

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Systems and language support for building correct, high performance distributed systems

Permalink

<https://escholarship.org/uc/item/4gj3z4tw>

Author

Killian, Charles Edwin

Publication Date

2008

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

**Systems and Language Support for Building Correct, High Performance
Distributed Systems**

A Dissertation submitted in partial satisfaction of the
Requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Charles Edwin Killian, Jr.

Committee in charge:

Professor Amin Vahdat, Chair
Professor Tara Javidi
Professor Ranjit Jhala
Professor Bill Lin
Professor Alex C. Snoeren

2008

Copyright
Charles Edwin Killian, Jr., 2008
All rights reserved.

The Dissertation of Charles Edwin Killian, Jr. is
approved, and it is acceptable in quality and form
for publication on microfilm:

Chair

University of California, San Diego

2008

DEDICATION

To my wife Kristina, for all her love and support.

EPIGRAPH

*No man thoroughly understands a truth
until he has contended against it.*

—Ralph Waldo Emerson

Furious activity is no substitute for understanding

—H. H. Williams

TABLE OF CONTENTS

Signature Page	iii
Dedication	iv
Epigraph	v
Table of Contents	vi
List of Figures	x
List of Tables	xii
Acknowledgements	xiii
Vita and Publications	xiv
Abstract of the Dissertation	xv
Chapter 1 Introduction	1
1.1 Today's Distributed Systems	2
1.2 Challenges	6
1.2.1 Programming Languages and Abstractions	7
1.2.2 Runtime Variations	8
1.2.3 Heisenberg Uncertainty Principle	10
1.3 Hypothesis	11
1.3.1 Mace	13
1.3.2 MaceMC	13
1.3.3 MaceMC-Performance	14
1.4 Summary	14
Chapter 2 Related Work	16
2.1 Related Languages and Toolkits	16
2.1.1 MACEDON	16
2.1.2 State-Event Systems	17
2.1.3 Declarative Languages	17
2.1.4 Library Toolkits	17
2.1.5 Aspect-Oriented Programming	18
2.1.6 High-Level languages	18
2.2 Error Location Tools	19
2.2.1 Classical Model Checking	19
2.2.2 Model Checking by Random Walks	19
2.2.3 Model Checking by Systematic Execution	20
2.2.4 Model Checking by Abstraction	20

2.2.5	Dataflow Based Static Analyses	21
2.2.6	Isolating Causes from Violations	21
2.2.7	Model Checking for Real-Time and Hybrid Systems	21
2.2.8	Performance Error Detection	22
2.3	Summary	23
Chapter 3	Mace Design	24
3.1	Overview	27
3.2	Mace	34
3.2.1	Layers	35
3.2.2	Concurrency	37
3.2.3	Failures	41
3.2.4	Analysis	44
3.3	Experiences	46
3.3.1	Performance Evaluation	49
3.3.2	Undergraduate Course	51
3.4	Summary	51
3.5	Acknowledgement	52
Chapter 4	Lessons of the Mace Language	53
4.1	Architecture Design	54
4.1.1	Interfaces	54
4.1.2	Component Architecture	59
4.2	Service Specification	65
4.2.1	TypeDefinitions	66
4.2.2	Persistent State	71
4.2.3	Execution	76
4.2.4	Debugging	81
4.2.5	Miscellaneous	85
4.3	Options	89
4.3.1	Field	89
4.3.2	Method	91
4.4	Programming Notes	91
4.5	Summary	93
Chapter 5	Mace Modelchecker	94
5.1	System Model	96
5.1.1	Distributed Systems as Programs	97
5.1.2	Properties	97
5.1.3	Liveness/Safety Duality	99
5.2	Model Checking with MaceMC	101
5.2.1	Finding Violating Executions	102
5.2.2	Finding the Critical Transition	105
5.3	Implementation Details	107

5.3.1	Preparing the System	107
5.3.2	Mitigating State Explosion	110
5.3.3	Biasing Random Walks	112
5.3.4	Tuning MaceMC	115
5.3.5	Diverging Event Handlers	116
5.4	MaceMC Debugger	117
5.5	Experiences	120
5.5.1	RandTree	121
5.5.2	Chord	123
5.6	Bugs	125
5.6.1	RandTree	126
5.6.2	Pastry	132
5.6.3	MaceTransport	137
5.6.4	Chord	140
5.7	Summary	148
5.8	Acknowledgement	149
Chapter 6	Analysis of Performance	150
6.1	Background	153
6.1.1	From Model Checking to Performance Testing	153
6.1.2	Basic System Model	154
6.1.3	Dealing with Time	154
6.2	Model Checking	155
6.2.1	Searching	158
6.2.2	Analysis	160
6.3	Implementation Details	164
6.3.1	Preparing the Test	164
6.3.2	Pending Event Lists	165
6.3.3	Simulated Network	166
6.3.4	Large Log Files	167
6.4	Experiences	168
6.4.1	BulletPrime	169
6.4.2	Pastry	171
6.4.3	Chord	173
6.4.4	RandTree	174
6.5	Limitations	175
6.6	Summary	176
Chapter 7	Conclusions and Future Work	178
7.1	Contributions	179
7.2	Future Work	180
7.2.1	Static Analysis	181
7.2.2	Finding Liveness-Violating Executions	181
7.2.3	Machine-Learning Based Performance Testing	182

7.2.4	Online Monitoring and Debugging	182
7.3	Summary	183
	Bibliography	184

LIST OF FIGURES

Figure 1.1:	The Mace architectural design.	12
Figure 3.1:	Bamboo DHT design architecture.	29
Figure 3.2:	Mace service composition for a DHT application using Recursive Overlay Routing implemented with Bamboo. Shaded boxes (dark for downcall, light for upcall) indicate the interfaces implemented by the service objects.	34
Figure 3.3:	States and Transitions for Bamboo Service Object	38
Figure 3.4:	State machine model for Bamboo. States are shown in light gray; event-transitions are represented by arrows with names in white and actions in dark gray.	39
Figure 3.5:	Message diagram for global sampling. In response to a scheduled timer, Node <i>A</i> routes a <i>GlobalSample</i> message to an identifier <i>id</i> by sending it to node <i>B</i> , which is forwarded to its owner, node <i>C</i> . Node <i>C</i> responds with a <i>GlobalSampleReply</i> message informing node <i>A</i> about node <i>C</i> , potentially causing a routing table update.	40
Figure 3.6:	Local and Distributed inconsistency and failure detection in Mace.	40
Figure 3.7:	Percentage of lookups that return a consistent result.	47
Figure 3.8:	Mean latency for lookups that return consistent results.	48
Figure 4.1:	This figure shows how timeout periods and triggers interact in detect specifications in Mace. Time is represented by the arrow going to the right. Arrows pointing up indicate triggers executing.	81
Figure 5.1:	State exploration: we perform an iterative, bounded depth-first search (BDFS) from the initial state (or search prefix): most periphery states are indeterminate. We execute random walks from the periphery states and flag walks not reaching live states as suspected violating executions.	101
Figure 5.2:	Compared performance of random walks with and without biasing.	114
Figure 5.3:	CDF of simulator steps to a live state at a search depth of 15.	115
Figure 5.4:	MDB session. Lines with differences are shown in italics (– indicates the error log, + the live log), with differing text shown in bold. The receiver is IP address 1.0.0.1 and the sender is 2.0.0.1.	118
Figure 5.5:	Automatically generated event graph for MACETRANSPORT liveness bug.	119

Figure 6.1:	System Architecture: first, the unmodified system is run through a set of training runs to determine a set of event duration distributions (EDD). Next, the system, along with the EDD, is fed into the Search algorithm, which will produce one or more anomalous paths. We then use the ExecutionSearch tool to analyze the anomalous path, giving us the most similar “normal” path. At this point, any debugging tool can be used to compare the two executions until the source of the bug is found.	161
Figure 6.2:	We first perform an exponential search ($E_1 - E_5$) to determine bounds for the divergence point, then a binary search ($B_1 - B_3$) to isolate the divergence point.	163

LIST OF TABLES

Table 3.1:	Lines of code measured in semicolons for various systems implemented in Mace and other distributions.	46
Table 5.1:	Example predicates from systems tested using MaceMC. <i>Eventually</i> refers here to <i>Always Eventually</i> corresponding to Liveness properties, and <i>Always</i> corresponds to Safety properties. The syntax allows a regular expression expansion ‘*’, used in the AllNodes property.	98
Table 5.2:	Summary of bugs found for each system. LOC=Lines of code and reflects both the Mace code size and the generated C++ code size.	122

ACKNOWLEDGEMENTS

I would like to acknowledge all of the people who have helped bring this work to fruition. Adolfo Rodriguez, from whom I inherited the MACEDON project which led to the development of Mace, Dejan Kostić, who helped drive many of the early requirements of the project, James Anderson, who helped build Mace and MaceMC, and co-authored each of these papers, Ryan Braud, who helped build and author Mace, and co-authored that paper, my faculty co-author Ranjit Jhala, and my co-author and advisor, Amin Vahdat. Additionally, this work would not have been so successful without the help of fellow students and users of the Mace toolkit, especially Meg Walraed-Sullivan, Justin Burke, Alex Rasmussen, and Flavio Junqueira, nor without the contributions of various smaller course projects, undergraduate research projects, and Masters projects, especially those of Duy Nguyen, Hakon Vesperej, Jim Hong, and Darren Dao. Thanks also to the shepherds and anonymous reviewers of our various papers, especially to Petros Maniatis of Intel Research. Not to be forgotten also are the staff and systems faculty at UCSD who contributed to the growth of this project and helped make sure the resources were available and functioning properly. Special thanks in this regard to Ken Yocum, Marvin McNett, Chris Edwards, Stefan Savage, Alex Snoeren, and Geoff Voelker.

Chapter 3 is an updated and revised copy of the paper by Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat, titled “Mace: Language Support for Building Distributed Systems,” as it appears in the proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation 2007. ©2007 ACM DOI <http://doi.acm.org/10.1145/1250734.1250755>. The dissertation author was the primary researcher and author of this paper.

Chapter 5 is an updated and revised copy of the paper by Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat, titled “Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code,” as it appears in the proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation 2007. The dissertation author was the primary researcher and author of this paper.

VITA

2002	B. S. in Computer Science, B. S. in Applied Math, with honors, <i>summa cum laude</i> , North Carolina State University
2004	M. S. in Computer Science, Duke University
2008	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

Dejan Kostić, Alex C. Snoeren, Amin Vahdat, Ryan Braud, Charles Killian, Jeannie Albrecht, James W. Anderson, Adolfo Rodriguez, and Erik Vandekeft. “High Bandwidth Data Dissemination for Large-scale Distributed Systems.” *ACM Transactions on Computer Systems*. (to appear) 2008.

Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. “Mace: Language Support for Building Distributed Systems.” In proceedings of *Programming Languages Design and Implementation*. June 2007.

Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. “Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code.” In proceedings of *Networked Systems Design and Implementation*. April 2007. Awarded Best Paper

Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. “Pip: Detecting the Unexpected in Distributed Systems.” In proceedings of *Networked Systems Design and Implementation*. May 2006.

Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekeft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. “Maintaining High-bandwidth under Dynamic Network Conditions.” In proceedings of *USENIX Annual Technical Conference*. April 2005.

Charles Killian, Frank Ruskey, Carla Savage, and Mark Weston. “Half-Simple Symmetric Venn Diagrams.” *Electronic Journal of Combinatorics*. 2004.

Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. “MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks.” In proceedings of *USENIX/ACM Symposium on Networked Systems Design and Implementation*. March 2004.

Jerry Griggs, Charles Killian, and Carla Savage. “Venn diagrams and symmetric chain decompositions in the Boolean Lattice.” *Electronic Journal of Combinatorics*. Volume 11, January 2, 2004. (An article about this result appeared in Science, Vol. 299, January 31, 2003 and it was the subject of a front page article in the January 2004 issue of SIAM News. Additionally, this work was featured in the December 2006 issue of the Notices of the AMS.)

Charles Killian, and Carla D. Savage. “Antipodal Gray Codes.” *Discrete Math.* Vol. 281, Nos. 1-3 (2004) 221-236.

Feiyi Wang, Raghu Uppalli, and Charles Killian. “Analysis of Techniques For Building Intrusion Tolerant Server Systems.” In proceedings of *Military Communications Conference*. Oct 13-16, 2003.

Feiyi Wang, and Charles Killian. “Design and Implementation of SITAR Architecture: A Status Report.” In proceedings of *Intrusion Tolerant System Workshop, Supplemental Volume on 2002 International Conference on Dependable System & Networks*. Washington D.C.. June 22-26, 2002. (unrefereed)

ABSTRACT OF THE DISSERTATION

Systems and Language Support for Building Correct, High Performance Distributed Systems

by

Charles Edwin Killian, Jr.

Doctor of Philosophy in Computer Science

University of California San Diego, 2008

Professor Amin Vahdat, Chair

Daily life involves the use of computers for everything from interpersonal communication to banking and transportation. But while everyday computation has become more decentralized and disconnected, advances in programming and debugging have centered on individual processes. It is still very challenging to write correct, high-performance distributed systems. Programmers can choose either to sacrifice correctness by accepting the complexity of building a distributed system from the ground up, or to sacrifice performance by using generic toolkits and languages which provide simplifying functionality like RPC, memory transactions, and serialization, which makes it easier to code correct systems. Where performance is deemed higher priority, systems are generally built using C++, and debugged by printing logs at each node and using ad-hoc tools for analysis, leading to complex, brittle implementations.

This dissertation posits that language support can significantly simplify the development of distributed systems without sacrificing performance, and can enable analysis to automatically find and isolate deep bugs in implementations affecting both

performance and correctness of distributed systems. We focus on finding a middle-ground between the canonical distributed systems design abstraction, state machines, and the classical programming tools used for modern high-performance distributed systems, C++, awk, sed, and gdb. That middle ground is a C++ language extension wherein users implement a distributed system using syntax yielding the performance and control of C++, but in a restricted programming model forcing them to structure their system as a state machine.

This dissertation presents the Mace language extension, runtime, and tools. Mace makes it possible to develop new high-performance distributed systems in a fraction of the time. We implemented more than 10 significant distributed systems using Mace, each both shorter and as fast, or faster, than the original. Structured programming in Mace also enabled development of the first model checker capable of finding liveness violations in unmodified systems code, and an automated performance tester to detect and isolate anomalies. These have been used to find and isolate previously unknown or elusive bugs in mature Mace implementations. Mace has been publicly available for four years, and is used worldwide by academic and industrial researchers in their own research.

Chapter 1

Introduction

Distributed systems are critically important today. Computers that control everything from stock markets and communication networks to transportation systems and hospitals are increasingly interconnected, with distributed computation and communication vital to their correct functioning. Moreover, these systems must execute not only correctly, but at a level of performance that significantly limits their implementation options. This combination of correct execution at high performance makes them a challenge to program. Despite the best efforts of systems developers over the past 30 years, we still produce systems with latent bugs or issues that are often discovered only by the end-user in production systems. Alternately, the systems may be inefficiently designed or under provisioned, causing them to perform poorly during periods of high load.

Failures of these distributed systems, while somewhat rare, are often spectacular and become headline news. There are the cases in which distributed systems failures cause the markets to cease trading [And05, Pre06], or their indexes to be incorrectly maintained [PLS07]. There are also cases in which communication systems, either telephonic [Gar05] or internet-based [Ara07] suffer complete failure due to bugs. In other cases, network connectivity failures or high load halts transportation systems [Car07, Pre07] and hospitals [Pre03].

1.1 Today's Distributed Systems

The field of modern distributed systems is about thirty years old. One of the first such systems, Grapevine [SBN83], was an experiment in distributed systems, and provided the first naming and service location system. Since its development, communication has become even faster and more widespread. Before, it would have been inefficient to break a computation or job into small pieces divided across vastly distributed resources, because the communication overhead of the pieces would quickly dominate the overall task, actually causing it to perform slower than traditional computation. But today, with new, fast, widely deployed networks, it makes sense to share the load across resources potentially distributed internationally.

Computers and distributed systems play a role in a wide variety of industries, and an entire survey could be written about them. This introduction will focus on just three industries whose distributed systems, and their failures, are well known throughout the world: financial, communication, and transportation systems.

