We expect to have some standard inter-core channels in upcoming computer architectures. For the purpose of performance evaluation in this chapter, similar to the approach taken by the previous work [BBD+09], we make use of shared memory for message passing. Changes made by a process into the shared memory address are first applied to the cache of the core that is running the process. Thanks to the cache coherence mechanism implemented in hardware, the changes in the cache of the source core will be propagated into *only* the cache of the destination core.

Notice that although our implementation transmits the unicast messages via a cache coherence mechanism, it is still faithful to the distributed vision of a manycore system, as there are separate channels per each pair of cores. In a centralized implementation on top of a cache coherence mechanism, a message would be written into the memory and read by all the cores in the system, resembling a broadcast message. It is demonstrated in the related work [BBD+09] that this approach is not scalable with number of cores, since it induces a burst of messages, whereas in the distributed implementation, the software makes use

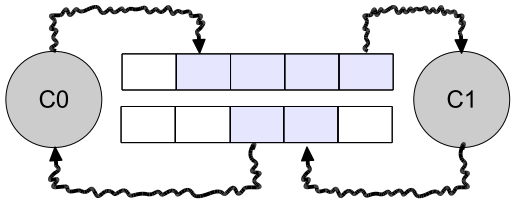

Figure 5.6: Two separate queues are used between each pair of cores.

of its knowledge about the application internal to efficiently decide to *where* and *when* each message must be sent.

In the following, we describe our messaging system and its integration into a user-level thread library for the efficient delivery of messages.

5.4.1 Message Queuing

As mentioned above, we make use of the cache coherence mechanism by writing/reading to a shared memory address, created by *shm_open* system call. To implement asynchronous message passing, we use more than one slot for sending messages. The size of each slot is 128 bytes, which is twice the cache line size. Matching the cache line size, allows for the least number of cache misses for transferring the message.

The multiple slots are wrapped into a queue. As illustrated in Figure 5.6, there are two queues between each two processes p_i and p_j: one for writing by p_i and reading by p_j and the other for reading by p_i and writing by p_j. Because of separate queues, there is no need for operating system locks to access the queues, which makes the design simple as well as efficient. Each queue has a head and tail pointer. The head pointer is moved by the reader and the tail by the writer. The reader process verifies the equality of head and tail pointers to check for new messages.

5.4.2 Message Delivery

As explained above, a process that communicates with n other processes must check for new messages from n separate read queues. After reading a message, the corresponding thread must be notified to process it. To implement this efficiently, we make use of *libtask* [lib], a user-level thread library. By doing so, we reduce the cost of delivering the message to that of a lightweight user-level context switch.

The architecture of our implementation is depicted in Figure 5.7. Upon reading a request from each queue, the requested thread blocks and its reading

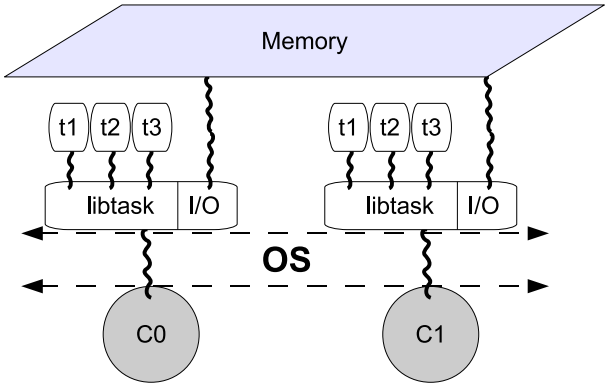

Figure 5.7: The architecture of the implemented framework for message passing in manycore systems.

destination will be added to the scheduler waiting list. The scheduler checks for all waiting reads and, upon receiving a message, loads the context of the corresponding reading thread. In other words, the developer will take advantage of the simple blocking read interface, while the back-end benefits from the asynchronous message passing implementation to gain high performance.

5.4.3 Agreement

We have implemented PaxosInside, Multi-Paxos, and 2PC in our manycore framework. As we explained, PaxosInside also relies on a PaxosUtility module, which can be any implementation of a Paxos-like system. We use Multi-Paxos as PaxosUtility. In this case, PaxosInside and Multi-Paxos implementations are collapsed, i.e., each node implements all three roles of Paxos: proposer, acceptor, and learner.

Following the messaging standard in *libtask*, a replica waits for the clients to connect (by *netlisten* function). Afterwards, the replica creates the send and the receive queues for future communications with the node and also creates a thread for reading the messages from the open connection. The thread will block by calling *fdread* on the connection and process the received message after scheduler wakes it. Note that while a user-level thread is blocked, the replica could still progress by processing other message in other threads.

The clients also call *netdial* on first call to a replica and after the queues are created, use the *fdwrite* function for sending the message. Besides, the client creates a thread for receiving messages from the opened connection, just as replica does. Since we have implemented standard interfaces provided by the library, the implemented protocols in our framework can be easily ported to a

network system with no change. (The library already supports TCP and UDP implementation of the messaging interfaces.)

5.4.4 Description of C++ Code

As one can notice from the PaxosInside pseudo code presented in Figure B.1 of Appendix B, the handlers implementing the acceptor role are simpler, compared to Multi-Paxos. This is because there is only one acceptor in PaxosInside, and thus the complexity due to dealing with a quorum of nodes is eliminated. The handlers of the proposer role, however, implement more logic for the safe recovery from failures. Overall, the C++ implementation of PaxosInside in our framework is 490 LoC as opposed to 581 LoC for implementation of Multi-Paxos, which is also used as PaxosUtility module. Note that the client code and message passing library are common in both of the implementations, and thus are not included in the reported LoC.

5.4.5 Persistent Storage in Multi-Paxos

Multi-Paxos requires storing the accepted value persistently, before responding to an accept message. The persistent stored data will be used by the restarted process, in the case of a crash. Typically, a hard disk is used for persistent storage of data. Inside a computing unit, however, we can think of cheaper and faster storages such as shared memory. The data in the shared memory is retrievable by the restarted process. The challenge here is that to make sure that the requested data is stored into the main memory before the learn message is sent by the acceptor. Otherwise, if the data was located in the cache, a sudden crash failure of the core will lose the data. We are not aware of a standard interface for explicitly asking this from the processor. Also, the probable workaround solutions for flushing the whole cache will not be efficient.

However, in manycore systems a core is considered faulty if it is slow and does not respond in time. The goal, therefore, is the continuous progress even with some slow cores. In this fault model, a simple write into the main memory is enough for a correct implementation of Multi-Paxos, and we do not have to pay for the expensive operation of writing into the hard disk.

5.5 Evaluation

In this section, we report on the evaluation of PaxosInside, Multi-Paxos, and 2PC.

We basically explore the following aspects: (1) The improved commit latency

and throughput by using PaxosInside; (2) The scalability of PaxosInside with the number of cores; and (3) The performance of PaxosInside and 2PC when a core becomes slow.

5.5.1 Experimental Setup

Our experiments make use of a machine with four 2.4 GHz Dual-Core AMD Opteron(tm) processors (8 cores in total) and 8 GB of RAM. The L1 cache size is 64 KB and L2 cache size is 1 MB. These machines run GNU/Linux 2.6.16.37. It is shown by the many years research in state machine replication that to make the consensus scalable, it must only be achieved between a few servers and the other nodes, i.e., clients, make use of that as a service. We have applied the same lesson by using three replicas in all the protocols, which are assigned to separate cores of 0 to 2, via *taskset* command. The clients are assigned to cores 3 to 7. The clients start sending requests to the replicas, after receiving a message from the load manager which is run on Core 7. There is no payload added to the requests and responses. In all the experiments, a client sends a request to Core 0, waits for the commit ACK, and then sends another request, till it finishes 100 requests. We run each experiment for three times and report the average values.

5.5.2 Workload

In the Paxos family of protocols, the messages are issued as a result of each client command, which is the type of traffic targeted by PaxosInside. In general, the read requests do also cause issuing Paxos protocol messages. This is because the read requests often require the last updated data, which is not necessarily updated in every learner, including the leader node. Thus, the read traffic can be treated as normal client command traffic.