Financial Systems

Banking systems are tightly interconnected, including e-check processing systems, credit card processing systems, ATMs, branch offices, and so forth. Before these distributed systems, but after the aforementioned forms of payment were introduced, sellers accepting these payment types were often at risk that the payment would not go through, and the seller would have to track down the buyer to collect money. Now, distributed systems have been established which connect the sellers and processing systems to the bank accounts. Verification takes place instantly, ensuring that the buyer has the money he/she is trying to spend. These systems are especially critical to program correctly because they are tracking someone's hard-earned money, and errors usually either mean the bank loses or the client loses, because of incorrect balances. Plus, there must be a coordination process between these systems, because the customer may be conducting several transactions at once, and expects a consistent view of what is going on in the bank account. Thus it is important to verify, given all the ways two transactions might interleave, that the balances match up with the transaction history. One common problem occurs if two processes read the balance

at the same time, each apply their transaction to that balance, then write the value back. If there is no detection of the concurrent transactions, then the first transaction to complete will essentially be lost when the second one completes.

But beyond just tracking an individual bank account, today's distributed systems also control the stock market. Stock values, purchase requests, and sell requests are all exchanged in a high-volume trading system around the world, both in private networks for brokerage firms and on the public internet. Designing distributed systems that effectively handle the high trading volume is a rather challenging task. Added to this burden is the addition of *program trading*, which in essence is when a computer program is written to automatically trade stocks under a set of conditions for which it then monitors. These two properties can in fact combine to produce conditions where the market races out of control, with the program traders adding significant load to the system as a whole, leading to a vicious cycle. Take for example a program trader that detects a significantly declining market and sells off stocks to get out before they decline too much. The program trader is unwittingly contributing to the decline of the market, and thus triggering other program trading with similar policies. This series of events causes the trading volume to increase overall, stressing market systems.

Safeguarding our market system against overloaded systems, software bugs, and instabilities in program trading is an immensely complicated task. It has involved backup systems, monitoring systems, over-provisioned networks, and the creation of mechanisms to halt the exchange when trading gets out of control—to prevent collapse of the economy. These systems must be maintained, tested, and themselves debugged, to avoid recovery mechanisms that fail as soon as they are deployed, and the software testing process for these systems must aggressively defend against a whole host of possible problems based on seemingly random and unknown user interactions.

Communication Systems

Communication systems supporting person-to-person communication are also a kind of distributed system, and are often dual-purpose to support a wide variety of other distributed systems. These systems can be low-level, such as traditional

telephony systems, which have dedicated resources for a continuous data flow, or they can be more modern software communication packages, such as voice-over-IP provider Skype, which provide streaming media using Internet best-effort service.

Once simple, telephone systems connected calls through the use of human operators who physically connected patch cables to connect callers to receivers. Since that time, call traffic has been modified to be transmitted along with other call traffic sharing a fiber-optic channel, and routed using automated electronic switching stations. The software for these systems is notably well tested and the PSTN (public switched telephone network) service generally delivers what is still considered the gold standard of 5-nines (99.999%) of reliability, better than other networks and distributed systems. Yet even in this environment, buggy software still manages to find its way to deployment, as evidenced in the 1990 failure of the AT&T long-distance switching system. In this instance, a bug in the software caused the long-distance switch to reboot, and in doing so, caused neighboring switches to suffer the same fate. The result was a nationwide long-distance outage over a period of about 8 hours. The bug was exercised by a certain timing of events that had not been tested, suggesting that existing tools for testing these distributed systems were not considering the range of possible problems.

More modern communication systems can deliver the same type of service as traditional telephone service, but do so by splitting communication media into little pieces, and shipping each of the pieces to destinations using the public Internet. The most popular such service is Skype, a voice-over-IP provider with 220 million users, 5 to 6 million of whom are generally online at any given time. To save Skype's bandwidth, and ultimately money, users of Skype donate a portion of their bandwidth to other users, routing calls around circumstances that prevent direct communication between users. Accordingly, the Skype client executes a peer-to-peer membership protocol to learn which peers are online, and to support the location of peers for indirect routing. However, the Skype software designers had not properly accounted for how the protocol would tolerate a massive restart of users' computers at roughly the same time, both drastically reducing the available network resources and increasing the flood of join attempts. This scenario became tested in deployment after a routine download of updates to Windows boxes through Windows Update caused a

large quantity of Skype users to restart their computers at nearly the same time. The result was a Skype outage affecting all users for a three day period. Again, the environment and methodology for testing this software was not adequate to support the scope of scenarios that might exist in deployed environments.

Transportation Systems

However, distributed systems do not always fail due to software bugs. They can just as easily fail due to misconfiguration or under provisioning. So, it is not sufficient to build tools that search for bugs in the programming itself, but instead the tools must consider how interactions with the environment or variations in the timing of events or the offered load might affect the system. Additionally, tools must be built to enhance our ability to perform post-mortem diagnosis of what happens in the deployed system, since inevitably we cannot solve all problems before deployment. This is the case of two distributed systems supporting transportation systems, which normally manage the safe and efficient transportation of people and cargo over rail and through the air.

In both the cases of rail-management systems and flight-tracking systems, operators are concerned with the efficient routing of moving vessels from source to destination, but they are primarily concerned with the safety of those moving vessels and their passengers. Unlike vehicular systems, in which routing and safety decisions are made by independent drivers on the roadways, for railroad and airplane systems these decisions are made under the supervision of a separate authority who handles the routing and scheduling of all vessels through a region of space; therefore, it may make more efficient and globally optimal decisions.

Of course, a primary requirement of these systems is that the information about where vessels are and where they are going must be made available to the operators making the decisions. In the case of a June 2007 aviation problem, a computer in the Atlanta system for processing flight plans and sending them to air-traffic controllers failed. To avoid shutting down, the traffic was diverted to the Salt Lake City center. However, the load proved to be too much for the Salt Lake City center, causing hundreds of flights to be cancelled or delayed.

In a similar scenario, an engineer for Caltrain in San Francisco was attempting to complete a network connection to a new central operations facility when the existing connection to the old central operations facility was disrupted. Initially blamed on a computer glitch, the inability to tell where the trains were and to signal them caused them to revert to their failsafe mode, which is to stay in their current positions. As a result there were no injuries, but at least 8 trains were stopped on the track during rush-hour commuting.

Clearly, failures in existing distributed systems, whatever the cause, have a huge social and financial impact on society. Failures can result from buggy software, hardware, under-provisioning, user error, malice or misconfiguration, or more often, a combination of multiple contributing factors. Consider next some of the specific challenges of building and testing that are unique to distributed systems.

1.2 Challenges

There are many challenges to building correct, high-performance distributed systems. Distributed systems must deal with network errors, disconnections, node reboots, and widely varying latencies and throughputs. Each node in a distributed system can only see its local state, and can only query other nodes' state through network messaging. This messaging has an inherent delay, so this report may not reflect current state, making it harder to coordinate and collaborate across nodes. Furthermore, as multiple nodes communicate with each other, the orderings of these messages are unpredictable, despite the expectation that the message sent first will be received first. In fact, the triangle inequality also does not apply: given three nodes A , B , and C , if A sends messages to B and C , then B forwards a message to C , the message from A to B to C might arrive before the message from A to C . These unexpected orderings make it challenging to get the algorithms correct, but they also impact performance. Performance problems can be caused by bugs in the code that are exercised when state is inconsistent across nodes. To add insult to injury, the nature of high-performance and real-time distributed systems also makes it impractical to include even modest amounts of logging overhead in a deployed

system, and therefore it is nearly impossible to piece together what happened during execution even with logging enabled.

1.2.1 Programming Languages and Abstractions

Among the many challenges in building these systems is that currently, programmers are forced to choose between simple expression and control over performance. Simple expression is obtained by using high-level programming languages and/or communication services, which simplify the construction of networked systems with general purpose abstractions. However, these general purpose abstractions tend to have lower overall performance, either due to higher overall load or the fact that they have to do more work than necessary to accommodate for all the ways the service *might* be used.

Take, for example, JAVA serialization. JAVA serialization is built under the idea that there is an object stream, and that the process reading from the object stream need not know the type of the object to be received next. In fact the next object could be *any* JAVA object, even those not familiar to the local JVM. Additionally, each field of the object may be at runtime replaced with another JAVA object that is a derivative of the actual field type. The implication of this design decision is that the serialized object stream must contain complete type information for each object and field. But, for practical distributed systems, this wholly general feature is unnecessary, as the messaging protocol will define with only few options what each field and object in the stream should be, making the actual serialized form much more simple. The more general JAVA form is simpler to code and use, but comes at the cost of larger serialized message sizes, and more CPU cost to serialize and deserialize the object.

Simple primitives for shared memory infrastructure or RPC can similarly be used to build a wide variety of communication patterns. But often using these primitives for building other abstractions is less than ideal from a performance standpoint, for much the same reasons as JAVA serialization is not ideal. So while researchers and prototypes sometimes use these services and languages, current distributed systems programmers often tend to work closer to the other end of the spectrum, choosing

a programming language that gives them complete flexibility over the performance and details of each component. But these programming languages require the programmer to focus on every detail, not just the ones that matter for the distributed system. The resulting large body of code leaves many cases where mistakes may be made, which leads to bugs and errors. Plus, much of the code is tedious, following a copy-paste-edit paradigm, where the same pattern is to be used, but must be tailored slightly differently in each case. If a problem is later found in the pattern, each case must be visited and fixed, and missed cases can also lead to bugs.

Also, the simpler high-level abstractions are easier to understand and reason about, which helps those implementations be more correct than they would be using low-level abstractions. But the practical systems are generally much more complex, and cannot be reasoned about in any convenient way. It is possible to build tools to process added instrumentation data collected from an actual execution of a specific system, but these tools and systems then have to be organically grown for each component needed, as opposed to a unified framework which is possible for higher level systems. Of course, when it comes to debugging, even the process of adding instrumentation can cause the result to change, making it impossible to reproduce the problem exactly. So we either seek a way to debug the system without keeping track of what it is doing, or to make the constant instrumentation overhead small enough to not affect the result.

1.2.2 Runtime Variations

Concretely, consider the challenges in building a simple distributed hash-table (DHT). A distributed hash-table distributes the buckets of a hash-table across nodes throughout the network. In doing so, it makes it possible to build a load-balanced system that distributes key-value pairs throughout the network. To tolerate the inherent churn in the network, DHTs replicate data at peer nodes, so that if nodes reboot, they download fresh copies of the data to prevent loss. Furthermore, since the DHTs are designed to be quite large, they maintain routing tables that enable efficient lookup of any particular datum. Generally, this means that from any node in

the DHT, any key-value pair can be determined by routing a query through at most $\lg n$ peers, where n is the total number of peers.

However, there are many problems which can occur that prevent this design from being met. First, consider a reboot operation. It will download a fresh copy of the data from a peer by design, but consider also a simultaneous query. The programmer naturally assumes the copy of the data will be received first, and tends not to consider the rather unlikely event that a query is received before the rebooting node can download the data again. But if the query does arrive first, a naive implementation of the DHT would respond with the incorrect response that the key has no value. This result is due to the fact that the node is a bucket maintainer for that key, and does not have any data for it. In this case, the ordering of messages is critical to whether a bug will be exercised. A programmer gets progressively worse at considering all the possible scenarios as the number of nodes grows larger, causing an exponential state space explosion.

Another problem occurs early on in the DHT execution, when the routing tables are incomplete and only basic routing is available. In this event, nodes do not know other good nodes to which to forward data, so the path of these queries can be quite long. The designer's view of this problem is that it is a "startup" problem only, and in the steady state is not a concern. But, this view overlooks problems where the design for the construction of routing tables takes an excessive amount of time, and becomes a problem for the average case.

A final example problem occurs when, due to node churn, the network is left in a bad state, and a routing loop forms. These loops can cause lookup messages to be forwarded indefinitely through the system. Further churn can eliminate the loops, "correcting" the system, and allowing the lookup message to complete. Instead of observing a correctness problem, an operator would instead observe a significant performance problem. Similarly, if the implementation language has high overhead in terms of CPU or memory usage, or certain machines are under-provisioned or overloaded, the performance of the whole system will degrade. This degradation is because in a distributed system, the complete operation involves several serial steps of nodes waiting for other nodes to complete, and the whole chain is brought down by the weakest link. This degradation is more subtly true in parallel systems that must

wait for the total response set before continuing—the slowest response will dominate the performance of the overall system.

1.2.3 Heisenberg Uncertainty Principle

Determining what is happening in a distributed system is much like locating a particle in a small region of space. Generally, users can only see the macro outputs—that is, those outputs that are results of application behavior. To get a better look at what is precisely happening in a distributed system, the state of the art is to add logging to each node in the distributed system. Increasing the amount of logging generally leads to equivalent increases in the visibility into the actions at each node.

But adding even small amounts of logging subtly affects the timings of events in the system. This modification may prevent or mask problematic behaviors of the system, making the error condition impossible to reproduce, even if the error condition is observed. Additionally, too much logging can slow down the performance of the system noticeably, an unacceptable outcome for a deployed system. Thus, distributed systems developers expend a lot of energy in making sure their logging has a minimal impact on the system, and can easily be disabled to achieve the best performance.

Even after instrumenting distributed systems, determining what is happening in the distributed system as a whole is still a challenging problem. Each node generates its own independent log file. These log files are kept on a local disk, so network I/O from logging does not impact system performance. Understanding what happened in a slow query, for example, involves tracing the query message from the source computer's log, matching up each message sent and received, skipping from log to log on different computers, and trying to piece together what happened.

Often, the exact trace of the message path is not enough to understand what happened, as more information is needed about the state of each node when it processes messages, and the other things going on simultaneously at the node. Understanding the state of the system often comprises relational properties of the variables at each node, collected globally. For example, to see the state of a tree, one would construct a map of the node each node considers its parent. Further, since clocks are

not synchronized across nodes, it is non-trivial to even produce a consistent snapshot of the system state.

1.3 Hypothesis

The hypothesis of this dissertation is that language support can significantly simplify the development of distributed systems without sacrificing performance, and can enable analysis to automatically find and isolate deep bugs in implementations affecting both performance and correctness of distributed systems. This dissertation provides a partial solution to the problem of building correct, high performance distributed systems, that both advances the state of the art, and is a practical solution that can be used in addition to and along with existing tools, systems, and programming experience.

- First, this dissertation argues that a new programming language can be developed for the domain of distributed systems, that allows programmers to easily write simple, efficient implementations that contain semantic information allowing the compiler to handle tedious code, generic tools that provide useful functionality, and automatically instrument the code with event processing and tracing.
- Second, this dissertation argues that to effectively test distributed systems, it is critical to test liveness properties, which ensure not only that the system never enters a bad configuration, but that it *eventually* accomplishes its goals. Further, the dissertation argues that it is possible to extend existing model checking techniques to effectively test these properties on unmodified implementations. Using model checking also avoids the problem of determining what happens during execution and understanding log files, because a model checker runs the unmodified code in simulation, where the model checker, and not the performance of the node, controls the interleaving of events.
- Third, this dissertation argues that the ideas and concepts of model checking may be extended to develop a new set of tools which can support the location of performance problems in distributed systems implementations. Specifically,

these tools can help find bugs in implementations that are masked by robust engineering, and would otherwise be ignored by correctness checking.

To this end, we have developed the Mace programming environment for distributed systems, the MaceMC model checker that can test liveness properties, and extended MaceMC to help find and isolate performance problems in unmodified distributed systems implementations. Figure 1.1 illustrates the Mace architectural design. The user writes the state-machine representation and the interface description of a distributed system service, which the Mace compiler translates into a high-performance implementation, debugging hooks, and its annotated structure. These pieces can then be used equally by user applications and tools designed to work generically with Mace implementations such as the MaceMC model checker and its extensions.

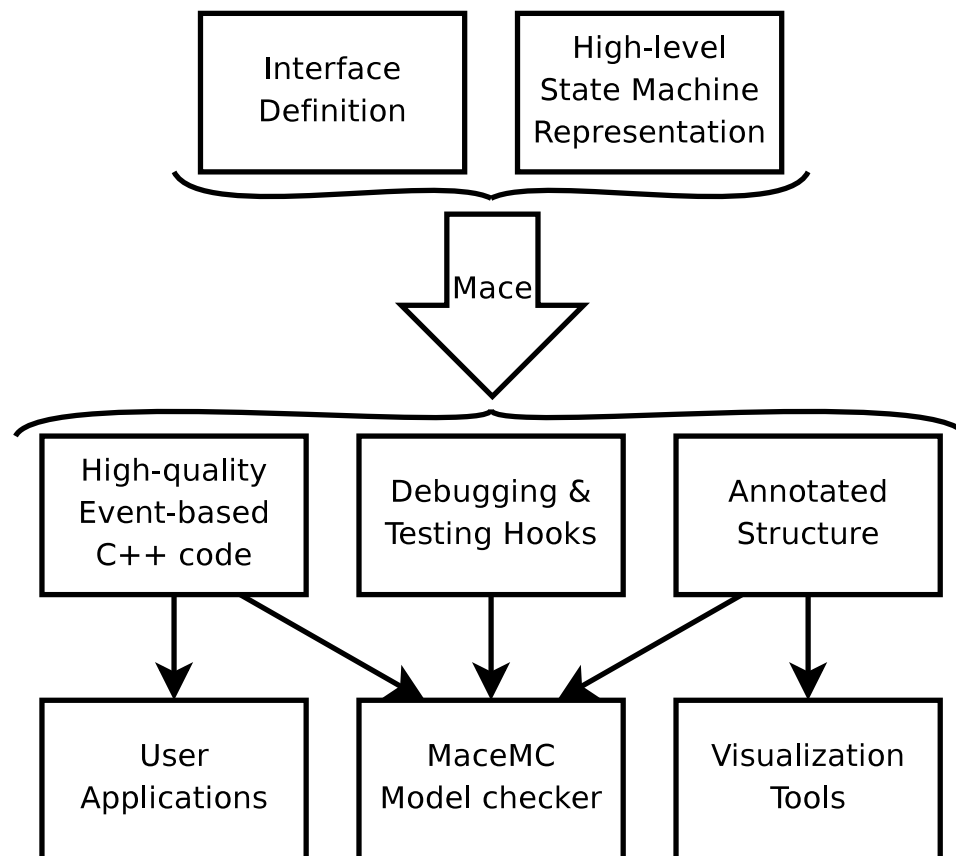


Figure 1.1: The Mace architectural design.

1.3.1 Mace

We developed the Mace programming language, runtime, and toolkit to simplify development of efficient, high-performance distributed systems implementations. Mace is fully operational, has been in development for five years, is publicly available for download, and has been used by researchers at UCSD, HP Labs, MSR-Asia, and a handful of universities worldwide in support of their own research and development. We have implemented more than ten significant distributed systems in Mace, most of which were originally proposed by others. This set includes Distributed Hash Tables [RGRK04a, RD01, SMK⁺01], Application Layer Multicast [CDK⁺03, JGJ⁺00, KBK⁺05], and network measurement services [DCKM04] ready to run over the Internet. In each of these cases, the Mace implementation was as fast or faster than the original implementation, and two to ten times shorter in source code. We have also used Mace in multiple graduate and undergraduate courses as a teaching tool, where students successfully used Mace as an implementation language for course projects.

1.3.2 MaceMC

To accompany the Mace compiler and runtime, we built many tools supporting debugging, each taking advantage of the support available from the domain-specific language. In particular, we built the Mace Model Checker (MaceMC), to support systematic automated testing of unmodified Mace implementations in the many scenarios to which each system could be exposed, simplifying the task of finding bugs. MaceMC can also test whether systems violate liveness properties. Once a violation is found, MaceMC can identify the specific point where an execution becomes *dead*, classifying an entire set of executions as liveness-violating, and directing the developer to the specific problem that caused the execution to violate liveness. MaceMC has been used to find more than 50 bugs in Mace implementations, and is part of the active development cycle for all core Mace developers.

1.3.3 MaceMC-Performance

We also developed a variant of MaceMC to help the developer search for performance problems. Performance problems are more difficult to detect automatically, because they do not have black-and-white boundaries. They require testing that considers a wide variety of probable or possible timings of events. This new variant takes input timing distributions and generates fully-reproducible time-based executions. It can then take anomalous executions and explore the space of similar executions, helping the developer understand where in the execution a performance problem may have occurred. This new technique for finding performance problems has allowed us to find an interesting new class of bugs—correctness bugs that are masked by robust design. For example, a bug sometimes preventing the construction of an overlay tree may get masked by a protocol designed to recover from network partitions. This bug would be overlooked by the original MaceMC, but found by the enhanced version since it would cause anomalous performance.

1.4 Summary

Mace, MaceMC and the performance tester have each advanced the state of the art towards easily building correct, high-performance distributed systems. It now takes experienced Mace developers a fraction of the time previously needed to implement a new distributed system once its design is conceived, and debugging and testing are greatly simplified thanks to the model checker and performance tester. Mace represents five years of development work and has been publicly available for four years. In addition to the Mace research contributions, the Mace distribution also represents many of the best-quality publicly-available implementations of the included services, and by itself represents a practical contribution that users worldwide recognize and utilize.

This dissertation presents Mace, MaceMC, and the performance-testing extension to MaceMC. Chapter 2 explores the space of related work. Chapter 3 describes the design of the Mace language and runtime, while Chapter 4 describes the language in detail. Chapter 5 describes the MaceMC modelchecker, and Chapter 6 describes

the variant of MaceMC that can find performance problems. The dissertation concludes with Chapter 7, which describes future directions and open problems.

Chapter 2

Related Work

The Mace toolkit includes a new domain-specific programming language, libraries, a model checker, and various tools for performance testing and monitoring. It is therefore closely connected to a large body of work in the area of languages, libraries and toolkits for building concurrent systems. We, however, focus our attention on those specific to distributed systems. It is also related to other systems for testing, and in particular, model checkers.

2.1 Related Languages and Toolkits

The features developed in the Mace language are not wholly new. Rather, Mace eclectically composes elements of language design available in other languages to develop a language suitable for high-performance distributed systems.

2.1.1 MACEDON

Mace builds upon the earlier MACEDON [RKB⁺04] work—a domain-specific language for fair comparisons of overlay systems. MACEDON also represents systems as I/O automata, but does not consider how compiler extensions that restrict specifications can support model checking, high performance, debugging, etc. Whereas MACEDON focused on building prototype lab experiments, we designed Mace as a practical, real-world environment for developing deployable high-performance, reliable applications.

2.1.2 State-Event Systems

The *state-event-transition* model on which Mace is founded is closely related to other event-driven languages and libraries. NESC [GLvB⁺03] is a language for building sensor networks with limited resources requiring static memory allocation. Broadly speaking, several researchers have investigated language support for building concurrent systems out of interacting components, such as CLICK [KMC⁺00] for building routers from modules and the FLUX OSKIT [FBB⁺97] for building operating systems. These approaches, however, target concurrent systems executing within one physical machine rather than distributed systems scattered across a network.

2.1.3 Declarative Languages

P2 [LCH⁺05] is a declarative, logic-programming based language for rapidly prototyping overlay networks by specifying data-flow between nodes using logical rules. While P2 specifications are substantially more succinct than those of Mace, we feel the corresponding specification is not as natural to programmers. Additionally, the P2 authors admit their runtime sacrifices performance—performing only within an order of magnitude for the best case. Also, while Mace is well-suited for building overlays, its applicability is broader. There is a line of work in the functional programming community for advanced, type-safe languages for distributed computation [SLW⁺05]. At the moment these languages are somewhat experimental, emphasizing full understanding of the semantics of high-level constructs for distributed programming and their interplay with the type system rather than enabling the rapid deployment of robust, high-performance distributed systems.

2.1.4 Library Toolkits

Several libraries and toolkits also support many of the common primitives required for building distributed systems. LIBASYNC [Maz01] uses a single-threaded event-driven model that makes extensive use of callbacks. LIBASYNC also provides some compiler support for dealing with remote procedure calls and for generating some of the serialization code for messaging. Another instance, SEDA [WCB01],

provides an architecture for event-based systems. Both of these systems focus on simplifying the implementation of event-driven code, rather than a structure for distributed systems.

2.1.5 Aspect-Oriented Programming

Mace uses ideas proposed by Aspect-Oriented Programming (AOP) [Kic96]. One of the first examples of AOP was a domain-specific language for writing distributed software [Lop96]. One primary contribution of Mace is the identification of different *concerns* that comprise a distributed system—e.g. the messages, events, transitions, failures, and logging—and designing a language that enables programmers to think about these in isolation. The Mace compiler seamlessly puts each of these together to create an efficient implementation of the system. An immediate payoff of this separation is the ease with which a programmer can log and monitor entire event flows without cluttering the code with print statements.

2.1.6 High-Level languages

There are several high-level languages for describing network protocols, rather than entire distributed systems. Some of these, such as LOTOS [BB87] and ESTELLE [BD87], are intended largely to formally specify protocols using message passing finite state machines. PROMELA [Hol03] and TLA [Lam02] are two more general languages that can be used to model concurrent systems. Instead of producing executable systems, they compile the description into large finite state machines to exhaustively analyze for errors. RTAG [And88], based on grammars, and PROLAC [KKM99], based on an object-oriented model, are two examples of protocol description languages that actually compile the description into executable code. Mace combines the benefits of both approaches by structuring the description of the system such that the subsequent compiled implementation is amenable to exhaustive analysis.

2.2 Error Location Tools

Our work is related to several techniques for finding errors in software systems. Some of these techniques fall under the broad umbrella of model checking. These can be further classified as *classical* model checking, model checking by *systematic execution*, and model checking by *abstraction*. Another related approach is the use of dataflow-based static analyses and the analysis of program text to find errors.

2.2.1 Classical Model Checking

“Model checking,” i.e., checking a system described as a graph (or a Kripke structure) was a model of a temporal logic formula independently invented two 1981 research papers [CE81, QS82]. A Turing award was recently awarded for this research result. Advances like *Symmetry Reduction*, *Partial-Order Reduction*, and *Symbolic Model Checking* have enabled the practical analysis of hardware circuits [McM00, AHM⁺98], cache-coherence and cryptographic protocols [DDHY92], and distributed systems and communications protocols [Hol97]. SPIN [Hol97] introduced the idea of state-hashing used by MaceMC. However, the tools described above require the analyzed software to be specified in a tool-specific language, using the state graph of the system constructed either before or during the analysis. Thus, while they are excellent for quickly finding *specification* errors early in the design cycle, it is difficult to use them to verify the systems *implementations*. MaceMC by contrast tests the C++ implementation directly, finding bugs both in the design and the implementation.

2.2.2 Model Checking by Random Walks

West [Wes86] proposed the idea of using randomization to analyze networking protocols whose state spaces were too large for exhaustive search. Sivaraj and Gopalakrishnan [SG03] propose a method for iterating exhaustive search and randomization to find bugs in cache-coherence protocols. Both of the above were applied to check safety properties in systems described using specialized languages which yield

finite state systems. In contrast, MaceMC uses randomization to find liveness bugs, and determine where in an execution a liveness violation occurred.

2.2.3 Model Checking by Systematic Execution

Two model checkers that directly analyze implementations written in C and C++ are Verisoft [God97] and CMC [MPC⁺02]. Verisoft, from which MaceMC takes the idea of bounded iterative depth-first search, views the entire system as several *processes* communicating through message queues, semaphores, and shared variables *visible* to Verisoft. It schedules these processes and traps calls that access shared resources. By choosing the process to execute at each such trap point, the scheduler can exhaustively explore all possible interleavings of the processes' executions. In addition, it performs stateless search and partial order reduction, allowing it to find critical errors in a variety of complex programs. Unfortunately, when we used Verisoft to model-check Mace services, it was unable to exploit the atomicity of Mace's transitions, and this combined with the stateless search meant that it was unable to exhaustively search to the depths required to find the bugs MaceMC found.

A more recent approach, CMC [MPC⁺02], also directly executes the code and explores different executions by interposing at the scheduler level. However, to avoid re-exploring states, CMC uses state hashing instead of partial-order reductions. CMC has found errors in implementations of network protocols [ME04] and file systems [YTEM04].

JAVAPATHFINDER [HP00] takes an approach similar to CMC for JAVA programs. Unlike VERISOFT, CMC, and JAVAPATHFINDER, MaceMC addresses the challenges of finding liveness violations in systems code and simplifies the task of isolating the cause of a violation. Additionally, VERISOFT, CMC, and JAVAPATHFINDER uniformly ignore time, and hence cannot find errors related to performance anomalies, as Mace performance tools can.

2.2.4 Model Checking by Abstraction

A different approach to model checking software implementations is to first *abstract* them to obtain a finite-state model of the program, which is then explored

exhaustively [Hol00, CDH⁺00, CW02, GS97, BR02, CPR06] or up to a bounded depth using a SAT-solver [CKL04, XA05]. This model can be obtained either by restricting the *types* of variables to finite values, as is done in FEAVER [Hol00], BANDERA [CDH⁺00], MOPS [CW02] or by using *predicate abstraction* [GS97] as is done in SLAM [BR00] and BLAST [HJMS02]. A related technique is *bounded model checking* implemented in CBMC, SATURN where all executions up to a fixed depth are encoded as a propositional formula, which is then solved using fast modern SAT solvers to find bugs. Of the above, only FEAVER and BANDERA can be used for liveness-checking of concurrent programs, and they require a user to manually specify how to abstract the program into a finite-state model.

2.2.5 Dataflow Based Static Analyses

Dataflow based static analyses are typically highly scalable. Examples include SPLINT [EL02], CQUAL [USW01, JW04], MC [ECCH00], and ESP [DLS02], which have found many bugs such as null-pointers, data races and buffer overflows in a variety of large applications. However, these methods are best at finding a particular class of errors. Furthermore, the relatively high rate of false positives prohibits their use for finding the subtle protocol errors targeted by our work.

2.2.6 Isolating Causes from Violations

Naik et al. [BNR03] and Groce [GV03] propose ways to isolate the cause of a safety violation by computing the difference between a violating run and the closest non-violating one. MaceMC instead uses a combination of random walks and binary search to isolate the cause of a liveness violation, and then uses a live path with a common prefix to help the programmer understand the root cause of the bug.

2.2.7 Model Checking for Real-Time and Hybrid Systems

There is a significant body of research on extending temporal logic and automata to account for the passage of time [ACD90, AH92], and for hybrid systems [ACH⁺95]. Moreover, there are model checkers like UPPAAL [BLL⁺96] and

HYTECH [HHWT97] for checking manually constructed models of real-time and hybrid systems. These systems find errors by locating executions that violate timed temporal properties. For example, they might locate executions that violate a specification of the form “event B must happen no later than 10 seconds after event A”. In contrast, our system runs on implementations and finds the “common case” performance and uses outlier executions to find performance anomalies.

2.2.8 Performance Error Detection

PIP is a concurrent, complementary technique for finding bugs in distributed systems [RKW⁺06]. PIP is an annotation language and an expectation checker that can be applied to executions. It provides a way to visualize distributed path-flow in a system and to write expectations to validate system paths. By writing a set of execution validators, the idea is that one can find performance bugs by looking at any non-validated paths. Our model checker is simpler to use and run because it does not require deployment of the system, it can automatically test a wide variety of executions, and it does not require careful examination of every possible distributed path-flow.

X-TRACE [FPK⁺07], an effort similar to PIP in many ways, focuses on the tracing of messaging between applications through extensions to the existing protocol stack. It allows tracing across protocol layers and networks, and allows developers to better understand the performance of their system. Like PIP, X-TRACE is focused on individual debugging of live executions, whereas our model checker is focused on automatic location and detection of anomalous executions. Finally, TREND-PROF [GAW07] allows users to measure the empirical computational complexity of implementations, by plotting the performance of the system across a range of input sizes. Divergences in expected behavior can pinpoint bottlenecks in the code e.g. functions whose run-times should grow linearly with input size, but instead grow at a faster rate. It is not clear if such techniques can be adapted to the dynamic and uncertain environment of distributed systems.

2.3 Summary

Despite previous and concurrent work in distributed systems toolkits, tools for finding bugs, and performance supporting tools, building distributed systems remains challenging. In light of this work, Mace seeks to bring together the best of these efforts with its own contributions to provide a complete programming environment for building distributed systems.

Chapter 3

Mace Design

Currently, there are three ways of specifying distributed systems. First, formalisms such as I/O Automata [Lyn96] or the Pi-Calculus [Mil89] can be used to model distributed algorithms as collections of finite-state automata (or processes), one for each node of the system, that interact by sending and receiving messages. Though these formalisms succinctly capture the essence of many distributed protocols and algorithms, they abstract away and ignore the low-level implementation details essential to the deployment of robust, high-performance systems. Second, higher-level programming languages such as Java, Python, and Ruby have eased some of the tedium associated with building distributed systems. However, they often introduce performance overheads and do not significantly simplify the task of ensuring system correctness or identifying inevitable performance problems.

Thus, developers seeking efficiency resort to the third option of assembling applications in an ad-hoc, bottom-up manner. While the resulting systems may be fast and reliable, they sacrifice structure, readability, and extensibility. The lack of structure in particular significantly limits the ability to apply automated tools, such as model checkers, to find subtle performance and correctness problems.

This chapter demonstrates how programming language, compiler, and runtime support can combine the elegance of high-level specifications with the performance and fault-tolerance of low-level implementations. We seek to drastically lower the barrier to developing, maintaining, and extending robust, high-performance distributed

applications that are readable and amenable to automatic analysis for performance and correctness problems.