There are particular usecases that can change the load on servers. For example, if the read requests do not necessarily ask for the last updated data, then they can be handled directly by each learner. Hence, the read traffic will be balanced on all nodes. In this case, if the proportion of the read traffics is much more than client command traffic, then PaxosInside's impact on reducing the overall load will be less profound. For the sake of generality, we do not assume the above particular cases for handling the read traffic in the experimental results, and all the requests in our experiments are, hence, write requests.

5.5.3 Micro-benchmarks

In this experiment, only one client is used, which is assigned to Core 3. Figures 5.8 and 5.9 depicts the average throughput and the average commit latency

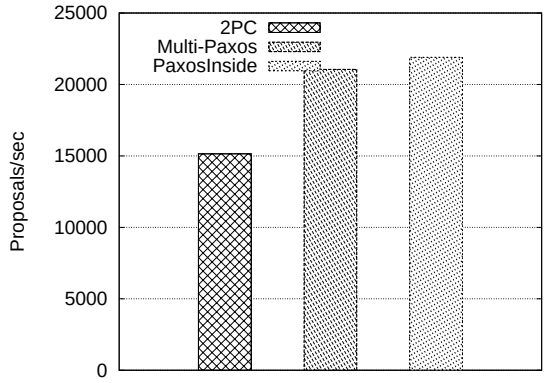

Figure 5.8: Throughput achieved by one client.

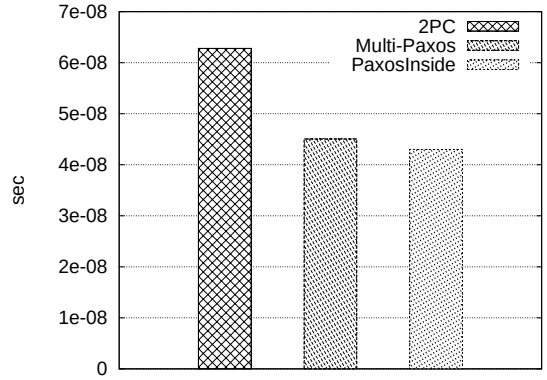

Figure 5.9: Commit latency observed by using one client.

experienced by the client, respectively. The commit latency is the delay between the time the request is sent by the client and the time that the reply is received by it. The throughput is the number of received replies per unit of time.

PaxosInside has the lowest latency whereas 2PC has the highest. The latency of Multi-Paxos is only slightly worse than PaxosInside because of the more message copy operations induced by sending more messages. 2PC, in particular, has a higher latency since it requires two phases to commit, versus one phase in failure-free scenario of PaxosInside and Multi-Paxos. The same pattern applies to the throughput, since the higher the latency of each commit, the fewer requests will be sent by the client per unit of time. This also shows that one client does not saturate the system bandwidth, and the only major bottleneck in throughput of this micro-benchmark, thus, is the commit latency.

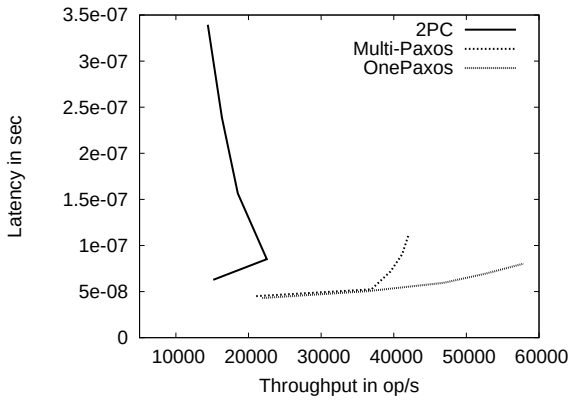

Figure 5.10: The latency vs. throughput by increasing the number of clients.

5.5.4 Scalability

Here we evaluate the scalability of the protocols by increasing the number of clients from one to five (we have eight cores in total.).[5] Figure 5.10 depicts the average commit latency vs. the total achieved throughput by the clients. PaxosInside is the most scalable protocol, as the throughput improves by a factor of three, by using four more clients. Multi-Paxos scales up to only two clients. Afterwards, by adding more clients, the throughput improves slightly, while the latency increases with a high steep. This makes its throughput with five clients stops at 42,031 op/s, 27% less than throughput of PaxosInside, which is still not saturated. 2PC is the least scalable protocol among them; the throughput increases by 50% after adding the first client, however, it even drops by adding more clients to slightly less than the original value. The reason is the numerous messages transmitted by 2PC. These results show that PaxosInside achieves the best performance and scalability among all considered alternatives.

5.5.5 Throughput in Failure Scenarios

We report on measuring the changes in system throughput when the leader core becomes slow. In these experiments, all five clients are sending requests to the replicas. We slow down Core 0 by running eight CPU-intensive processes on it; each process is a bash script that continuously multiplies a number by itself. Figure 5.11 plots the throughput of PaxosInside when the leader becomes slow, as well as the normal non-faulty case. After the clients detect the slow leader, they send their requests to other nodes. The non-leader node after receiving the

[5] We avoided using the cores allocated to the replicas for the client processes. That would make the analysis of the results more complicated both because of the added load on the replica cores and the lower latency between the clients and the replicas.

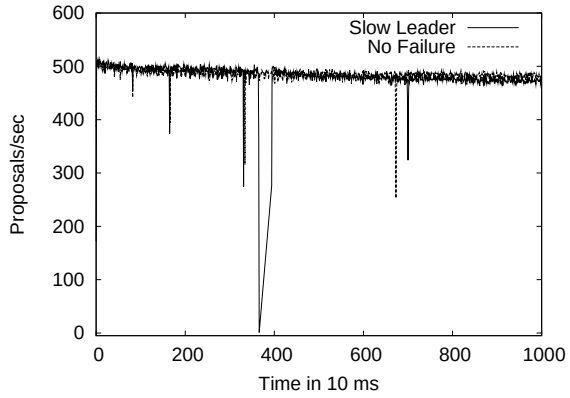

Figure 5.11: The changes in throughput achieved by PaxosInside when the leader is slow.

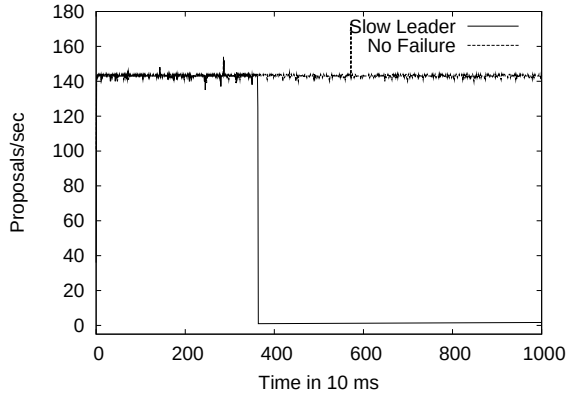

Figure 5.12: The changes in throughput achieved by 2PC when the leader is slow.

clients request, try to become the leader in PaxosUtility. After that, it sends the proposals to the active acceptor. During the leader change process, the throughput drops to zero.

Figure 5.12 plots the throughput of 2PC for the same experiments. Since 2PC is a blocking protocol, a few requests can commit after Core 0 becomes slow and the throughput drops to zero. It is worth noting that the 2PC throughput would suffer from slow Core 2 in the same way, but PaxosInside would continue progressing since Core 2 is neither the leader nor the acceptor.

The experiment results show that by using PaxosInside, not only the performance of consensus between multiple cores improves, but also it has the advantage of continuous progress even with some slow, non-responding cores.

5.6 Related Work

Barrelfish [BBD+09], an implementation of a multikernel model, pioneered the idea that a manycore can be viewed as a distributed system. Key information of the kernel is replicated on several cores and a 2PC (a blocking agreement) algorithm ensures the consistency of the replicas. Unlike PaxosInside, 2PC does not ensure progress with slow, non-responding cores.