The main difficulty in building these systems arises from the distributed, concurrent, asynchronous, and failure-prone environment in which distributed systems run. System complexity requires that applications be *layered* on top of fast routing protocols, which are built on top of efficient messaging layers. *Concurrency* and asynchrony imply that events simultaneously take place at multiple nodes in unpredictable orders. Messages may be delivered in arbitrary orders, dropped completely, or delayed nearly indefinitely. Nodes may at any time have multiple outstanding messages in-flight to other nodes and multiple pending received messages ready to be processed. Further, an arbitrary subset of nodes or links may *fail* at any time, leaving the system as a whole in a temporarily inconsistent state.

These properties make maintaining performance and correctness difficult. A single high-level system request may require communication with many nodes spread across the Internet. Often, even seemingly correct distributed system implementations perform an order of magnitude more slowly than expected. *Analyzing* executions to find the source of such problems frequently reduces to searching for a needle in a haystack: among (at least) millions of individual message transmissions, algorithmic decisions, and the large number of participating nodes, which network link, computer, or low-level algorithm resulted in performance degradation?

While each of these problems has well-known solutions, the task of addressing these issues simultaneously proves to be quite challenging because of their subtle interactions. For example, object-oriented design is the canonical way to build systems from sub-systems, but for distributed systems, hiding internal state from other layers results in serious performance penalties and duplicate effort. Similarly, there are standard ways to detect and handle failures, but the code for doing so must be interspersed (usually repeatedly at multiple points) with the code for handling common case operation, not only obfuscating the code but also eliminating the high-level structure required to use techniques such as model checking [God97] and performance debugging [AMW⁺03, BIMN03, CKF⁺02].

In this chapter, we present Mace, a new C++ language extension and source-to-source compiler for building distributed systems in C++. Mace seamlessly com-

bines *objects*, *events*, and *aspects* to simultaneously address the problems of layering, concurrency, failures, and analysis. While these are well known programming language ideas, the key advances of Mace are twofold. First, we unify, in one development environment, the diverse elements required to build robust and high-performance distributed systems. Second, and more importantly, by defining a *language extension* to write distributed systems, we are able to *restrict* the ways in which such systems can be built. For our domain, this restriction is both *expressive* enough to permit the compilation of readable high-level descriptions into implementations matching the performance of hand-coded implementations and *structured* enough to enable the use of automatic, efficient, and comprehensive static and dynamic analysis to locate and understand behavioral anomalies in deployed systems.

By using a structured—yet expressive—approach tailored to distributed systems, Mace provides many concrete benefits:

- Mace allows the programmer to focus on describing each layer of the distributed system as a reactive state transition system, using events and transitions as the basis for system specification. This explicitly maintains the structure given by high-level formalisms while enabling high-performance implementations.
- Mace uses the semantic information embedded in the system specification to automatically generate much of the code needed for failure detection and handling, significantly improving readability and reducing the complexity of maintaining internal application consistency.
- Mace supports automatic profiling of each individual causal path—the sequence of computation and communication among nodes in a distributed system corresponding to some higher level operation, *e.g.*, a lookup in a distributed hash table. Mace exports a simple language that allows developers to match their expectations of both system structure and performance against actual system behavior, thereby isolating performance anomalies.
- Mace’s state transition model enables practical model checking of distributed systems implementations to find both safety and liveness bugs. The Mace model checker, MaceMC, successfully finds subtle bugs in a variety of complex dis-

tributed systems implementations. Most of the bugs we found with MaceMC were quite insidious, present in mature code, and could not be found without exploiting the structure preserved by Mace.

3.1 Overview

Drawing from our experience building a variety of distributed systems based on high-level specifications, we categorize the gap between specification and low-level features essential to real deployments into the following categories:

- **Layers:** To manage complexity, network services consist of a hierarchy of layers, where higher level layers are built upon lower levels. The canonical example is the Internet protocol stack where, for example, the physical layer is responsible for modulating bits on a medium; the link layer delivers packets from one node to another on the same physical network; the network layer delivers packets between physical networks; and the transport layer provides higher level guarantees such as reliable, in-order delivery to application end-points. Each layer builds upon well-defined functionality of the layer below it and can typically work on a variety of implementations of the underlying layer's interface.
- **Concurrency:** A distributed implementation must properly contend with and exploit concurrency to maximize performance. For example, an overlay routing application must simultaneously contend with the application layer sending requests to the routing layer, the networking layer receiving new messages that must be passed up to the routing layer, and timers executing scheduled tasks. While these events may be interleaved on a single node, they can also be arbitrarily interleaved across nodes.
- **Failures:** A robust implementation must account for the inevitable failures of different components or nodes of the distributed system. Failures are often difficult to detect; for instance it is impossible to distinguish between a failed node and one that is particularly slow. Further, the remaining nodes must correctly update their state to reflect the new configuration, lest inconsistencies lead to further errors.

- **Analysis:** Given the complex operating environment, there are always performance bottlenecks and correctness issues that arise because the developers overlooked some subtle scenario or miscalculated some parameter like a message timeout. An implementation must be structured and readable enough to permit the manual and automatic analyses required to fix such performance and correctness problems.

While well-known techniques address each of these issues in isolation, the primary challenge in our setting is to devise mechanisms that help the programmer resolve the tensions arising from complex interactions between the four problems. For example, the standard solution to the problem of layering is Object-Oriented design. However, for high-performance applications, treating layers as black boxes that hide their internal state and mask failures leads to performance bottlenecks. For instance, higher level routing algorithms greatly benefit from lower level information about link latencies and knowledge about which nodes or links have failed. Further, multiple sources of concurrency complicate the task of propagating information consistently between layers.

Similarly, failures make it difficult to design a layering mechanism. The approach of masking low-level failures—while appealing because it simplifies higher layers—is insufficient in distributed environments because it sacrifices significant performance gains available from notifying the upper layers of the failure. For instance, the transport layer could mask failures by buffering sent messages and attempting to resend them until it succeeds; however, doing so would prevent higher layers from adjusting their own state to achieve better performance. An example of this is a multicast layer reconfiguring its tree structure for higher throughput after a failure. Unfortunately, the task of notifying the upper layers is complicated by the fact that failures can happen concurrently with other system events. Further, concurrency makes it tricky to cleanly separate the failure detection and handling code from the rest of the common-case code, obfuscating the resulting system and destroying structure.

Standard techniques such as profiling to find performance bottlenecks and model checking to find pernicious bugs typically cannot be applied to distributed systems implementations. In ad-hoc implementations, the code that handles concur-

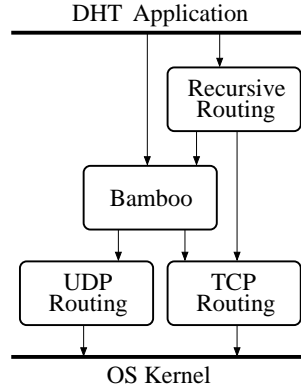


Figure 3.1: Bamboo DHT design architecture.

rency and failures obscures code structure making manual and automated reasoning difficult (but essential). The principle technique used by developers to analyze deployed systems is tedious ad-hoc logging that clutters the code and often delivers only limited value because the programmer must manually stitch together spatially and temporally scattered logs.

Finally, concurrency makes it difficult to reason about or to even replicate behaviors (due to non-deterministic factors like network latencies and scheduling decisions), thereby severely increasing the time and effort required to find complex bugs via testing. In our experience, particularly subtle bugs may remain latent for weeks in a deployed system. Further, because these bugs often result from inconsistencies between the state at multiple nodes, the subsequent departure or failure of a node after the bug manifests itself can push the system back into a consistent state, masking the bug and making it even more difficult to track.

Thus, to develop high-performance systems from high-level specifications, we must devise techniques to architect the system, to determine when failures have occurred, and to propagate and exploit information throughout the architecture. These techniques must operate in a concurrent setting, enable modularity and reuseability, and explicate the high-level structure of the algorithm, thereby enabling manual and automatic system level analyses.

Mace Design Principles

To address the challenges posed by the domain of distributed applications, we base Mace on three fundamental concepts.

- **Objects:** Mace structures systems as a hierarchy of *service objects* connected via explicit interfaces. We use an object to implement each layer of the system running on an individual node. The interface for each layer specifies both the functionality *provided* by that layer as well as any *requirements* that must be satisfied to use that layer.
- **Events:** Mace uses *events* as a unified concurrency model for all levels of the system: within an individual layer, across the layers at a single node, and across the nodes comprising the entire system. Each event corresponds to a method implemented by a service object.
- **Aspects:** Mace provides *aspects* to describe computations that cut across the object and event boundaries; in particular, aspects define tasks that need to be performed when particular conditions become satisfied.

While each of these ideas has been studied extensively in isolation, we demonstrate that they combine synergistically to preserve the high-level structure of the distributed system and simultaneously address the complexity and challenges of building robust, high performance implementations. We use a popular Distributed Hash Table (DHT) to illustrate the challenges associated with building distributed systems and our approach to addressing these challenges. DHTs support `put` and `get` operations on a logical hash table whose actual storage is spread across multiple physical machines, and thus form a convenient abstraction for building higher-level applications like distributed file systems [DKK⁺01, MMGC02]. The key properties of a DHT implementation are scalability and robustness to failure. We consider a DHT built on the Bamboo routing protocol [RGRK04a] (similar to Chord [SMK⁺01] or Pastry [RD01]).

Layers

Nodes in Bamboo self-organize into a structure that enables rapid routing of messages using node identifiers. This protocol forms a single layer of the DHT as shown in Figure 3.1. Bamboo is built on top of a TCP subsystem that maintains network connections and delivers messages and a UDP subsystem that sends latency probes. A recursive routing subsystem routes messages to the node owning a given key by asking Bamboo for the next hop towards the destination. The DHT application layer uses the lower layers to store and retrieve data.

Mace enables programmers to build layered systems by using objects to implement individual layers and events to facilitate interaction across layers. For each layer, the programmer writes interfaces specifying the events that may be received from or sent to the layers both above and below. A layer’s implementation consists of a *service object* that must be able to receive and may send all the events specified in the interfaces. *Thus, Mace combines objects and events to enable programmers to build complex systems out of layered subsystems, thereby abstracting functionality into layers with specified interfaces and allowing the safe reuse of different implementations (meeting the same interfaces) of a particular layer in different systems.*

Concurrency

In Bamboo, a key challenge is to provide fast message routing while simultaneously dealing with node churn – i.e. the arrival and departure of nodes from the system. To achieve this goal, the system must concurrently process network errors, messages from newly created nodes, and it must periodically perform maintenance to ensure routing consistency.

In Mace, each service object consists of a *state-transition* system beginning in some initial state. Each node progresses by sequentially processing *external* events originating from the application, the physical network layer, or self-scheduled timers. Upon receiving an event, the service object executes a corresponding *transition* to update its state, during which it may transitively send new events to the layers above and below, each of which are processed synchronously without blocking until completion. Furthermore, a transition may queue new external events locally by

scheduling timers and remotely by sending network messages. Once processing for a given external event completes, the node picks the next queued external event and the process repeats.

The Mace event-driven model provides a unified treatment of the diverse kinds of concurrency that must be handled in an efficient implementation: the reception of messages from other nodes (via the transport layer), the reception of high-level application requests, the firing of timers, and cross-layer communication all correspond to events that the relevant layers must handle via appropriate transitions. Additionally, Mace ensures that the transitions execute without preemption, freeing the programmer from worrying about exponential interleavings of concurrent executions. Finally, because Mace automatically dispatches events through a carefully tuned scheduler, Mace systems can achieve the throughput necessary for high performance applications with minimal programmer involvement. *Thus, objects, events, and aspects enable Mace to describe each layer of a complex application with the simplicity and conciseness of high-level models. When combined with the modular layering mechanism, Mace provides a succinct representation of the entire computation stack for each node of the distributed system.*

Failures

Bamboo builds an overlay network which forms a logical ring among the nodes. To create and maintain this topology, each node keeps references to its adjacent peers in the ring. If one node fails, the application-level state which correspond to the relationships between that node and its neighbors may become inconsistent, breaking the overlay structure.

Mace uses aspects to cleanly specify how to consistently update local state in response to a variety of cross-layer events, such as: node arrivals, departures, and application-level failures. The developer can specify predicates over the variables of a given node that test for programmer-specified inconsistencies. Mace generates code to evaluate the predicate whenever the relevant variables change and to execute the aspect when the predicate is satisfied. Aspects provide an ideal mechanism for specifying and detecting failures and inconsistencies, because without them, the developer would have to undertake the tedious and error-prone task of manually placing check-

ing code throughout the system, additionally reducing readability. When a failure occurs, the Mace runtime sends notification events to the appropriate layers. Upon receiving these events, the system executes recovery transitions. *Thus, Mace combines objects, events, and aspects to provide clean mechanisms for specifying, notifying, and handling various types of failures and for maintaining the consistent internal state necessary for fault-tolerant implementations.*

Analysis

Bamboo routes messages through several intermediary nodes, so tracing the forwarding path for a specific message manually, for instance to debug the timing of a request, involves inspecting multiple physically scattered log files. Mace simplifies such analysis tasks through preservation of the explicit high-level structure of the distributed application with three techniques. First, Mace uses aspects to separate code needed to log statistics, progress, or debugging information from the actual event handling implementation. By removing the distracting logging statements, Mace keeps the system code readable. Second, Mace exploits the structuring of the computation into causally related event chains to generate event logs, which may be spatially and temporally scattered. These event logs can be *automatically* aggregated into flows describing high-level tasks, extracting the events at individual nodes corresponding to some higher-level operation. The structure preserved in the flows allows developers to use automated analysis techniques to find and fix performance anomalies.

Third, the modular structure of Mace applications enables developers to test the system using simulated network layers that facilitate deterministic replay, as described in § 5. MaceMC combines these deterministic layers with a special scheduler that iterates over all possible event orderings. MaceMC systematically explores the space of possible executions to find subtle bugs in the system. The event-driven nature of Mace applications reduces the number of interleavings that must be analyzed, enabling MaceMC to search deep into the execution space. *Thus, objects, events, and aspects combine to structure Mace implementations that enable automated analysis techniques to improve the performance and reliability of the distributed application.*

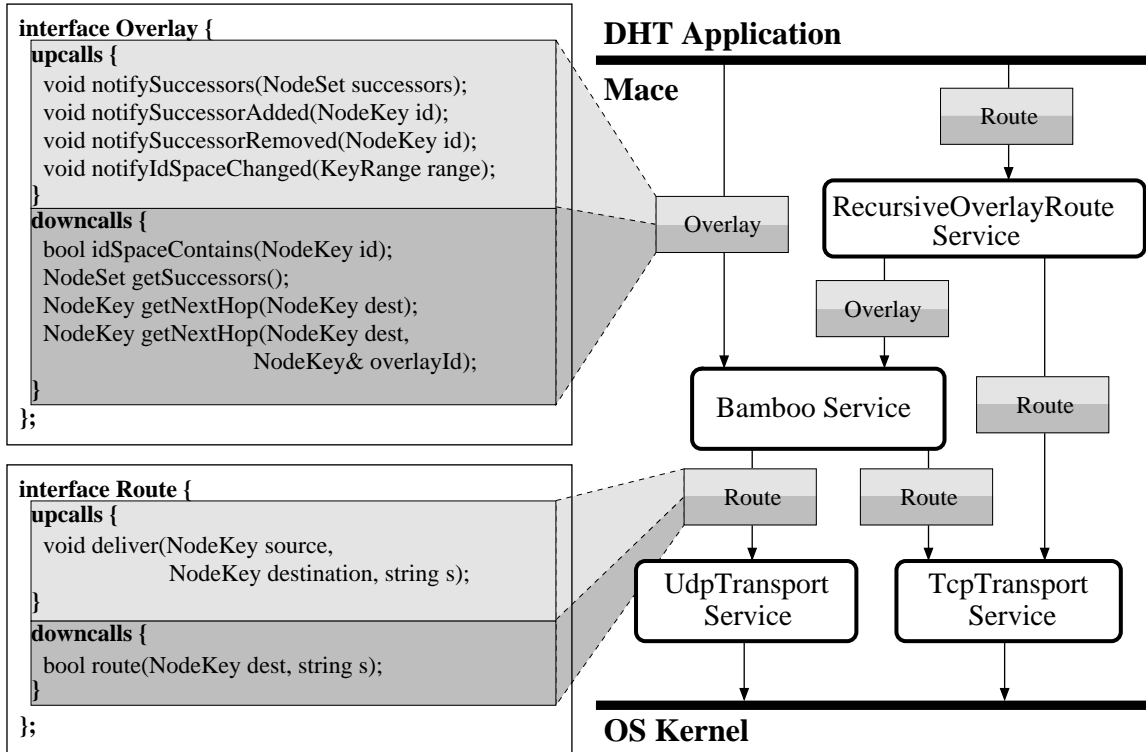


Figure 3.2: Mace service composition for a DHT application using Recursive Overlay Routing implemented with Bamboo. Shaded boxes (dark for downcall, light for upcall) indicate the interfaces implemented by the service objects.

3.2 Mace

We now describe the details of how Mace combines objects, events, and aspects to generate high performance, fault-tolerant implementations from high-level specifications. Our C++ language extensions structure each service object as a state machine template with blocks for specifying the interface, lower layers, messages, state variables, inconsistency detection, and transitions containing C++ code implementations of event handlers. The template syntax allows the Mace compiler to enforce the architectural design by performing a high-level validation of the service object. Additionally, the structure gives the Mace compiler the necessary information to automatically generate efficient glue code for a variety of tasks that in previous, ad-hoc implementations, had to be manually inserted by the developer.

3.2.1 Layers

To specify a distributed system in Mace, the programmer simply specifies the set of layered *service objects* (abbreviated to services) that comprise a single node and the implementation of the required interface for each service object. Figure 3.2 depicts a more detailed view of the Bamboo architecture (shown earlier in Figure 3.1), including the interfaces.

Interfaces. An *interface* comprises a set of *downcall* events and a set of *upcall* events. Upper layers send downcalls received by lower layers. Lower layers send upcalls received by upper layers. We model events using methods – sending corresponds to calling the appropriate method, and receiving corresponds to executing the method. In Figure 3.2 on the left, we show two interfaces: **Overlay** and **Route**. For each interface, the top half (lightly shaded box) corresponds to the upcall events, and the lower half (darkly shaded box) shows the downcall events. (Note that the syntax shown is for illustrative purposes only. The actual syntax is described in § 4.1.1.)

Architecture. Developers layer service objects implementing higher layers on top of service objects implementing lower layers. To facilitate modular design and seamless replacement of one service object with another, we specify for each service object the set of lower-level interfaces it *uses* and the upper-level interface it *provides*.

- **Used Interfaces:** When specifying a service, the developer declares each lower-level service with a name and an interface. The service may send any of the downcall events specified in the interface to any of the lower layers, and it must implement all the upcall events to receive any callbacks.

Bamboo uses two lower-level services of type **Route**, which it binds to local names **TCP** and **UDP**. The Bamboo implementation can call `downcall_route` for the **TCP** or **UDP** service object implementing the lower level, as `route` is a downcall event in the used interface. Similarly, the lower-level **TCP** and **UDP** services can invoke the `deliver` callback on Bamboo, as it is an upcall in the **Route** interface.

- **Provided Interface:** When writing a service, the developer specifies how upper layers can use the service via a *provides* interface. The service must imple-

ment all downcall events specified in the provides interface and may also send callback events to upper layers, typically in response to some prior request.

Figure 3.2 shows that all arrows pointing to Bamboo have type `Overlay`, indicating that Bamboo provides the `Overlay` interface to upper-level services. Thus, the Bamboo service object must be able to receive the `getNextHop` event from the upper layers, as it is a downcall event in the `Overlay` interface. Likewise, the Bamboo service may send an upcall `notifyIdSpaceChanged` event, which must be implemented by any upper layers using Bamboo.

Static Checking. The Mace compiler performs two compile time checks to enforce that each service object meets the requirements of the interfaces that it uses and provides. First, the compiler checks that the object implements methods corresponding to all the upcall events in the used interfaces and the downcall events in the provided interface. Second, the compiler checks that the object only calls methods corresponding to downcall events in the used interfaces, and the upcall events in the provided interface.

The service specification explicitly names lower-level services because this knowledge is required to build the service. For example, Bamboo requires two transports: one for sending protocol related messages (TCP by default) and one for probing (UDP by default). However, any upper layers that use a given service are unknown when specifying the service, hence those need not and cannot be explicitly named. We observe that the upcall events sent to upper level services are in response to previous requests made by those services. Thus, the Mace compiler automatically generates code such that every downcall is accompanied by a reference to the source of the downcall, and the service employs this reference to determine the destination of the subsequent upcall.

By explicitly decomposing the whole system using layers and interfaces, Mace allows implementations of subsystems to be easily reused across different systems, as any service implementation that meets the statically checked interface specifications can be used as the subsystem. For example, our DHT application works equally well by replacing the Bamboo service object with service objects implementing the Chord or Pastry algorithms, which also provide the `Overlay` interface.

3.2.2 Concurrency

The standard way of modeling distributed algorithms at a high-level is with state-transition systems. Mace enables developers to reap the many benefits of this structured approach by requiring them to specify each service object as a state transition system where the transitions represent the execution of the methods corresponding to received events. Given specifications for individual service state machines, Mace can automatically compose layers to obtain an efficient, structured system implementation. A state machine specification comprises two basic entities: states and transitions.

States. States are a combination of the finite high-level control states of the service protocol, along with the (possibly infinite) data states corresponding to values taken by variables such as routing tables, peer sets, and timers. Figure 3.3 shows how the programmer specifies the high-level states and state variables of the Bamboo service. The finite high-level states, `init`, `preJoining`, `Joining`, and `Joined` correspond to the four stages of joining the system. The state variables `myhash`, `myleafset`, `mytable`, and `range` correspond to the node’s unique identifier, the set of peers and routing table maintained by the node, and the space of keys assigned to the node. In addition, Bamboo uses two timers, one of which is automatically rescheduled at `MAINTENANCE_TIMEOUT` intervals by Mace compiler generated code.

Transitions. There are four kinds of transitions and corresponding events: upcalls received from lower layers, downcalls received from upper layers, scheduler events received from self-scheduled timers, and aspect transitions which occur immediately following the other event types. Methods implement the transitions and update the state upon receipt of the corresponding event. Figure 3.3 shows different kinds of transitions corresponding to events the Bamboo service object may receive. The full syntax is described in § 4.2.3, but briefly, a keyword labels each method and indicates its transition type. Each transition method can be guarded by a predicate over the state variables. This condition may reference the current high-level service state, service state variables, or event parameters. The transition only fires if the guard is true.

```

states { init; preJoining; joining; joined; }
state_variables {
  NodeKey myhash;
  leafset myleafset;
  KeyRange range;
  Table mytable;
  timer global_maintenance __attribute__((recur(MAINTENANCE_TIMEOUT)));
  timer join_timer;
}
transitions {
  /* Other transitions . . . */
  guard(state == joined) {
    scheduler global_maintenance() {
      NodeKey d = myhash;
      for(int i = randint(ROWS); i < ROWS; i++) {
        d.setNthDigit(randint(COLS), B);
      }
      NodeKey n = make_routing_decision(d);
      downcall_route(n, GlobalSample(d), TCP);
    }
    upcall forward(const NodeKey& src, const NodeKey& dest,
      NodeKey& nextHop, const GlobalSample& msg) {
      nextHop = make_routing_decision(msg.key); return true;
    }
    upcall deliver(const NodeKey& src, const NodeKey& dest,
      const GlobalSample& msg) {
      downcall_route(src, GlobalSampleReply(msg.key, myhash), TCP);
    }
    upcall deliver(const NodeKey& src, const NodeKey& dest,
      const GlobalSampleReply& msg) {
      update_state(src, msg.destHash, true/*known live*/, true/*do probe*/);
    }
  }
}

```

Figure 3.3: States and Transitions for Bamboo Service Object

To understand how the programmer structures the code for each service into events and transitions, consider the high-level state machine for the Bamboo object illustrated in Figure 3.4. The figure shows what happens when Bamboo receives events such as application join requests, network messages, or timers causing reattempted joins. The system begins in the `init` state, and transitions to the `preJoining` state

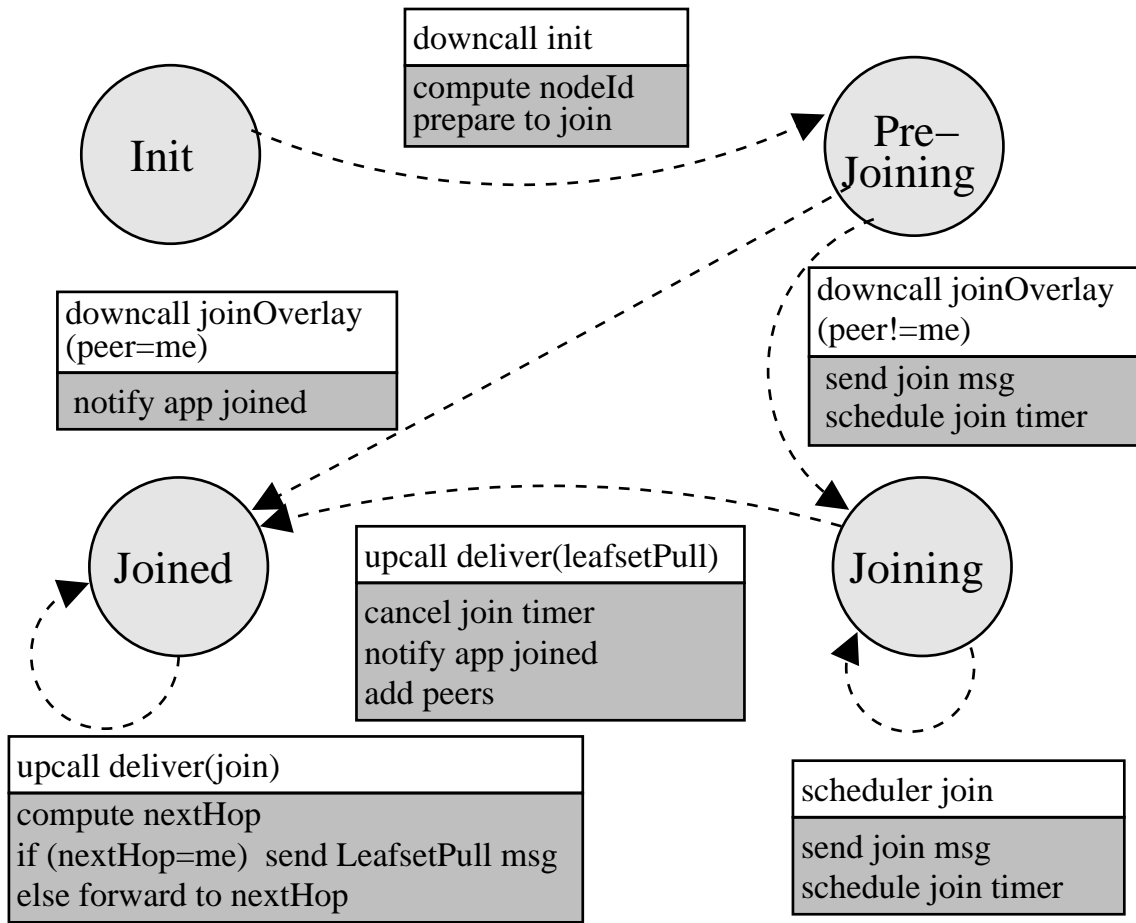


Figure 3.4: State machine model for Bamboo. States are shown in light gray; event-transitions are represented by arrows with names in white and actions in dark gray.

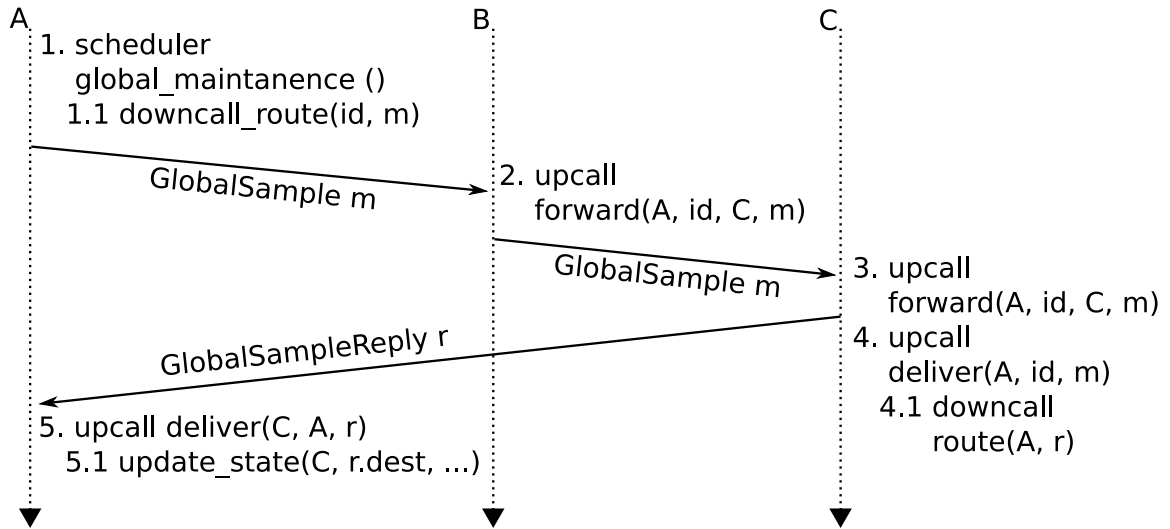


Figure 3.5: Message diagram for global sampling. In response to a scheduled timer, Node *A* routes a *GlobalSample* message to an identifier *id* by sending it to node *B*, which is forwarded to its owner, node *C*. Node *C* responds with a *GlobalSampleReply* message informing node *A* about node *C*, potentially causing a routing table update.

```

aspect<range> rangeChange(const KeyRange& pre_range) {
    upcallAll(notifyNewRange,range);
} // local detection of changes to the range variable
detect {
    leafset_monitor {
        nodes = myleafset;
        interval = 5 SEC;
        timeout = 5 * 60 SEC;
        interval_trigger(leafnode) {
            downcall_route(leafnode, LeafsetPush(myhash, myleafset));
        }
        timeout_trigger(leafnode) {
            leafFailed(leafnode);
        }
    }
    suppression_transitions {
        upcall deliver(src, dest, const LeafsetPull& m) {
            reset_leafset_monitor(src);
        }
    }
}
} // distributed detection (across myleafset)

```

Figure 3.6: Local and Distributed inconsistency and failure detection in Mace.

upon receiving a downcall event `init` from the application. When it subsequently receives the downcall event `joinOverlay`, it transitions to the `joining` state if it is not its own `peer` (captured via a predicate guarding the event), or to the `joined` state otherwise, in either case sending appropriate notification events and scheduling timers. In the `joining` state it periodically sends `join` messages to other nodes requesting to join the system, and finally, when it receives a `deliver(leafsetPull)` message from another node, it moves into the `joined` state. The rest of a node's life is spent in the `joined` state, where it periodically globally samples the other nodes to improve its local routing information.

Sequence diagrams are an informal technique programmers use to reason about low level interactions, such as those taking place while in the `joined` state. Figure 3.5 shows a sequence diagram depicting the interaction between nodes performing global sampling to improve routing tables. The periodically scheduled `global_maintenance` timer fires on node `A` causing it to select a random routing identifier `id`, and to then send a `GlobalSample` message to the node `B`, which is the next hop along the route. This message gets (transitively) forwarded by `B` until it reaches `C`, which actually owns the identifier `id`. `C` then sends a `GlobalSampleReply` message back to `A`, which, upon receiving the reply, may update its routing information.

Once the programmer has worked out the details of the protocol using the sequence diagram, it is straightforward to code in Mace using transitions and events. Figure 3.3 shows (in order) how the events corresponding to (i) the firing of the `global_maintenance` timer at `A`, (ii) the forwarding of the `GlobalSample` message to `B` and `C`, (iii) the final delivery message to the destination `C`, and (iv) the delivery of the `GlobalSampleReply` back to `A`, are implemented as transitions in the Bamboo service object. The bodies of the respective transitions implement the actions taken upon receiving the corresponding events shown in Figure 3.5.

3.2.3 Failures

Mace's use of service objects and events greatly simplifies the task of detecting, notifying, and handling failures and inconsistencies. While layering is essential for building complex services, the information hiding endemic to layered systems often

makes it difficult to deliver the best performance or the most agile fault handling. For example, when a DHT application’s socket breaks due to a node failure, the TCP transport layer could attempt to mask the error. However, doing so may prevent Bamboo from being able to route around the failed node, leading to degraded performance or incorrect message delivery. Rather, the TCP transport layer must propagate the error to Bamboo so that it can update its routing table and leafset. Bamboo, in turn, further propagates the error to the DHT application, so that it can redistribute the keys assigned to the failed node. Mace provides clean mechanisms for layering network services while also making it easy to deliver error notifications automatically from one layer to another when required for performance or fault tolerance.