Barrelfish exploits the cache hierarchy inside the processors to efficiently broadcast messages via multicast trees. A slow node in the multicast tree can delay the propagation of the message to the rest of the nodes. This approach contrasts with that of PaxosInside, which is precisely designed to address the problem of slow cores, and hence does not use any multicast tree to broadcast messages to the replicas. Otherwise, a faulty, slow core would make all its child nodes under the multicast tree to be unresponsive as well. This would reduce the probability of having a majority of responsive cores, which is the essential assumption for progress of consensus protocols.

A multicast tree in Barrelfish is created based on the measured latency of the core-core communications. The actual delay between two cores depends on the current load of the cores as well as on the traffic inside the processing unit. It is actually not clear that a particular created multicast tree remains the most efficient when the load on the cores changes, which occurs quite often in a fairly loaded processor. In other words, the performance of a multicast tree highly depends on the temporary workload. Thus, not to lose the generality, we avoid multicast trees in benchmarking the performance of the implemented algorithms.

Mencius [MJM08] was derived from Multi-Paxos to distribute the load of client commands among multiple leaders [LHA02]. Assuming a balanced load of client commands received by the leaders, it partitions the space of Paxos instance numbers among the leaders: each leader proposes the received client commands only for its range of instance numbers. By doing so, the leaders can, in total, process more aggregate commands from clients. Note that each leader still has to communicate with all the acceptors to make a proposal. If the load is not balanced on the leaders, the loaded leader could forward its traffic to the other under-loaded leaders, which causes higher delays. The under-loaded leaders also have to skip their share of the instance space, which would not help the load balancing objective. In contrast, PaxosInside targets the load on each leader individually, and is not limited by assuming a balanced load on the leaders.

By reducing the number of messages exchanged between servers (non-client nodes), each leader in PaxosInside can process more client commands. The main insight of PaxosInside can be applied to any protocol of the Paxos family.

Mencius could also benefit from the main insight of PaxosInside and increase the system throughput further. This would enable a Mencius leader to be assigned to a single separate acceptor, and indeed increase the overall throughput.

Some protocols of the Paxos family target the commit latency of client commands [Lam06, DMS, Lam05]. In Basic-Paxos, each client command takes four message delays between the servers. Multi-Paxos, which has been successfully integrated into a number of practical deployed [CGR07,LLPZ07,Bur06] systems, behaves similarly to Basic-Paxos for the first command, but requires only two message delays between servers for the next commands. This does not include the RTT delay between the client and the leader. Fast-Paxos [Lam06], using more replicas ($3f + 1$), saves the delay between the leader and the acceptors by allowing the client to optimistically send the accept_request messages directly to the acceptors. Collisions between commands from different clients can be resolved by spending more steps. The average latency can be lower if the rate of collisions is low. If collisions are frequent, classic Paxos actually outperforms Fast-Paxos [Lam06].

In scenarios where the throughput of the system is a bottleneck, the number of client commands is very high, and the probability of collisions increases accordingly. PaxosInside is designed for high-throughput systems and reducing the number of consensus phases is not targeted by the algorithm. Fast-Paxos cannot outperform the throughput of Multi-Paxos, as the number of sent/received messages to/from each acceptor does not change; although the leader-to-acceptor messages of Multi-Paxos are eliminated in Fast-Paxos, the messages must be sent to more acceptors, $3f + 1$. For $f = 1$, the message/node is equal to six per command, which is the same number as Multi-Paxos. So called BFT protocols [KAD$^+$07, CL02] tolerate not only crashes but also Byzantine faults: these include arbitrary faults and malicious behavior. BFT protocols, because of aiming stronger guarantees, are more expensive than the widely-deployed consensus algorithms [CGR07, LLPZ07, Bur06] (to which PaxosInside belongs).

Chun *et al.* [CMS08] suggests to put the replicas of a BFT protocol inside a multicore machine, saving some computing units. The remote clients send requests to the replicas inside the multicore machine via the network. PaxosInside, in contrary, is based on a solution to scalability of applications inside the multicore. The clients of PaxosInside are, hence, inside the multicore machine and agreement between the distributed state among them is a must, and not an option. PaxosInside solves the agreement problem in very efficient way while tolerating the slow cores.

Since Paxos requires only a majority ($f + 1$) of the replicas to progress, in failure-free scenarios, f of the nodes can be excluded from an execution. This observation is leveraged by Cheap-Paxos [ML04] to improve the throughput. Yet, this optimization comes with liveness penalties. For example, with three Replicas r_1, r_2, and r_3, if r_1 fails and afterward r_2 fails, then the system cannot

progress until r_2 recovers. In other words, the recovery of r_1 does not help since r_2 has the crucial last state of the system. In comparison, PaxosInside can progress as long as any two of the three replicas are responding. For example, in the above scenario, PaxosInside progress as soon as either r_1 or r_2 starts responding. In this sense, PaxosInside does not jeopardize the liveness of Paxos and yet offers higher performance.

5.7 Summary

This chapter initiates the study of message passing consensus algorithms in a manycore system. In short, we show that such a non-blocking agreement protocol can be efficiently implemented among multiple cores to ensure the consistency of replicated data.

We proposed PaxosInside, a Paxos-like consensus protocol that attains good performance by using only a single acceptor, which is replaced only in case of non-responsiveness. PaxosInside was specifically designed with a manycore system in mind: roughly, it transmits fewer messages than alternative consensus protocols, which reduces the load both on the core interconnect and the leader core's cache.

We showed how to efficiently implement PaxosInside in a manycore system by using two separate queues between each pair of cores and cheaply delivering the received messages via libtask, a user-level thread library. Our experimental results conveyed the very fact that, on a manycore system, PaxosInside outperforms Multi-Paxos, the most efficient message passing consensus to date, and even a classical (blocking) 2PC algorithm. PaxosInside might block if both the leader and the acceptor are not responsive at the same time. In this scenario, which is arguably very rare, PaxosInside progresses after at least one of them starts responding. For the setup that uses three cores, PaxosInside and the Paxos family of algorithms can tolerate the same number of faults, i.e., one.

I am sorry to write such a long letter, but I did not have time to write a short one.

Blaise Pascal

6 Conclusion

In this thesis, 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.

The proposed approach, Dervish, makes use of a model checker running in parallel with each distributed node to explore the possible future states. To make our approach feasible, the performance of the model checker should be improved. We introduce the notion of event chain: a sequence of events triggered by a non-network event and continued by some network messages. For the algorithms that produce relatively short event chains, e.g., DHT, we designed and implemented consequence prediction, an algorithm for selectively exploring future event chains of the system.

For complex algorithms with long event chains, we introduced LMC, a novel approach in model checking distributed algorithms. LMC separates the network state from the global state and focuses on the remaining system states, which is the required part for invariant checking. Besides, instead of keeping track of the system states, LMC keeps the traversed local states by each node separately; the system states are created temporarily only to check the invariants against them. We then used LMC to find bugs in some complex consensus algorithms, including PaxosInside, the first consensus algorithm proposed and implemented for manycore environments.

6.1 Summary of Results

The summary of the results of this thesis centers around four axes:

Deep Online Model Checking of Distributed Systems We introduced the approach of running a model checker in parallel with each distributed node. The model checker is restarted before slows down by the exponential explosion problem. At each run, the model checker instead of the initial state, starts from the current live state of the system. This approach enables the model checker to check for the more relevant states: states that are likely to be reached from the current live system state. This technique enabled us to find some new bugs in mature implementations of RandTree, Chord, and Bullet′, presented in Chapter 3.

Execution Steering We introduced the notion of execution steering in distributed systems. After an inconsistency is predicted by the model checker, Dervish steers the execution away from it before it actually can occur. This is possible because the model checker can simulate packet transmission in time shorter than propagation latency, and because it can simulate timer events in time shorter than the actual time delays. The presented results in Chapter 3, demonstrates the possibility of this approach by steering the execution away from the predicted bugs in Chord and Paxos.