Mace employs upcalls to signal higher layers of potential performance and correctness issues. It may be possible to correctly handle an issue entirely at a lower layer, but with suboptimal performance. If this is acceptable to upper layers, then they can simply ignore the corresponding upcall. However, for best performance it may be necessary to register handlers for such upcalls. While such cross-layer communication usually obfuscates code and eliminates many of the benefits of layering, we leverage our event-based structure to cleanly separate the *notification* and *recovery* code from the rest of the system that executes in the non-exceptional case.

Mace addresses the remaining challenge of providing programmers with a succinct but flexible mechanism for *detecting* both failures and inconsistencies through the use of *aspects*. Aspects provide a unified way to maintain consistent state, regardless of whether the state needs to be updated in response to an expected protocol event, such as a node arrival, or an unexpected event, such as a failure. Mace aspects check for two types of inconsistency/failure detection: those that involve purely local state and those that involve multiple nodes.

Local Failure Detection. Failures occurring in distributed systems can be characterized via inconsistencies in the values of state variables. A *local* failure occurs when the values of state variables at a single node are inconsistent. For example, in our DHT application built on top of Bamboo, the data that each node is responsible for depends on the key space specified by the **range** variable. In other words, the views of the range of the DHT layer and the Bamboo layer must be synchronized, and if they

are not, recovery action must be taken so that the DHT relocates the data according to the new range, potentially involving communication with remote system nodes.

Such failures can be specified using a predicate that characterizes the inconsistent values, i.e., which becomes true when the values of the variables are inconsistent. Thus, such failures can be locally detected by monitoring the predicate, and firing an event when the predicate becomes true. In Mace, the programmer specifies how failures should be detected and how to react to the failure using an *aspect transition*. A failure occurs when this predicate is true, which fires an error event and notifies the upper layers of the inconsistency.

Consider the example in the top of Figure 3.6, showing a local detection aspect that specifies that an inconsistency failure occurs when the value of the variable `range` changes (the aspect only fires when the monitored variables change, so no additional guard function is needed). When the change occurs, Mace sends a `notifyNewRange` event to the upper DHT layer indicating that its portion of the key space has changed and prompting the DHT to reorganize stored data appropriately. This aspect will correctly react to *all* events that change the range, whether it be the arrival of a new peer adhering to the Bamboo protocol or an unexpected peer failure.

The local detection aspect checks the predicate only at transition boundaries, avoiding notification of state that may become temporarily inconsistent in the middle of a transition. Thus, aspects and events provide a clean way to separate the failure detection, notification, and handling from the rest of the “common-case” code. We implement detection by keeping a shadow copy of monitored state variables, checking and updating them after each transition. In ad-hoc implementations, built without language support, the programmer would have to manually insert the check and notification each time the variables might be modified. In addition to greatly reducing readability, this task is error-prone, especially as the code evolves or is maintained by multiple programmers.

Distributed Failure Detection. A *distributed* failure occurs when the values across two or more nodes are inconsistent. For example, in Bamboo, each node maintains the set of its immediate peers in the state variable `myleafset`. Each such peer, in turn, must include the node in its own set of known peers. A distributed failure occurs if some element of a node’s leafset does not include the node in its own set of

peers. As a node cannot directly access the other nodes' internal state, the only way to determine the presence of such a failure is to actively exchange information across nodes, checking that the received information is consistent, and if so, returning messages acknowledging consistency. If the originating node receives the acknowledgment before a timeout occurs, it confirms that no failure has occurred.

The programmer specifies how to detect and react to distributed failures in Mace using a *detection aspect*, defining elements for the aspect, as shown at the bottom of Figure 3.6. The `nodes` are the set of nodes monitored by the aspect. In this example, it is the nodes stored in the set `myleafset`. The `interval_trigger` method indicates how the probes are sent to the elements of `nodes`. Here, the node sends `LeafsetPush` messages with the current value of `myhash` and `myleafset` state variables to the elements of `myleafset`, once every 5 seconds. Finally, a `suppression_transitions` block indicates the response expected from the other nodes, together with a timeout before which the response must arrive. Here, it stipulates that the node must receive a `LeafsetPull` message from each of the other nodes in `myleafset` before a timeout period of 5 minutes elapses.

Mace generates extra state and the code needed to keep track of the last time it has heard from each monitored node. It sets a timer to fire sometime after it expects to receive an acknowledgment of a particular remote configuration. If the timeout occurs and the guard is true, then Mace calls the event specified in the `timeout_trigger`, notifying upper layers of the failure. As in the case of the local failures, the transition corresponding to the error event corresponds to the code that implements the recovery mechanism. Likewise, Mace simplifies the detection of distributed failures by separating the detection code into an aspect and automating the process of sending the probe messages and detecting timeouts.

3.2.4 Analysis

By using objects and events to preserve the high-level structure of the distributed system, Mace can automate a variety of post-development analyses that find performance or correctness problems.

Execution Logging and Debugging. Mace uses aspects to generate debugging and logging code without cluttering the service specification. Mace exploits the preserved structure to enable different levels of automatic logging. First, with event-level logging, the generated program logs the beginning and end of each high-level event. This process captures the order and timing of events at each node. With state-level logging, every time a transition finishes, the generated program logs the node’s complete state, which indicates the change caused by the transition. Finally, with message-level logging, the generated program additionally logs the content and transmission time for each message sent from or received by the node.

We have used the automatically generated logs to implement MDB, a replay debugger for Mace distributed applications. MDB collects all individual node log files centrally and allows the developer to single step, forward and backward, through the execution of individual nodes. The developer may move from node to node, inspecting global system state, in a manner similar to traditional single process debuggers.

Causal-Paths. Mace provides a more advanced form of logging that aggregates execution events distributed across multiple nodes into a set of causal paths. Each path starts at a given node, with a particular seed event, and contains the sequence of all events that are causally, transitively related to the seed event. For example, if the seed event is a request generated by a particular node, then the causal path includes the sequence of messages (and resulting events and transitions) that span the different nodes until the response returns to the requester.

To obtain such causal paths in a Mace application, the programmer specifies the seed event where the path begins and the event that ends the path. Mace tags all the relevant, causally related activity that occurs between the seed and the end (i.e., all events, transitions, messages sent and received) with a dynamically generated path identifier and generates logs such that events distributed across multiple nodes can be collected using their shared path identifier. As a result, Mace enables logging at a semantic-level and allows programmers to understand and analyze the behavior of the system at a high level. In addition, previous work [RKW⁺06] describes how the causal-path logging done by Mace can be *automatically* mined to find and fix performance anomalies, by comparing the causal paths resulting from actual executions with programmer-specified high-level *expectations*.

Table 3.1: Lines of code measured in semicolons for various systems implemented in Mace and other distributions.

System	Mace	Distribution
Bamboo	500	1800
BulletPrime	1000	2800
Chord	250	3400
Overcast	450	NA
Pastry	600	3600
Scribe	300	500
SplitStream	200	331
Vivaldi	100	250

While this earlier work on the benefits of causal paths to performance debugging is independent of Mace, it requires significant manual logging in standard, unstructured C++ applications. We have found that more than 90% of the logging required for causal path analysis can be automatically inserted by the Mace compiler, significantly lowering the barrier for leveraging the benefits of such performance debugging tools.

Model Checking. A high-level model of a distributed system enables exhaustive analyses like model checking to find subtle bugs in either the protocol or implementation of a distributed system. Mace allows developers to use the same analysis to find subtle errors in the actual implementation of the system by making it easy to systematically explore the space of executions of the implementation. The full details of the model checker, MaceMC, are found in § 5.

3.3 Experiences

In this section, we outline some of our experiences developing distributed applications with Mace. Mace itself is implemented as a source-to-source compiler in Perl using a recursive descent parsing module. The Mace compiler emits C++ code, which is then compiled using any C++ compiler such as g++. We have implemented over ten substantial distributed systems in Mace, many of which we have run across the Internet, in part on testbeds such as PlanetLab [PACR02]. In ad-

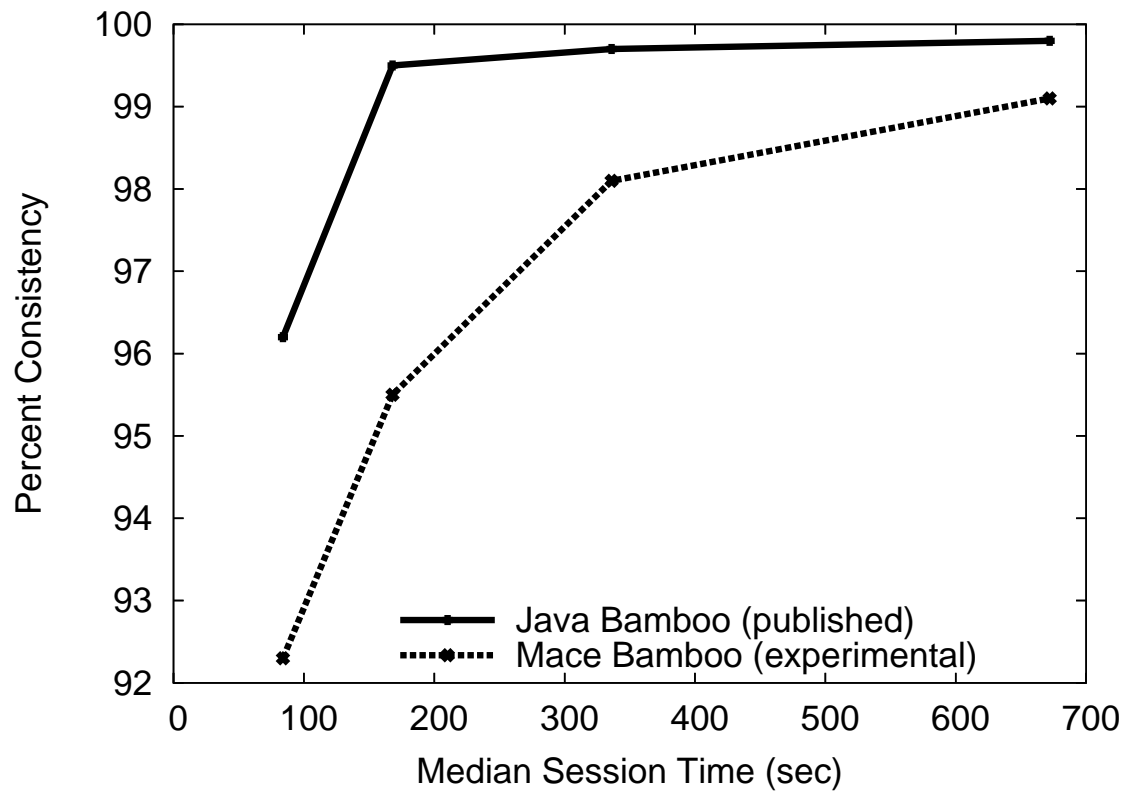


Figure 3.7: Percentage of lookups that return a consistent result.

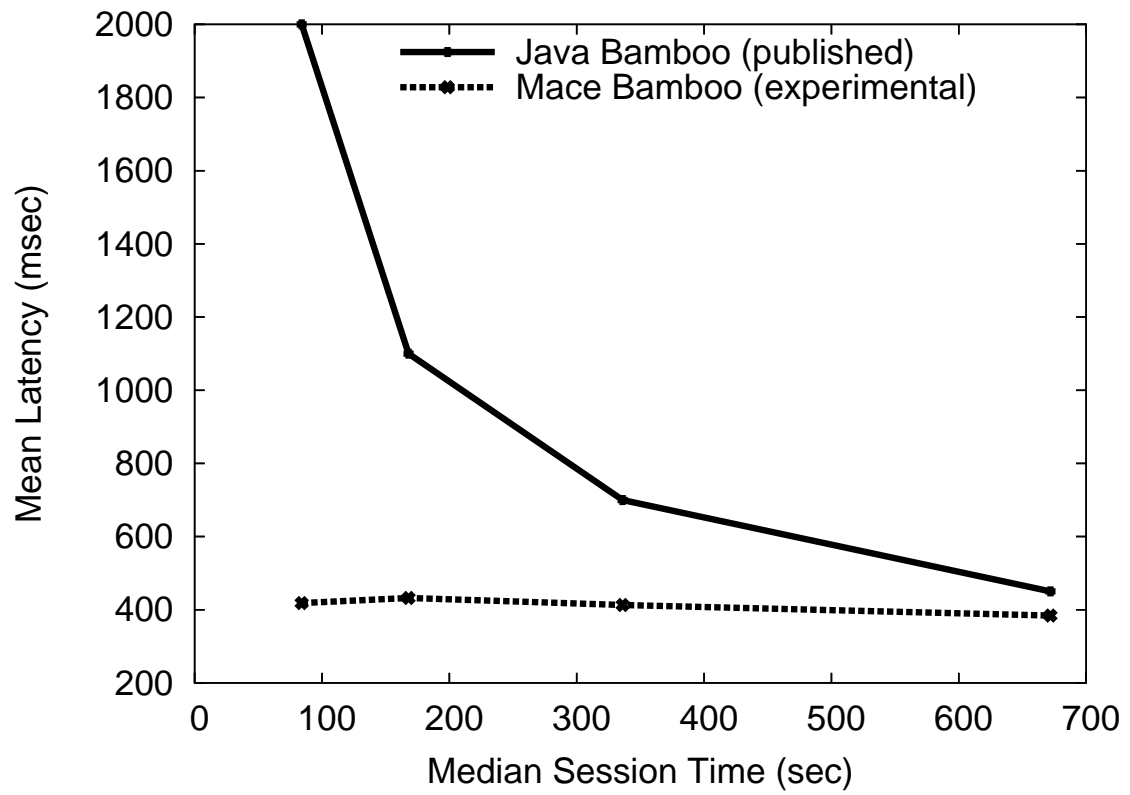


Figure 3.8: Mean latency for lookups that return consistent results.

dition to the Bamboo implementation discussed here, we have also implemented the systems shown in Figure 3.1. These systems include Chord [SMK⁺01], Pastry [RD01], Scribe [RKCD01], SplitStream [CDK⁺03] (from the FreePastry [fre06] distribution), BulletPrime [KBK⁺05] (from the MACEDON [RKB⁺04] distribution), Overcast [JGJ⁺00] (not available to us for line counting), and Vivaldi [DCKM04]. Excepting BulletPrime (which was written in the MACEDON language), each of these services was originally developed in unstructured C++ or Java.

The Mace compiler eliminates many tedious tasks that must otherwise be hand-implemented to achieve high performance, such as message serialization and event dispatch, and correspondingly drastically reduces the implementation size. A Mace service object implementation contains a block for specifying message types (essentially a `struct` with optional default values), for each of which the compiler generates a class containing optimized methods to serialize and deserialize the message to and from a byte string that can be sent across the network. The Mace compiler also generates methods to automatically perform event sequencing and dispatch. The generated code selects the next pending event, performs locking to prevent preemption, evaluates any guard tests for the transition, executes the appropriate method implementing the event handler (assuming the guards succeeded), tests any aspect predicates that might have been updated by the transition, and finally releases the acquired locks. Overall, we find that the structure imposed by Mace greatly simplifies the implementation by allowing the programmer to focus only on the essential elements, without compromising performance or reliability.

3.3.1 Performance Evaluation

To evaluate the performance of Mace systems, we compare our Bamboo implementation in Mace with its well-tested counterpart [RGRK04b]. To distinguish the two versions, for this section we will refer to our implementation as Mace-Bamboo. We chose Bamboo because of its excellent performance, detailed published performance evaluation, and its publicly available and well documented code base. Bamboo is a highly optimized Java implementation of a distributed hash table, based originally on Pastry [RD01]. We compare behavior of node lookups under churn. Lookups operate

by forwarding a message using increasing prefix matching to nodes whose identifiers are progressively closer to the key. Bamboo explores the limitations of previous protocols in providing consistent routing in the presence of node churn, and proposes several modifications to Pastry to allow it to deliver high consistency and low latency even when nodes are entering and leaving the system at a high rate.

Consistency is a measure that captures whether different nodes routing to the same identifier will reach the same destination. This is the most important requirement for correct performance of applications using a DHT, since they rely on being able to share data by using the same identifier to store and retrieve values. Our exercise of re-implementing Bamboo serves to show the simplicity of implementing distributed systems in Mace and our ability to generate robust, efficient, and high performance code. Two experienced Mace developers implemented the primary Bamboo algorithms in twelve hours (excluding the reliable UDP transport), starting from an existing Mace Pastry implementation.

To compare against published Bamboo experimental results (we attempted to reproduce the published results but could never achieve them, most likely due to having fewer machines), we prepare a framework that matches, to the best of our ability, the original experimental conditions. The experiment consists of 1000 Bamboo nodes organized into groups of 10 performing simultaneous lookups of random keys. A lookup result is considered consistent if a majority of the 10 nodes return the same result. Each group of 10 nodes performs lookups according to a Poisson process with an average inter-lookup delay of 1 second. For the runs, we vary the median churn rate also according to a Poisson process, ranging from on average 8 deaths per second to 1 death per second.