Model Checking Distributed Algorithms Chapter 4 introduced LMC, a novel approach in model checking distributed algorithms. Because of removing the network element from the memorized states, much less system states are created at the start, which results in 300∼8,000 times speedup in LMC. Using Dervish augmented with LMC, we rediscovered a previously reported bug in Paxos implementation as well as a bug in a new consensus protocol, PaxosInside.

PaxosInside We explored, for the first time, the feasibility of implementing a (non-blocking) consensus algorithm in a manycore system. Chapter 5 presented PaxosInside, a new *consensus* algorithm that takes up the challenges of manycore environments, such as limited bandwidth of interconnect network as well as the consensus *leader*. A unique characteristic of PaxosInside is the use of a single *acceptor* role in steady state, which in our context, significantly reduces the number of exchanged messages between replicas. Furthermore, we presented an efficient implementation of PaxosInside on a manycore system.

6.2 Future Work

This thesis opens new opportunities for future works, which are categorized in the following.

6.2.1 On Symbolic Execution

Symbolic execution (SE) sounds like a promising alternative for model checking in Dervish. It has the advantage of considering unexpected receipt of messages from arbitrary nodes, including new joining nodes. Nevertheless, the current performance of SE tools is not good enough for this purpose. Perhaps SE techniques that work on standard higher level data structures, e.g., map, list, and set, will have a higher chance on combating the exponential state space explosion problem in distributed algorithms.

6.2.2 On Incremental Model Checking

Dervish restarts the model checker to be able to incorporate the current live state into the model checking process. This approach has the drawback that the restarted model checker will redundantly explores some of the states that it has already visited. It is more favorable then to develop techniques to incorporate the current live state into the running model checker without, however, discarding the previous visited states.

6.2.3 On Collaborative State Exploration

Each node in Dervish runs a separate copy of the model checker, which explores the global system states independently. Alternatively, the state space could be partitioned among the collaborating model checkers to make the model checking more efficient by avoiding redundant explorations.

6.2.4 On Collaborative Filter Installation

Currently, Dervish applies a conservative approach in filter installation: all nodes install filters for all the predicted bugs. Although effective for avoiding inconsistent states, some early filtered events could negatively affect the system performance since the predicted inconsistency would not necessarily manifest at all runs. Alternatively, the model checkers running on distributed nodes could collaboratively decide on the optimal point of intervention in the predicted erroneous paths.

6.2.5 On Model Checking Heuristics

In Dervish, the model checker is restarted after some thresholds. Therefore, no matter the search algorithm is complete or not, the exploration performed by the model checker in its lifetime is incomplete. The search heuristics that sacrifice the completeness of the exploration in favor of some more interesting states are then very welcome. Consequence prediction presented in Chapter 3 was just one of such heuristics, and there are lots of space for developing new ones. It is especially more interesting if the heuristics come with some proved bounds on the covered states.

6.2.6 On Invariants

The model checking as well as other similar techniques reduces the problem of software testing to that of specifying the right, effective invariants of the system. This task turns out to be non-trivial in practice since reasoning about the global properties on distributed nodes is very difficult. Either eliminating this step in the ideal case or at least helping in detecting the right invariants could save lots of testing time and help find more bugs.

6.2.7 On Model Checking Distributed Algorithms

The performance of LMC could hugely be improved by skipping some system states if we know that they are not relevant to the particular user-specified invariant. In Chapter 4, we applied this scheme for the Paxos main safety property. For future works, one can think of methods to automatically prune the system states according to a given invariant.

6.2.8 On PaxosInside

PaxosInside was a first step towards exploring the implementability and benefits of consensus algorithms in manycore systems: we believe that the road ahead is full of interesting discoveries. The consensus algorithm could be further optimized, taking advantage of the assumption that the processes will not crash. Alternatively, one can include crash and Byzantine faults in the core fault model and devise protocols that can efficiently tolerate them. Developing application on top of the presented message passing framework and benchmarking their scalability compared with the centralized version are another interesting future works.

◇◇◇

A

Example Run of Consequence Prediction on a Small Service

Figure A.4 shows a small service where each node has a local counter and two states. In the initial state, the node issues a Query message to another node. Upon receiving a Response message, the node stops its operation. We use this example to illustrate the differences between consequence prediction and dynamic partial order reduction.

Figures A.1, A.2, and A.3 show, respectively, the entire space of reachable states, the transitions explored by a dynamic partial order reduction algorithm, and transitions explored by Consequence Prediction. There are 16 reachable states in the system, 20 paths, and 24 edges. A dynamic partial order reduction algorithm explores 11 of these edges, and consequence search explores 14 edges. Note that dynamic partial order reduction explores first one complete path, then inserts branches to consider certain interleavings. In contrast, consequence search explores event chains initiated by internal actions, as well as sequences of these chains where the last action of one sequence changes the state in which next local action occurs.

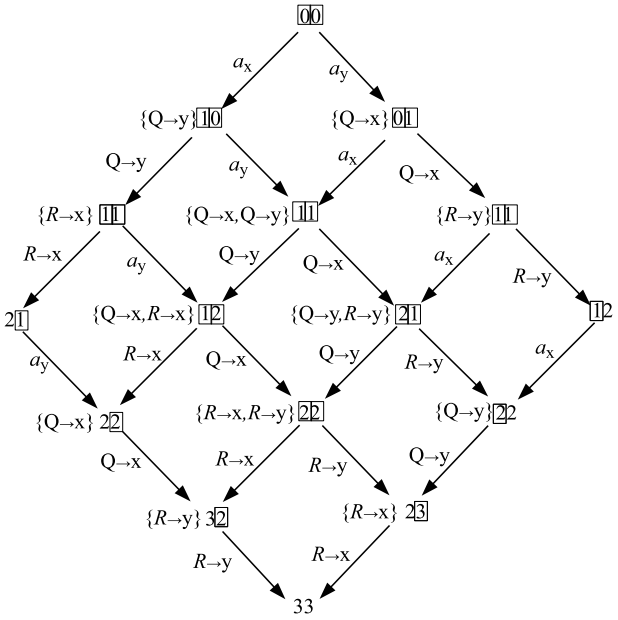

Figure A.1: Reachable states of the system in Figure A.4

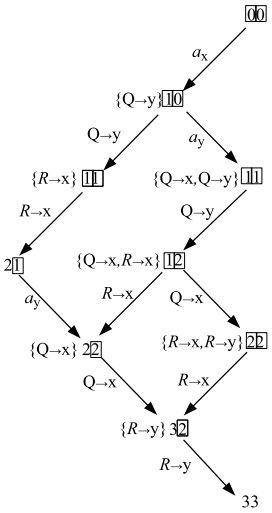

Figure A.2: A Dynamic Partial Order Reduction

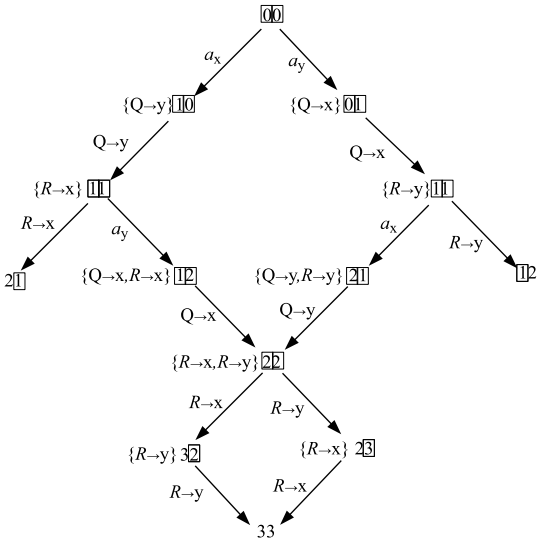

Figure A.3: Consequence Prediction

```
1 downcall (state == init) maceInit(MaceKey ip) {
2   localCounter=0;
3   state=started;
4   peerId = ip;
5 }
6
7 downcall (state == started) doQuery() {
8   localCounter++;
9   state=querySent;
10  downcall_route(peerId, Query());
11 }
12
13 upcall (state != init) deliver(const MaceKey& src, const MaceKey& dest,
14                                 const Query& msg) {
15  localCounter++;
16  downcall_route(src, Response());
17 }
18
19 upcall (state != init) deliver(const MaceKey& src, const MaceKey& dest,
20                                 const Response& msg) {
21  localCounter++;
22  state=finished;
23  downcall_route(src, Response());
24 }
```