We run 1000 Bamboo instances on 16 physical machines (the published Bamboo results used 40 machines), using the ModelNet [VYW⁺02] network emulator with a single FreeBSD core. Each of the physical machines is a dual Xeon 2.8Mhz processor with 2GB of RAM. During the runs, load averages ranged from 0.5 to 1.5. The emulated topology consists of an INET network with 10,000 nodes, 9,000 of them routers. Client bandwidths on the topologies ranged from 2-8Mbps. To start the experiment, nodes were staggered, starting one on each machine each second for a minute. The churn and lookup schedules began as soon as all nodes were live. This experimental

setup differs from the published Bamboo experiments in that the stagger-start of our experiments is at a much faster rate, we do not wait for the network to settle after starting all nodes, we run with 1/3 the number of machines, and our request load is 10 times higher.

Figure 3.7 shows the consistency numbers for the Java-Bamboo and Mace-Bamboo. The published consistency values demonstrate near-perfect consistency at all churn levels. While still above 92% consistent, Mace-Bamboo nodes are slightly less consistent than their counterpart, though they track its performance closely. However, as shown in Figure 3.8, the Mace-Bamboo latency outperforms Java-Bamboo at each of these churn levels, and by a factor of 5 at high churn levels.

3.3.2 Undergraduate Course

To aid in the evaluation and development of Mace, we have used it in two undergraduate networking courses, and it has also been used in several graduate course projects. During the spring quarter of 2005 and the spring quarter of 2006, students in advanced undergraduate networking classes at UCSD were “asked” to program in Mace for a class project. None of the students enrolled in the class had been exposed to Mace previously. The project involved implementing a peer-to-peer file sharing program loosely based on the popular FastTrack protocol. The protocol includes a number of distributed concepts, such as: flood-based searching, distributed election, and random network walks. To prepare for the project, students were given a one-hour introduction to Mace, a list of protocol messages (to support inter-operation), and a skeleton template for a basic Mace service. 90% of the students successfully completed the project and a majority expressed a preference for programming in Mace relative to Java or C++.

3.4 Summary

In this chapter, we argued for the benefits of language support to construct robust, high-performance distributed systems. The principle challenge in this environment is resolving tensions between the tasks of developing a clean layering system,

handling concurrency and failures, and preserving enough structure to enable automated performance and correctness analyses. The key insight behind Mace is that objects, events, and aspects can be seamlessly combined to simultaneously address the intertwined challenges.

Mace’s language structure and restrictions enable a number of important features that are otherwise difficult or impossible to express with existing languages: language support for failure detection, causal path performance and correctness debugging, and model checking unmodified Mace code. We have employed Mace to build more than ten significant distributed applications, which have been successfully deployed over the Internet. Others are using Mace to support their own independent research and development. Using automated debugging tools that exploit the Mace structure to find and fix problems, exploiting the flexible architecture to reuse optimized subsystems across applications, and leveraging the uniform and efficient event-driven concurrency model, Mace system specifications are about a factor of five smaller than original versions in Java and C++, while delivering better performance and reliability.

3.5 Acknowledgement

Chapter 3 is an updated and revised copy of the paper by Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat, titled “Mace: Language Support for Building Distributed Systems,” as it appears in the proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation 2007. ©2007 ACM DOI <http://doi.acm.org/10.1145/1250734.1250755>. The dissertation author was the primary researcher and author of this paper.

Chapter 4

Lessons of the Mace Language

In the prior chapter, we described the design, implementation, and evaluation of the Mace programming language extension to C++. In this chapter, we describe in detail the grammar of Mace, and some lessons we learned in the process of designing Mace. Some code examples appear in Chapter 3, and some more short examples are given in this chapter. Full examples of services implemented in Mace can be found in its HOWTO documentation, and also from the public release of the code.

Implementing a system in Mace entails breaking it up into a layered, asynchronous, state-event-system. Each layer should be divided by a clean separation of concerns, providing a generic service interface to higher level services. The interface must be designed such that the operation requested can be completed without having to block. If an operation would have to block, it should be considered two operations, a request and a completion, which can be delayed to a future callback. The only way to access the state of any given layer is through an event-method from its interface, protecting its state from outside harm. This chapter will describe how this is accomplished in the Mace language.

Programming in mace is described in the following sections. Section 4.1 covers the architectural design grammar, which includes the layering of services. Section 4.2 presents the individual components of the Mace language grammar for services, while Section 4.3 details the set of options that can be supplied to the compiler in the grammar in various places. Finally, Section 4.4 gives a few notes about programming in Mace.

4.1 Architecture Design

The complete structure of a Mace specification is given in § 4.2. But the first part of this specification syntax describes the definition of the architectural design of the service—both the interface it provides, and the layering of services atop other services. This section first describes the syntax of the header files that we use to define the interfaces (§ 4.1.1), and then the syntax for this first part of the component architecture (§ 4.1.2).

4.1.1 Interfaces

The first task when implementing any system is to break it up into its interface and layer design. For a DHT application, this interface will generally be directly tied to the service it provides. In this case, the interface in question should have a method to *get* data based on a key, to *put* data based on a key, to check whether the DHT *contains* a given key (without retrieving the data), and to *remove* the given key. In Mace, we'll use the *MaceKey* type, which can support a variety of types of keys, to represent keys or addresses.

This can generally be written in a C++ header file as:

```
class DHTInterface {
public:
    virtual string get(const MaceKey& key);
    virtual bool containsKey(const MaceKey& key);
    virtual void put(const MaceKey& key, const string& data);
    virtual void remove(const MaceKey& key);
};
```

Unfortunately, this cannot be directly used in Mace, as some implementations of a DHT will likely not be able to respond to *get* or *containsKey* requests without checking with other nodes.

Lesson 1: In designing an interface, we must strike a balance between the perfect interface for the service implementation we have in mind, and a general implementation of that interface. When designing a new interface, we should ask

ourselves what requirements the interface design itself places on the implementation of the service. If an interface lends itself to only a single implementation, it is probably not the right interface.

In this case, the best thing to do is to separate the request from the result, allowing the result to be asynchronously “called back” or *upcalled*. A service that does provide immediate availability of the result can simply perform the *upcall* immediately.

Mace uses two types of interface files to describe service interfaces. An interface for a service (to be used by applications or higher-level services) is called a *service class*, while the upcall interface (or call back interface) of the service is defined by the set of *handlers* that are registered with the service. This relationship can be seen in Figure 3.2 (though it uses a different syntax than the one shown below). The downcalls make up the *service class*, while the upcalls make up the *handlers*.

A handler syntax file looks like this:

```
Handler: 'handler' Name '{' HandlerStatement* '}'
Name: (a token/word that gives the name of the Handler)
HandlerStatement: CppStatement | Function
CppStatement: (C++ statement for constants, enumerations, etc.)
Function: 'virtual' (?) FunctionDecl FunctionImpl
FunctionDecl : (C++ function declaration without terminator)
FunctionImpl : ';' | '{' (C++ function body) '}'
```

The handler filename should be *NameHandler.mh*, where *Name* is replaced by the name of the handler. It will generate a C++ class whose name is *NameHandler*. C++ statements that are parsed will be pasted into the generated C++ class, and can be used to define constants or enumerations. Each of the functions defined will also be included in part of the generated C++ class. A function can either be declared (ending with a semicolon), or defined (ending with a method body), and can be virtual, or non-virtual. As with C++, virtual methods can be overridden by implementing service handlers, and non-virtual methods cannot (the latter generally are useful in an interface only to transform parameters and call another virtual method). Methods cannot be defined as purely virtual—this is because handler classes must have default implementations for any method, for the null reference handler implementation. The

default implementation for a declared (but not defined) method will therefore be to abort execution at runtime.

Another transformation the Mace compiler makes is to add a `registration_uid_t` parameter to each method. The `registration_uid_t` parameter is used to identify a specific registration of a specific *handler* with a specific instance of a *service class*. In a handler method, it tells the called function which service instance is calling the method. All *service class* functions also have a `registration_uid_t` parameter added to them, and they tell the implemented service which registered *handler* should receive *upcalls* related to this function call.

The handler interface of the DHT service, therefore, is used to asynchronously deliver the result of calls from its primary interface. For example, the result of a call to the ‘get’ method will be returned by a call to the handler’s ‘dhtGetResult’ method, and the result of a call to the ‘containsKey’ method will be returned by a call to the handler’s ‘dhtContainsKeyResult’ method. The ‘put’ and ‘remove’ calls, in our design, provide no feedback when they eventually complete, and therefore do not have a callback.

We call the DHT handler object the ‘DHTData’ handler, and define it as such:

```
handler DHTData {
    virtual void dhtContainsKeyResult(const MaceKey& key, bool result)
    { }
    virtual void dhtGetResult(const MaceKey& key,
                             const mace::string& value,
                             bool found) { }
};
```

The service class interface therefore is largely the same as the desired interface, but does not immediately return the value. The service class interface is defined in *NameServiceClass.mh*, and its syntax is the following:

```
ServiceClass: ‘serviceclass’ Name ‘{’ ServiceClassStatement* ‘}’
Name : (a token/word that gives the name of the service class)
ServiceClassStatement : CppStatement | Function | Handlers | MaceBlock
CppStatement: (C++ statement for constants, enumerations, etc.)
Function: ‘virtual’(?) FunctionDecl FunctionOptions(?) FunctionImpl
FunctionDecl : (C++ function declaration without terminator)
FunctionOptions : ‘[’ (key) ‘=’ (value) ( ‘;’ (key) ‘=’ (value) )* ‘]’
FunctionImpl : ‘;’ | ‘{’ (C++ function body) ‘}’
```

```

Handlers : 'handlers' HandlerName (',' HandlerName)* ';'
MaceBlock : 'mace' WhenToInsert '{' MacFileBlocks '}'
WhenToInsert : 'provides' | 'services'
MacFileBlocks : (mac file syntax blocks)

```

This is much like the handler interface description, except that it allows two additional types of statements, and synchronous options on functions. The first additional type of statement is the *Handlers* statement. It tells which Handler interface files are upcall handlers associated with this service class. For each handler in the list, the generated service class C++ object will contain a *registerHandler* method, which allows a *Handler* object of the given type to be registered to receive upcalls in response to the downcalls made to the service class. In the DHT case, it will list 'DHTData'. For any DHT service implementation, a DHTDataHandler should first be registered for a particular registration uid, then calls made on the DHT service should pass in the same registration uid, telling the service which registered handler to return upcalls to. This way, the same service can be shared by multiple higher level services, using the registration uids to keep their data distinct.

The second additional type of statement is a MaceBlock element, which allows the interface designer to add blocks of code to services that either *provide* this interface, or use a lower level *service* that does. The syntax of these will be described later, but this block can for example be used to (1) indicate strings that should be generally deserialized into messages through 'method_remappings' (see *TransportServiceClass.mh*), or (2) add transitions to every implementation such as to support automatic bootstrapping on initialization (see *OverlayServiceClass.mh*).

The function options (**FunctionOptions**) within service classes specify synchronous compiler options that can be used to make it easier to use a service from code that does not wish to use it in an asynchronous manner. By defining these options, an extra class is generated in the *NameServiceClass.h* file that automatically blocks the calling thread until the appropriate upcall is made, resuming its operation and filling in the functional parameters. To take advantage of these options, the interface designer should specify 4 things:

syncname The name of the function that will provide synchronous behavior

callback The function of the handler that is the upcall response for this downcall.

id The parameter of the function that can be used to match the request and response (the function listed in **callback** must have the same parameter of the same type),

type The type of synchronous behavior to use. Implemented options include ‘block’ and ‘broadcast’. Under ‘block’, only one outstanding operation for any given **id** can be outstanding, and subsequent operations will block waiting for the first. Under ‘broadcast’, the first outstanding operation for any given **id** will cause a call to the implementation, and others will just wait. When a response is received for the first, the result will be delivered to all waiting threads.

The DHT service class interface is defined as follows:

```
serviceclass DHT {
    virtual void containsKey(const MaceKey& key)
        [syncname=syncContainsKey; type=block;
         id=key; callback=dhtContainsKeyResult];
    virtual void get(const MaceKey& key)
        [syncname=syncGet; type=block; id=key; callback=dhtGetResult];
    virtual void put(const MaceKey& key, const mace::string& value);
    virtual void remove(const MaceKey& key);

    handlers DHTData;
}
```

Here you will notice essentially the same four functions, though in each case void is returned since a callback provides the result. *containsKey* and *get* both define synchronous options for callbacks, to make it easy to use DHT from C++ threads that may block. An example of how to use the generated *syncGet* method is shown here:

```
pair<string, bool> get(const string& k) {
    boost::shared_ptr<SynchronousDHT::SyncGetResult> p =
        dht->syncGet(MaceKey(sha160, k));
    return pair<string, bool>(p->value, p->found);
} // get
```

The generated *syncGet* method takes the same parameters as the original method, waits until the result is available, and returns a generated object that contains each of the parameters from the matched callback method.

4.1.2 Component Architecture

The first part of a Mace specification describes the component architecture of the service. The syntax is:

```

ComponentArch: Name Provides(?) Registration(?) GenOpts(?) Services(?)
Name: 'service' (token giving the name of the service) (?) ';'
Provides: 'provides' ServiceClass ( ',' ServiceClass ) (*) ';'
ServiceClass: (the name of a defined service class interface)
Registration: 'registration' '=' RegType ';'
RegType: 'static' | 'dynamic' '<' (registration object type name) '>'
GenOpts: Trace(?) Time(?)
Trace: 'trace' '=' ( 'off' | 'manual' | 'low' | 'med' | 'high' ) ';'
Time: 'time' '=' ( 'uint64_t' | 'MaceTime' ) ';'
Services: 'services' '{' ServiceVariable(*) '}'
ServiceVariable: InlineFinal(?) ServiceClass Handlers(?) VarName
                  SVOptions SVImpl(?) ';'
InlineFinal: 'inline' | 'final'
Handlers: '[' Handler ( ',' Handler ) (*) ']'
Handler: (the name of a defined handler interface)
VarName: (token for the variable name of the service class)
SVOptions: RegistrationUid(?) DynamicRegistration(?)
RegistrationUid: '::' (the number of a fixed registration uid)
DynamicRegistration: '<' (the type name of the registration type '>')
SVImpl: '=' ( (another service variable name) |
              (service name) '(' (parameters) ')' )
              )

```

The second line, required, gives the name of the service. The name itself is optional, and if omitted, will be the basename of the file. If provided, it should match the basename of the file for proper integration with the build system.

The next line, the provides line, tells the *service class* this service provides. This service should implement each of the methods in the interface as *downcall* transitions. Then, the compiler adds a set of upcall helper methods to return results to the higher-level service. Recall that in each *ServiceClass*, a set of handlers may be listed. Given each method *methodname* in some handler listed, a method *upcall_methodname* is created, and when called, will return an upcall to a higher-level service. The signature of the generated upcall method matches exactly the method generated from the handler interface, including the generated *registration_uid* parameter. That *registration_uid* parameter is used to determine to which of the registered handlers to

actually deliver the upcall. If the `provides` line is omitted, the default is to provide the ‘Null’ service, which contains no additional methods other than *maceInit* and *maceExit*.

For each service class listed in the `provides` line, a function of the name *new_service_serviceclass* will be generated, with the parameter list based on the services and constructor parameters (§ 4.2.2). These functions will be declared in the file named `ServiceName-init.h`. This file can be included by other C++ files without including any actual code from the service itself, which helps reduce compile time and increases code protection/isolation, as no actual details of the service need be known except for the constructor list and the service class it provides.

Lesson 2: Mace provides a sort of extension of the OSI layered model. On top of the transport layers, which are provided to services through the transport services, each distributed system will provide its own set of layers through the component architecture. However, unlike OSI, the purpose of each layer above the transport will vary for each distributed system, and no total-ordering is possible for all interfaces, as they may be composed over each other in creative ways.

The registration line determines whether handlers are registered with this service “statically” or “dynamically”, based on the `RegType` grammar element. Statically registered handlers are the default, and provide basic operation. But consider, for example, the case of the *Overlay* and *Group* service classes. These are essentially the same services, with both providing join and leave methods. The major difference between these interfaces is that the group interface provides a `MaceKey` parameter and can be used for multiple groups simultaneously. The present design of these interfaces is to use static registration. Services, such as *Scribe*, that can distinguish groups use the group interface, while other services, such as *Overcast*, would use the overlay interface. However, both of these services are implementations of a tree protocol. To enable modularity, the *Tree* service class has to include a `groupId` parameter, which is summarily ignored by services that don’t support groups. These interfaces were designed before dynamic registration was an option, and have remained to support

existing services. However, dynamic registration presents a new design option to consider.