Figure A.4: Example Service in Mace language

B

PaxosInside: Pseudo Code

The pseudo code for our PaxosInside algorithm, explained in Section 3, is presented in Figure B.1. If the proposer recognizes itself as the leader of the active acceptor, Variable *IamLeader* is set. The leader does not need to send a prepare_request message to the acceptor and starts directly with the accept_request message using the last promised proposal number. Variable A_a refers to the active acceptor Id; *ap* is the map structure keeping the accepted proposals; *IamFresh* indicates that the acceptor has adopted no leader yet. The highest proposal number is stored in Variable *hpn*. Initially the highest proposal number is equal to -∞. Procedure *init*, presented in Figure B.2, initializes the mentioned variables. As for the rest of the variables, *pn* is the proposal number, *in* is the Paxos instance number, v is the value (proposed or accepted), P is the list of the proposers, L is the list of learners, *me* is the node Id, and *YouMustBeFresh* indicates that the proposer expects to be the first proposer that contacts the acceptor.

Upon a failure of the active acceptor, the leader first checks whether the others still believe him as the leader or not. If not, one other proposer has taken its position (probably because of a false leader failure alarm). In this case, it relinquishes the leadership position and return. Otherwise, it calls *selectAcceptor* function to select a new acceptor that located on a separate node than the leader node. The leader then announces the change of the active acceptor, *AcceptorChange*, through PaxosUtility. It also attaches the uncommitted proposed values to the *AcceptorChange* entry. The failure of this step indicates that another item is chosen for the current instance of PaxosUtility. In this case, the leader returns from this procedure to try again later. In case of success, how-

```
1  Upon AcceptorFailure
2    if (!IamLeader) return;
3    (P_i,instance) = PaxosUtility.lastLeader();
4    if (P_i ≠ me) //somebody thought I am dead
5      A_a = null; IamLeader = false;
6      return;
7    A'_a = selectAcceptor();
8    proposals = uncommitedProposals();
9    success = PaxosUtility.propose(instance,
10                 AcceptorChange(A'_a,proposals));
11   if (!success) return;
12   A_a = A'_a;
13   IamLeader = false;
14
15 Upon LeaderFailure
16   propose();
17
18 proc propose()
19   if (IamLeader)
20     in = next_uncommited_instance_number();
21     v = getAny(in);
22     sendto A_a accept_request(in, pn, v);
23   else
24     YouMustBeFresh = true;
25     pn = new_pn();
26     if (A_a == null)
27       (A_a,instance,proposals) =
28         PaxosUtility.lastActiveAcceptor();
29       if (A_a == me) return;
30       success = PaxosUtility.propose(instance,
31                                  LeaderChange(me, A_a));
32       if (!success)
33         A_a = null; return;
34       registerProposals(proposals);
35       YouMustBeFresh = true;
36     sendto A_a prepare_request(in, pn, YouMustBeFresh);
37
38 Upon Receive prepare_response(A_i, pn, ap)
39   if (IamLeader || A_i ≠ A_a) return;
40   IamLeader = true;
41   registerProposals(ap);
42   in = next_uncommited_instance_number();
43   v = getAny(in);
44   sendto A_a accept_request(pn, v);
45
46 Upon Receive prepare_request(P_i, pn, YouMustBeFresh)
47   if (pn > hpn)
48     if (IamFresh != YouMustBeFresh)
49       return;
50     IamFresh = false;
51     hpn = pn;
52     sendto P_i prepare_response(pn, ap);
53   else sendto P_i abandon(hpn);
54
55 Upon Receive accept_request(P_i, in, pn, v)
56   if (pn ≠ hpn)
57     sendto P_i abandon();
58   else if (ap[in] ≠ null)
59     multicast L learn(in, ap[in]);
60   else
61     ap[in] = (pn, v);
62     multicast L Learn(pn, accepted);
```


Figure B.1: PaxosInside Algorithm

ever, the leader resets Variable *IamLeader* because it has to start from the first phase of Paxos with the new active acceptor.

Upon failure of the current leader, a proposer tries to take its position by calling Procedure *propose*. The procedure then obtains the active acceptor Id and sends a prepare_request message to it.

Procedure *propose* proposes a value for the next uncommitted instance number. If the node is already the leader, it directly sends an accept_request message to the active acceptor. Otherwise, it sends a prepare_request message to the active acceptor, in accordance with the first phase of the Paxos algorithm. If the active acceptor Id is unknown to the proposer, it must be obtained via PaxosUtility. The *lastActiveAcceptor* method checks the sequence of committed entries looking for the last *AcceptorChange* entry; this entry contains the active acceptor Id. Next, the proposer adds a *LeaderChange* entry via PaxosUtility. The failure of this step indicates that another item is chosen for the current instance of PaxosUtility. In this case, the procedure resets the value of A_a and returns. We assume that the implementation retries the failed attempt via timers or some other mechanisms. In the case of success, before sending the prepare_request message, it first registers the proposed values which have been recorded with the last AcceptorChange entry. If the acceptor is supposed to be a fresh backup acceptor, it also sets Variable *YouMustBeFresh* which is sent by the message.

Upon receipt of the prepare_request message from proposer P_i, the acceptor verifies the highest proposal number *hpn* to be less than the requested proposal number, *pn*. Otherwise, it sends an abandon message back to proposer P_i. If Variable *IamFresh* is set but Variable *YouMustBeFresh* is not, it indicates that the proposer expected the acceptor to be already adopted by the last leader. However, due to the acceptor reset, the acceptor has lost its data, including *hpn* and *ap*. This check avoids the cases where the active acceptor silently reboots before the leader switch. In this case, the last leader should switch the rebooted acceptor.

Upon receipt of the prepare_response message from the active acceptor, the proposer claims the leadership position by setting Variable *IamLeader*. The *getAny* method, presented in Figure B.2, picks a value to be accepted for the instance *in*. The picked value can be any given value, unless there is already a proposed but uncommitted value for the instance *in*. This case can occur in change of the active acceptor, when some proposed values are not committed yet by the previous active acceptor. If any proposal matches the instance number *in*, to avoid inconsistency, the proposer picks the same previously proposed value. It then sends an accept_request message to the active acceptor.

Upon receipt of the accept_request message from the leader, the acceptor first checks for the proposal number. Also, it checks that there is no proposal accepted corresponding to the instance number, i.e., $ap[in]$. Otherwise, it broad-

```
1  proc init()
2    IamLeader = false; A_a = null;
3    ap = emptyMap(); hpn = −∞
4    IamFresh = true;
5
6  proc getAny(in, ap)
7    v = proposed[in];
8    if (v ≠ null) return v;
9    v = nextClientRequest();
10   proposed[in] = v;
11   return v;
12
13 proc registerProposals(proposals)
14   foreach p in ap
15     proposed[p.in] = p.v;
```

Figure B.2: The implementation of Procedures init, getAny, and registerProposals, in PaxosInside Algorithm

casts the learn message of the accepted proposal again to cover the cases that the lost learn message has motivated the proposer to retry. It then stores the proposal in the accepted proposal map, $ap[in]$. Afterwards, the accepted proposal is broadcasted to all the learners accordingly.

C

PaxosInside: Proof of Correctness

Here, we prove the correctness of the algorithm presented in Appendix B. We first prove some properties for the entries in PaxosUtility, which we then use to prove that no two different values would be accepted for the same instance number. The proof for the simple case where there is no change in the active acceptor nor the leader node, is trivial and similar to the proofs of Paxos. Here, we focus on the complex cases where the algorithm switches the leader and the active acceptor.

PaxosUtility contains entries for changing the active acceptor, i.e. AcceptorChange, and entries for changing the leader, i.e. LeaderChange. We define the *Global leader* and *Global acceptor* as follows:

definition: In the sequence of PaxosUtility entries, the node which has inserted the last LeaderChange entry is the *Global leader*. Similarly, the active acceptor announced by the last AcceptorChange message, represents the *Global acceptor*. We use GL_i to represent the ith Global leader and GA_i to represent the ith Global acceptor.

Lemma 1: An AcceptorChange entry is inserted only by the Global leader.

Lemma 1 is guaranteed by lines 3..13 of Figure B.1. In Line 4 the leader verifies that it is still the Global leader. It also keeps the index of the last empty instance number, *instance*. Later at Line 10, it proposes the AcceptorChange message for that instance number. The failure of this phase implies that another node has inserted something in the meanwhile. In this case, the handler returns to retry the procedure later from scratch. Therefore, the AcceptorChange message is inserted only by the Global leader.

According to Lemma 1, the Global acceptor represents the active acceptor with which the Global leader is working.

Now we prove by induction that the same value will always be accepted for a particular instance number. The first step is to show that a Global leader does not propose two different values for the same instance number when it switches between the acceptors. Hereafter, we use Pair (v,i) to represent Value v and Instance Number i of a given accept_request messages.

Lemma 2a: Suppose that GL_l has issued two accept_request messages, (v_a,i_a) and (v_{a+1},i_{a+1}), to two consecutive Global acceptors GA_a and GA_{a+1}, respectively. If $i_a = i_{a+1}$, then $v_a = v_{a+1}$.

Lemma 2a is directly followed by the implementation of the Procedure *getAny* in Figure B.2. There, the leader first checks the history of the proposed values. If any value has already been proposed for the requested instance number, then the procedure returns the same value. Hence, as long as the Global leader is not changed, the proposed value for a particular instance number will be always the same.

The next step is to show that an acceptor accepts the same proposals from two consecutive Global leaders.

Lemma 2b: Suppose that the active acceptor GA_a accepts two accept_request messages, (v_l,i_l) and (v_{l+1},i_{l+1}), from two consecutive Global leaders, GL_l and GL_{l+1}, respectively. If $i_l = i_{l+1}$, then $v_l = v_{l+1}$.

Node GL_{l+1} becomes the Global leader only after successfully inserting a LeaderChange entry via PaxosUtility. In the algorithm presented in Figure B.1, this happens only at Line 30 inside the Procedure *propose*. It also implies that the value of Variable $Iamleader$ is false (Line 25). GL_{l+1} will not start proposing values unless the value of Variable $Iamleader$ changes to true (Line 21). Line 41 is the only location where the value of this variable is changed to true upon receipt of a prepare_response message. It indicates that the active acceptor has received the prepare_request message, approved the proposal number, and responded by the prepare_response message which is also piggybacked by all the previous accepted proposals, *ap*. The received accepted proposals are registered by the leader (Line 42). The registered values will be later used for all the next proposals in Procedure *getAny*. In other words, the GL_{l+1} will propose the same values that has already been accepted by acceptor GA_a.

Similar to Basic-Paxos, GA_a will reject all the other potential issued accept_request messages by GL_l after sending the prepare_response message to GL_{l+1}. On the other hand, as we showed above, if GA_a has accepted any value from GL_l for a particular instance number, it will not receive any different value from GL_{l+1} for that sequence number. Consequently, GA_a always accept the same values from two consecutive Global leaders.

Having Lemma 2a and Lemma 2b, now we present the correctness proof of the algorithm.

(*) Suppose that two acceptors GA_a and $GA_{a'}$ accept two accept_request messages, (v_a,i_a) and ($v_{a'}$,$i_{a'}$), received from the Global leaders GL_l and $GL_{l'}$, respectively, where $l' \geq l$ and $a' \geq a$. If $i_a = i_{a'}$, then $v_a = v_{a'}$.

The proof is by induction on the size of sequence of entries in PaxosUtility. Assume that property (*) holds when PaxosUtility has k entries. We prove that it still holds when PaxosUtility has $k + 1$ entries.

Recall that the entries in the PaxosUtility utility are either AcceptorChange or LeaderChange. If the $k + 1$th entry is AcceptorChange, based on Lemma 1 it is inserted by the last Global leader. Thus, the GL is the same and the GA changes. This is the case in Lemma 2a for which we proved that no two values will be proposed for the same instance number. If the $k + 1$th entry is LeaderChange, we can assume that GA is the same during this change. This is provided by the Lines 29..30 in Figure B.1, where the new leader takes the same active acceptor as was taken by the last leader. This case is covered by Lemma 2b for which we proved that no two values will be accepted for the same instance number. Consequently, if we assume that no two values are accepted for the same instance number in the first k entries of PaxosUtility utility, this also holds for the first $k + 1$ entries.

Now, to complete the proof, we need to show that the theory holds for $k = 2$. We can make it hold by an initialization process. At the start up, the node with the smallest Id can insert two entries for LeaderChange and AcceptorChange to announce itself as the Global leader and its active acceptor as the Global acceptor. Because, no change in the roles happens in the initial case, neither for the leader nor for the active acceptor, then the theory directly holds for this case.

D

LMC: Soundness Verification Proofs

We argue here that the soundness verification module of Dervish, implemented by Procedure `isSequenceValid` of Figure 4.7, reports a given system state as valid if and only if the corresponding set of sequenced events is valid.

Theorem 1: If Procedure `isSequenceValid` returns `true` then the input system state is valid.

This is trivial since the procedure executes the sequence of events of each local node in order and only if these events are enabled. Therefore, the resultant global state could also occur in a real run, i.e., it is valid.

Theorem 2: If the input set of sequenced events is valid, then Procedure `isSequenceValid` returns `true`.

We use here partial and total orders. Basically, the input sequences of events (one for each node state) define a partial order on the events. Considering all local nodes, there might be more than one way to combine their event sequences, i.e., obtaining a total order between the events of all the nodes. A total order must also respect event causality. For example, an event in node n_1 that receives a message from node n_2 cannot be executed until the message is generated by an event in node n_2.

The partial order between the events is defined by the happens-before relation [Lam78] as follows.

Definition: Let e_i^n be the ith event of node n. Relation happens-before, $\rightarrow$, between two events is defined by the following rules: (i) $e_i^n \rightarrow e_{i'}^n \Leftrightarrow i \leq i'$, (ii) e_i^n generates the network message used by $e_{i'}^{n'} \Rightarrow e_i^n \rightarrow e_{i'}^{n'}$.

Notice that happens-before is antisymmetric, i.e., $e \rightarrow e', e' \rightarrow e \Rightarrow e = e'$. The total order of events executed by Procedure `isSequenceValid` could be obtained by sequencing the events in the time order that they are executed by the procedure. The following lemma expresses Theorem 2 using the notion of total order. The set of all total orders is denoted TO. Each Total Order $TO_i \in TO$ is a sequence of a subset of events that respects the partial orders. The projection of node n on Total Order TO_i is denoted TO_i^n.

Lemma 1: $\exists TO_i \in TO.\ \forall e.\ TO_i.e \notin TO \Rightarrow \neg\exists TO_j.\ \mid TO_j \mid > \mid TO_i \mid$.

Lemma 1 indicates that, if the size of a total order, obtained by any given algorithm, cannot be increased, then no algorithm could obtain a bigger total order. In other words, since Procedure `isSequenceValid` is finished by checking the possibility of increasing the size of the created total order, if the result is not a total order that includes all the input events, then there is no such total order. We prove Lemma 1 by contradiction. Assume there exists such TO_j. Then we have

$$\exists k \geq 1, n_m \in nodes(0 < m \leq k).\ \mid TO_j^{n_m} \mid > \mid TO_i^{n_m} \mid \tag{D.1}$$

Let $e_f^{n_m} = TO_j^{n_m}[\mid TO_i^{n_m} \mid +1]$, where Operator $[i]$ gives the ith element of the sequence. In other words, $e_f^{n_m}$ denotes the first new event of node n_m in Total Order TO_j that is not included in Total Order TO_i. Observe that $e_f^{n_m}$ could not be a local event since it would then have been enabled in TO_i as well.

Let $s(e)$ be the event that has generated the message handled by Network Event e. Let $n(e)$ be the node of Event e. We use sm to denote $s(e_f^{n_m})$. We have $sm \notin TO_i^{n(sm)}$, otherwise sm would have made $e_f^{n_m}$ enabled in TO_i. Therefore we have $e_f^{n(sm)} \rightarrow sm$, and consequently $e_f^{n(sm)} \rightarrow e_f^{n_m}$. This leads us to the following equation:

$$\forall n_m(0 < m \leq k).\ \exists n_{m'}(0 < m' \leq k).\ e_f^{n'_m} \rightarrow e_f^{n_m}, e_f^{n_m} \not\rightarrow e_f^{n'_m}$$

This is obviously a contradiction since it implies a first element while at the same time demands a smaller element than that.

Bibliography

[AVY08] Matthew Arnold, Martin Vechev, and Eran Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 143–162, New York, NY, USA, 2008. ACM.

[BBD+09] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP*. ACM, 2009.

[BDIM04] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[BM05] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 1052–1059, Washington, DC, USA, 2005. IEEE Computer Society.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *TACAS*, 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[Bur06] M Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, volume 11, 2006.

[CCC+05] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, October 2005.

[CCG+03] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *TSE*, 30(6), 2003.

[CCZ+07] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and

Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*, 2007.

[CDK+03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth Content Distribution in Cooperative Environments. In *SOSP*, October 2003.

[CGJ+02] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*, June 2002.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: an Engineering Perspective. In *PODC*, 2007.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[CL02] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[CMS08] B.G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 287–292. USENIX Association, 2008.

[CRSZ02] Yang Chu, S. G. Rao, S. Seshan, and Hui Zhang. A Case for End System Multicast. *Selected Areas in Communications, IEEE Journal on*, 20(8):1456–1471, Oct 2002.

[DAKV09] Darren Dao, Jeannie R. Albrecht, Charles Edwin Killian, and Amin Vahdat. Live Debugging of Distributed Systems. In *Compiler Construction*, 2009.

[DKK09] Pierre-Évariste Dagand, Dejan Kostić, and Viktor Kuncak. Opis: Reliable distributed systems in OCaml. In *ACM SIGPLAN TLDI*, 2009.

[DMS] D. Dobre, M. Majuntke, and N. Suri. CoReFP: Contention-Resistant Fast Paxos for WANs. Technical report, Technical report, TU Darmstadt, Germany, 2006.

[DR03] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *OOPSLA*, 2003.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on information theory*, 29(2), 1983.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[FPK+07] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[FQ03] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Model Checking Software*. Springer, 2003.

[GAM+07] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *NSDI*, 2007.

[God96] P. Godefroid. Partial-order methods for the verification of concurrent systems–An approach to the state-explosion problem, volume 1032 of. *Lecture Notes in Computer Science*, pages 1–143, 1996.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM New York, NY, USA, 1997.

[GR06] R. Guerraoui and L. Rodrigues. *Introduction to reliable distributed programming*. Springer-Verlag New York Inc, 2006.

[GW94] Patrice Godefroid and Pierre Wolper. A Partial Approach to Model Checking. *Inf. Comput.*, 110(2):305–326, 1994.

[GY10] R. Guerraoui and M. Yabandeh. Independent Faults in the Cloud. In *LADIS*, 2010.

[HJMQ03] T.A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, 2003.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, 2002.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[JEK+90] 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.

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[JKBK+08] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus Routing: The Internet as a Distributed System. In *NSDI*, San Francisco, April 2008.

[JM06] Muhammad Umar Janjua and Alan Mycroft. Automatic Correction to Safety Violations. In *Thread Verification (TV06)*, 2006.

[JMK+08] Navendu Jain, Prince Mahajan, Dmitry Kit, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *OSDI*, December 2008.

[KAB+07] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[KAD+07] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review*, 41(6):45–58, 2007.

[KAJV07] Charles E. Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[KBK+05] Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. Maintaining High Bandwidth under Dynamic Network Conditions. In *USENIX ATC*, 2005.

[KRA+03] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *USITS*, 2003.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Com. of the ACM*, 21(7):558–565, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[Lam01a] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[Lam01b] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[Lam05] L. Lamport. Generalized consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[Lam06] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LGW+08] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^{3}S: Debugging Deployed Distributed Systems. In *NSDI*, 2008.

[LHA02] L. Lamport, A. Hydrie, and D. Achlioptas. Multi-leader distributed system, November 21 2002. US Patent App. 10/302,572.

[lib] libtask. http://swtch.com/libtask/.

[LLPZ07] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[LS79] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. 1979.

[Lyn96] N.A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[ME04] Madanlal Musuvathi and Dawson R. Engler. Model Checking Large Network Protocol Implementations. In *NSDI*, 2004.

[Mit02] J. Mitchell. Multiset rewriting and security protocol analysis. In *Rewriting Techniques and Applications*. Springer, 2002.

[MJM08] Yanhua Mao, Flavio Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *OSDI*, 2008.

[ML04] M. Massa and L. Lamport. Cheap paxos. In *DSN*, 2004.

[MPC+02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[MPR06] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *ICTAC*. Springer, 2006.

[MPR10] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-Modular Counterexample-Guided Abstraction Refinement. In *SAS*, 2010.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[MS02] D. Manivannan and Mukesh Singhal. Asynchronous Recovery Without Using Vector Timestamps. *J. Parallel Distrib. Comput.*, 62(12):1695–1728, 2002.

[NCF05] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative Execution in a Distributed File System. In *SOSP*, 2005.

[PHR+09] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating systems transactions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 161–176, New York, NY, USA, 2009. ACM.

[QTZS07] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[RCD+04] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, 2004.

[RD01] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *SOSP*, 2001.

[RGK+05] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A Public DHT Service and Its Uses. In *SIGCOMM*, August 2005.

[RKB+04] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, 2004.

[RKW+06] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[SA06a] K. Sen and G. Agha. Automated systematic testing of open distributed programs. *Fundamental Approaches to Software Engineering*, pages 339–356, 2006.

[SA06b] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *FASE*, pages 339–356, 2006.

[Sch90] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[SKAZ04] Sudarshan M. Srinivasan, Srikanth K, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX ATC*, 2004.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[SMRD06] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. *SIGOPS Oper. Syst. Rev.*, 40(4):389–402, 2006.

[VYW+02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*, December 2002.

[WKK+08] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A. Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI*, 2008.

[WLK+09] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The Theory of Deadlock Avoidance via Discrete Control. In *POPL*, 2009.

[Yab10] Maysam Yabandeh. Model Checking Tools for Software System Implementations. Technical report, 2010.

[YACK09] M. Yabandeh, A. Anand, M. Canini, and D. Kostic. Almost-invariants: From bugs in distributed systems to invariants. Technical Report NSL-REPORT-2009-007, EPFL, 2009.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, April 2009.

[YFG10] Maysam Yabandeh, Leandro Franco, and Rachid Guerraoui. One Acceptor is Enough. Technical Report LPD-REPORT-2010-01, EPFL, January 2010.

[YK09] Maysam Yabandeh and Dejan Kostic. DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems. Technical Report TR-2009-05, EPFL, 2009.

[YKKK09] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[YKKK10] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 28(1):1–49, 2010.

[YSE06] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors. In *OSDI*, 2006.

[YTEM06] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

[YVKK09] Maysam Yabandeh, Nedeljko Vasić, Dejan Kostić, and Viktor Kuncak. Simplifying distributed system development. In *HotOS*, May 2009.

List of Figures, Algorithms, and Tables

Figure 1.1 Execution path coverage by a) classic model checking, b) replay-based or live predicate checking, c) Dervish in deep online testing mode, and d) Dervish in execution steering mode. A triangle represents the state space searched by the model checker; a full line denotes an execution path of the system; a dashed line denotes an avoided execution path that would lead to an inconsistency. 2

Figure 1.2 An inconsistency in a run of RandTree; Safety property: children and siblings are disjoint lists 5

Figure 1.3 An example execution sequence that, thanks to execution steering, avoids the inconsistency from Figure 1.2. 8

Figure 2.1 A simple model of a distributed system 13

Figure 2.2 The classic DFS-based (Depth First Search) algorithm to model checking distributed systems. 14

Figure 2.3 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. 15

Figure 3.1 High-level overview of Dervish. This figure depicts the high-level architecture of Dervish and its main components. 22

Figure 3.2 Consequence Prediction Algorithm 25

Figure 3.3 Full state space, consequence search, and partial order reduction in an example with internal actions of three distinct nodes 27

Table 3.1 Summary of inconsistencies found for each system using Dervish. 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 succinctly. The C++ numbers do not include the line counts for libraries and low-level services that services use from the Mace framework. . . . 38
Figure 3.4 An inconsistency in a run of RandTree. Root (n_9) appears as a child. . . . 40
Figure 3.5 An inconsistency in a run of Chord. Node n_c has its predecessor pointing to itself while its successor list includes other nodes. . . . 42
Figure 3.6 An inconsistency in a run of Chord. For node n_a, its successor and predecessor do not obey in ordering constraint. . . . 44
Figure 3.7 MaceMC performance: the elapsed time for exhaustively searching in RandTree state space. . . . 47
Figure 3.8 Scenario that exposes a previously reported violation of a Paxos safety property (two different values are chosen in the same instance.). . . . 48
Figure 3.9 Scenario that includes *bug2*, where node n_b resets but after reset forgets its previously promised and accepted values. This leads to violation of the main Paxos safety property (two different values are chosen in the same instance.). . . . 49
Figure 3.10 In 200 runs that expose Paxos safety violations due to two injected errors, Dervish successfully avoided the inconsistencies in all but 1 and 4 cases, respectively. . 51
Figure 3.11 In this experiment we run Paxos across an emulated wide area network using ModelNet. The experiment contains 200 runs in which the same two errors as in Figure 3.10 were injected. Dervish successfully avoided the inconsistencies in all but 6 and 3 cases, respectively. . . . 52
Figure 3.12 In 100 runs that expose a Chord safety violation we identified, Dervish successfully avoided the inconsistencies in all cases. . . . 53
Figure 3.13 The memory consumed by consequence prediction (RandTree, depths seven to eight) fits in an L2 CPU cache. . . . 54
Figure 3.14 The average amount of memory consumed by each explored state. . . . 55

Figure 3.15 Dervish slows down Bullet′ by less than 5% for a 20 MB file download. 56

Figure 4.1 State transition in model checking distributed systems. In (a) the classic global approach, the model checker creates the entire state space of the global states, whereas in (b) our proposed local approach, the network element is eliminated from the stored states and the model checker keeps track of only local states. 58

Figure 4.2 A simple distributed tree algorithm. Node 0 sends a message to all its children. Each node forwards the message to its children. 60

Figure 4.3 The global state space of the example tree in Figure 4.2 as explored by a global model checking approach. The initial local state of each node is denoted "-". The state of node 0 and 4 is changed to "s" and "r" after the send and receive of the message, respectively. The network element of the global state is represented by the set of in-flight messages. Each arrow depicts a transition in the model checker from one global state to another. The label besides each arrow indicates the event that triggers the transition. Although the global states inside the rectangles are duplicates, they are not joined into one state, for simplicity of presentation. . 61

Figure 4.4 Local model checking approach of the example tree in Figure 4.2. The first column indicates the changes into the shared network element. The middle column shows the set of local states of node 0 to 4. The initial local state of each node is denoted "-". The state of node 0 and 4 is changed to "s" and "r" after the send and receive of the message, respectively. The first event is the local event of node 0 that generates the message. The generated message is then added to the shared network element. At each step, an event is selected and is executed on all local states of the destination node. The resultant states are added to the list of visited local states if they have not been visited before. The last column shows the new system states created after each step. 62

Figure 4.5 In our local approach, the handler execution works only on local states and produces new local states. Local and system states are denoted "LS" and "SS", respectively. The messages are not removed from the shared network component after execution. The new system states are created after a new local state is produced. The soundness verification checks the validity of a system state, only after an invariant violation is reported. 63
Figure 4.6 The altered handlers in local model checking. 64
Figure 4.7 Local model checker algorithm. 65
Figure 4.8 The elapsed time in model checking Paxos where only one out of three nodes proposes a value. 72
Figure 4.9 The number of explored states. 73
Figure 4.10 The consumed memory. 75
Figure 4.11 The overheads of LMC in model checking Paxos in which a bug is injected. 76

Figure 5.1 Non-uniform latency in inter-core communication; Cores C_0 and C_1 share the same L2 cache and communicate much faster than Cores C_0 and C_3 that have to go through the interconnect network. 82
Figure 5.2 The reduced number of messages in PaxosInside compared to collapsed Multi-Paxos deployed on three nodes. The dotted box represents the node boundary. The dashed messages, which do not cross the node boundary, do not consume the node bandwidth. P, A, and L represent the proposer, acceptor, and learner roles, respectively. The grayed acceptors and consequently the communications to/from them are eliminated in PaxosInside. 85
Figure 5.3 The interaction between nodes in Basic-Paxos. This example consists of one proposer, three acceptors, and two learners. In Multi-Paxos, the leader skips the first phase, i.e., prepare_request and prepare_response. . . . 89
Figure 5.4 The interaction between nodes in PaxosInside to replace failed acceptor A with another backup acceptor A'. In Step 1, the leader makes sure that it is still known as the leader by a majority of the nodes. Then in Step 2, it announces the change of the active acceptor. Finally in Step 3, it sends a prepare_request message to the new active acceptor A'. 95

Figure 5.5 The interaction between nodes in PaxosInside when proposer P' takes the leadership position from leader P. In Step 1, proposer P' inquires for the active acceptor Id. It then announces itself as leader in Step 2. Finally in Step 3, it sends a prepare_request message to the active acceptor. 96
Figure 5.6 Two separate queues are used between each pair of cores. 98
Figure 5.7 The architecture of the implemented framework for message passing in manycore systems. 99
Figure 5.8 Throughput achieved by one client. 102
Figure 5.9 Commit latency observed by using one client. 102
Figure 5.10 The latency vs. throughput by increasing the number of clients. 103
Figure 5.11 The changes in throughput achieved by PaxosInside when the leader is slow. 104
Figure 5.12 The changes in throughput achieved by 2PC when the leader is slow. 104

Figure A.1 Reachable states of the system in Figure A.4 114
Figure A.2 A Dynamic Partial Order Reduction 114
Figure A.3 Consequence Prediction 114
Figure A.4 Example Service in Mace language 115

Figure B.1 PaxosInside Algorithm 118
Figure B.2 The implementation of Procedures init, getAny, and registerProposals, in PaxosInside Algorithm 120

About the Author

Maysam Yabandeh was born in Qom, Iran, in 1982. He received a diploma in physics and mathematics from National Exceptional Talents School, Qom, in 2000. He then started his undergraduate studies at Electrical and Computer Engineering department, University of Tehran (Fanni). He finished his bachelor in Software Engineering in 2005, after receiving the honorable award for achieving the first rank among the graduating students.

He achieved the forth place in the national entrance exam for graduate studies, in computer engineering field, in 2004. Following that he received the third place in National Student Olympiad, Computer Science field. He then started his graduate studies at ECE department, University of Tehran, where he finished his master thesis, "Concurrent Multipath Transferring in IP Networks", in 2007.

After two six-month internships at EPFL, Switzerland, he started his PhD at IC department. During his PhD he worked on various interesting topics, including software testing, model checking, and [Byzantine] fault-tolerant consensus protocols.