A dynamic registration is one in which the same Handler object can be dynamically registered with multiple registration uids, in contrast with a static registration which is either fixed or based on instantiation order. A dynamic registration associates the registered object at both levels with a registration uid and a static handler registration. The new registration uid is computed as the hash of the concatenation of the static registration uid and the object being registered, allowing deterministic registration uids based on the object. This is critical, because the same object registered dynamically with the same static registration must result in the same registration uid, to support coordination across nodes.

Consider using dynamic registration to merge the *Overlay* and *Group* interfaces. Then, implementations of a *Group* service declare:

```
registration=dynamic<MaceKey>;
```

This allows them to associate a registration uid with each group key. Then, calls to join and leave use the dynamically allocated registration uid, and either service can query to determine the key associated with the registration. Tree services then need not contain a **groupid** parameter in the method signatures, instead they get it from the dynamically registered object.

Dynamic registration is a relatively new language feature, and is still being evaluated. It enables services with mostly similar interfaces to actually provide the same interface, but requires a dynamic check to make sure the supplied service actually provides the same registration. Furthermore, as dynamic registrations have to match up between the higher and lower level services, care must be taken to have a small number of dynamic registration options, lest modularity cease to be useful. In that sense, a small sacrifice in interface design may be desirable over too much flexibility that prevents modularity.

Next in the language definition come two optional generation options. The first defines the trace level of the generated code (how verbose the auto-generated tracing should be). The ‘manual’ option is not yet fully supported, and is identical in practice to ‘off’. The difference by design is that under the ‘off’ option, all logging

using the mace logging facility is removed from the generated code, including manual log messages added by users, whereas the ‘manual’ option copies manually inserted user logs into generated code, but adds no instrumentation by the Mace compiler. Both currently behave as ‘manual’. The remaining options auto-generate increasingly verbose trace logs. Under ‘low’, a statement is logged when each transition, routine, or `auto_type` method is called, logging the values of parameters (except for Message parameters, for which only the name of the message is logged). Under ‘med’, message details are logged as well, and additionally the beginning and ending of each transition, routine, and `auto_type` method are logged, allowing a sort of stack trace view of the log file, so one can see where each message was logged from. Finally, under ‘high’, in addition to the logs of ‘med’, the state of a service is logged at the end of any non-const transition, to create a view of how the state of the system changes over time.

Lesson 3: Between the compiler’s ability to automatically instrument a service with logging, and the logging infrastructure’s ability to enable and disable logs at a fine grained level, the user needs to write fewer and fewer of his/her own logs. Many services can be built and debugged without the user ever writing a single log message. This is because most user debug logging is just to indicate the state of a given variable, the occurrence of an event, or the parameter to a method. The compiler understands these at the granularity of events, which makes it possible to generate an appropriate degree of logging automatically.

The other generation option that can be specified is the type used to represent timestamps. The two options are `uint64_t`, which is slightly more efficient, and `MaceTime`, which is modelchecker-friendly. If a service will be model checked, and it contains any time-based non-determinism (comparing the latency of operations, for example), then `MaceTime` should be used. Functionally, the primary difference is that when using `MaceTime`, helper functions (and not standard operators) should be used to compare two timestamps, or to combine/scale timestamps. These functions expose to the model checker the operations of time, helping it distinguish between a deterministic time value and a non-deterministic time value, allowing it to properly

explore non-deterministic comparisons of time. During actual execution, the system runs as though with real time values.

The final portion of the architecture section of a Mace implementation is the set of lower level services a service will use. The services block can contain any number of service variables.

Lesson 4: Though simplistic layered systems, like a tree service running over just a transport service, assume a single stack of layered service objects, this is neither a limitation nor always practical. Mace instead supports a *directed acyclic graph (DAG)* of services, which preserves the notion of layering, but greater flexibility and modularity. One such DAG was shown in Figure 3.1. Each service may use multiple lower level services, and provide multiple interfaces and be used by multiple higher-level services. This requires a bit of extra effort when coding some services, as the service must be shareable, but the benefit is greater re-usability.

Each service variable is declared to provide a given service class interface. For every method *meth* in the service class interface, a method is generated for the service of the name *downcall_meth*, allowing the service to make calls into the lower level service. As with the upcall generated methods, the signature matches the generated signature exactly (modulo method remappings, § 4.2.5), including the generated registration uid parameter. The variable name of the service variable represents the statically registered uid, and can be passed as the final parameter to the downcall method to tell it on which service variable to make the call. If there is only one service variable that provides the given function, and it is not being used with dynamic registration, or if a default has been set explicitly, this parameter may be omitted, and the Mace generated code will determine upon which service variable to make the call.

Service variables represent the statically allocated registration uid they have received. The registration uid can be manually set using the ‘RegistrationUid’ syntax, and should be done if non-parallel service instances need to talk to each other. As an example, for a client and server to talk over a shared transport, they must each allocate the transport the same registration uid. As registration uids are allocated sequentially

increasing starting at 0, generally large integers should be used for fixed registration ids. To indicate that the service variable will be used with dynamic registration, use the ‘DynamicRegistration’ syntax, which specifies the type for registration. A runtime check will assure that the types match up with what the service provides. Then, in addition to the declared service class, a dynamic registration service can be used with the `DynamicRegistrationServiceClass<typename T>` service defined in the `DynamicRegistration.h` file of the `interfaces` directory under the `services` directory within Mace. Downcall methods will be generated for each of these as well.

Next, for each handler listed for a service class of a named service variable, the service implementation will generate appropriate methods for receiving callbacks from the service variable, and will generate the code for registering the service with the lower level service as a handler. The service should therefore implement each of these methods as *upcall* transitions. The set of handlers can be restricted using the ‘Handlers’ syntax, which will let the service register only the specified set of handlers with the service variable. Restricting the handlers will reduce the generated code size, reduce the number of warnings from the Mace compiler about unimplemented interface methods, and can in rare circumstances be used to share services in unconventional interface methods ways.

Each service listed in the `services` block will be included in the constructor for a service, unless modified by either ‘inline’ or ‘final’, according to the ‘InlineFinal’ syntax. This allows any service to be overridden at construction time by passing in a different implementation of the service class. The ‘inline’ keyword prevents the variable from being part of the constructor, and will even prevent the service variable from being an actual variable of this service, but instead is only constructed inline as needed to pass it into lower level services. No methods, handlers, etc. will be generated as a result of inline services. The purpose of inline services is to allow configuration of the lower level services, beneath the services used by this service. For example, to make a transport that is shared by two lower level services, make it inline, then pass it into both of the services.

The ‘final’ keyword also prevents a service from being part of the constructor, but otherwise it behaves like any other service. That is to say, the implementation or

construction specified by the ‘SVImpl’ syntax is the final choice of what will implement that service, and cannot be overridden at construction time.

Finally, the syntax describes the default (or final) implementation of a service should an overriding value not be available. This can be omitted, but then must be provided at construction time¹. Options for an implementation are to either provide another service variable, in which case a dynamic cast will be performed at construction time to ensure a match, or a service name, with its constructor parameters and services. This will cause the method *new_service_serviceclass()* to be called with these same parameters, to construct the given service.

Lesson 5: Service construction can be one of the hardest parts of building services in Mace, because by design it is highly modular. The component architecture provides a flexible language for defining how to construct lower-level services, including declaring some as inline or final. All other services may be overridden by higher-level services, or by the application creating the services. A helpful tip when creating a service is to diagram the service DAG used in the application to help you construct all the services appropriately.

4.2 Service Specification

The specification of a Mace service is given by:

```
MaceService: ComponentArch ServiceBlocks(*)
ServiceBlocks: TypeDefinitions | PersistentState | Execution
                | Debugging | Miscellaneous
TypeDefinitions: Typedefs | AutoTypes | Messages
PersistentState: Constants | ConstructorParameters | States
                | StateVariables
Execution: Transitions | Routines | Detect
Debugging: Properties | StructuredLogging
Miscellaneous: MethodRemappings | MInclude
```

Anything that appears before the ‘ComponentArch’ block is copied into the generated output files. This therefore is an ideal place to put include statements for

¹This is commonly done for simulated application services, as the modelchecker requires a service to avoid default construction anyway.

necessary C++ header files. The service blocks may appear in any order, but no more than once in the main specification file. The Transitions block is required to be present, but the others are all optional. The ‘ComponentArch’ was described in § 4.1.2, but the others are described in the following sections.

4.2.1 TypeDefinitions

There are a three types of type definitions commonly used in distributed systems implementations: Typedefs, AutoTypes, and Messages.

Typedefs

The first of these are basic C++ typedefs, which are essentially used to give a new name to an existing type. The syntax is

```
Typedefs: ‘typedefs’ ‘{’ Typedef(*) ‘}’
Typedef: ‘typedef’ ExistingType NewTypeName ‘;’
```

The syntax is identical to a simple renaming typedef in C++. Typedefs in Mace cannot map new structure or class definitions to a typename, as was commonly done in older C designs. In Mace, we find that these typedefs generally serve two common usages, (1) to give names to various integral types that represent something specific, such as `registration_uid_t`, or (2) to map complex STL types or template types into more conveniently typed names. In addition to standard C++ types, the typedefs block may reference earlier defined typedefs, or auto-types.

AutoTypes

Auto types in Mace are a convenient place to create all kinds of struct or class types, while letting the compiler generate formulaic domain specific code for you. The syntax is

```
AutoTypes: ‘auto_types’ ‘{’ AutoType(*) ‘}’ ‘;’
AutoType: TypeName TypeOpts ‘{’ Typedef(*) Field(*) CppConstructor(*)
          Method(*) ‘}’ ‘;’(?)
TypeOpts: ‘__attribute((’ TypeOpt (‘,’ TypeOpt)(*) ‘))’
TypeOpt: AttributeName ‘(’ SubTypeOpts(?) ‘)’
SubTypeOpts: SubTypeOpt (‘;’ SubTypeOpt)(*) ‘;’(?)
```

```

SubTypeOpt: (Value | Key '=' Value)
Field: Type VarName TypeOpts(?) ( '=' DefaultValue )(?) ';'
Method: MethodDecl MethodOptions(?) MethodImpl
MethodDecl : (C++ function declaration without terminator)
MethodOptions : '[' (key) '=' (value) ( ';' (key) '=' (value) )* ']'
MethodImpl : '{' (C++ function body) '}'

```

There may be any number of auto types, and they may reference previously-defined auto-types, or types defined in the typedefs block. The common use for an auto-type is to behave essentially like a struct—each field listed will be copied as a public member of the output class. Several auto-generated methods will be generated: (1) a default constructor which initializes each parameter to its default value, or `Type()` otherwise; (2) a constructor with each parameter as a field (in the same order) with default values as given for fields, which sets the value for each field based on what's passed in; (3) accessor methods of the form *get_fieldname* which return a copy of the field—these can be used as function pointers to STL algorithms or collection methods taking a function pointer to compute over the objects; (4) a method to print the object by printing each of its fields, and (5) methods to serialize and deserialize the object according to the `Serializable` interface, by serializing and deserializing each object in turn, which includes normal and XMLRPC serialization.

An example auto types block might look like:

```

auto_types {
    VersionedData {
        int32_t version;
        string data;
    };
}

```

This defines a class called `VersionedData` in the scope of the service, with public fields `version` and `data`, auto-serialization and printing methods, a default constructor, and a constructor which takes each of the fields. Many other examples of auto types can be found in the downloadable Mace code.

There are many ways to customize an auto-type, and these will be described now.

Typedefs As with the typedefs block, new types may be named within the context of an auto-type. These will be created as inner-types of the auto-type, and serve the same purpose as the typedefs block.

CppConstructors One can define any number of C++ constructors for your auto-type, which can override the default constructors that are otherwise auto-defined. The syntax for these is as it would be for standard constructors in C++.

Methods You can define any number of C++ methods as member functions of the auto-type. These use the normal syntax for defining methods in C++, but may contain an optional ‘MethodOptions’ block between the method signature and the method body, which may contain a ‘trace’ option that changes the trace level for this function only. These methods cannot directly access variables or methods of the service itself, though the auto-type is declared as a friend of the service instance. The auto-type may contain a pointer to the service instance, which can be passed in by the constructor, and this pointer can be used to access variables and methods of the service.

FieldOptions Each field of the auto-type can be given a variety of options that control how it is generated within the auto-type. These options overlap substantially with other fields in the specification, and so are described in detail in § 4.3.1.

Serialize The ‘serialize’ option with sub-option ‘no’ (`serialize(no)`), can be specified for the auto-type, which causes it not to have the serializable methods generated for it.

Comparable Auto-type objects are not, by default, comparable—meaning there are no equality or comparison operators defined automatically. The ‘comparable’ option can be specified, and admits three sub-options: ‘equals’, ‘lessthan’, and ‘hashof’. The value provided for each should be ‘default’, other options are not presently supported. If ‘equals’ is supplied, then an `operator==` method will be defined. If ‘lessthan’ is supplied, then an `operator<` method will be

defined, and ‘equals’ will be implied. Finally, ‘hashof’ will cause a hash template to be generated for use in hash_maps. A full specification would be `comparable(equals=default; lessthan=default; hashof=default)`. The generated methods will consider the fields in order, omitting those fields that are marked ‘notComparable’.

Private Giving the ‘private’ option with the sub-value ‘yes’ has the effect of making the fields of the auto-type private, instead of the public default, while leaving the auto-type methods as public. This is to allow normal C++-style protection for parameters.

Node The node attribute tells the Mace compiler this auto-type represents a node. This causes it to add several auto-generated features for the type. First, an ‘id’ field is generated for the object, which is hidden and protected. The value can be retrieved using *getId()*, which will return a copy of the id. The id field will be the first parameter to any constructor, and must be set at the time of construction. The purpose of setting the node attribute is twofold. First, future compiler options can take advantage of types that are known to represent nodes, in the same way Mace takes advantage of the MaceKey and NodeSet types, but with more richness. Second, the node attribute is for use with the `mace::NodeCollection` data type, which is a hybrid map and set type. Nodes in the collection can be located and referenced using only their MaceKeys, as with a map, but the key is also automatically a field of the mapped type, or value type, as though it were a set. Part of this support for node collections is the addition of a `getScore` function, which is the default function pointer used with the `greatestScore` and `leastScore` functions of the node collection. By default, `getScore` simply returns 0.0, but by giving the name of a field as the ‘score’ sub-option, it can return that field instead. (example `node(score=delay)`).

Messages

The messages block allows one to define messages for sending to other nodes. The syntax is described below:

```

Messages: 'messages' '{' Message(*) '}' ';'
Message: TypeName TypeOpts '{' Fields(*) '}' ';' (?)
Fields: Type VarName TypeOpts(?) ';'

```

Example:

```

messages {
  JoinRequest {
    MaceKey source;
  }
  JoinReply __attribute__((number(6))) {
    NodeSet peers;
  }
}

```

This example contains two messages, one named *JoinRequest*, and the other *JoinReply*. Each message has a single field. The *JoinReply* message will be allocated message number 6 while *JoinRequest* will get its number by sequential assignment.

The only valid type option for message fields is the ‘dump’ option, which can be used to restrict how a message field is printed in its output methods. The sub-options and values for ‘dump’ are described in § 4.3.1. Serialization is automatically generated for each message, and cannot be modified. Methods similarly cannot be added to messages. To support distinguishing messages during deserialization, each message of a service is assigned a sequential number in the order listed, from an `int8_t` numerical space. To support sharing messages across services, the ‘number’ type option may be set for the message type, with a sub-value of the desired number. Subsequent messages will be one greater than this message. (Example: `number(5)`). The serialized form, then, will contain the number at the beginning of the bytestream, and is automatically handled for dispatch and deserialization by deserialization code.

Messages are highly optimized for minimizing copies of data, and therefore cannot be stored past their stack-lifespan. The message itself is a set of const references, which either point to the variables passed in by constructor, or to the variables of a separate struct generated just for the message. The former is used for serializing a message, while the latter for *de*-serializing a message. Messages therefore should not be passed by copy or stored, as these references will become invalid and cause memory problems.

In some cases, it is desirable to save a message, so as to preserve its contents for future delivery. Presently, there are two ways to do this. First, you can create an auto-type with the same fields, and either send the auto-type in the message (and store the auto-type), or write a constructor for the auto-type that takes the message and initializes all the fields. The second way to do this is to serialize the message into a string, so it may be deserialized later. If the goal will be to re-deliver the message, the latter option may be preferred, as the string can be delivered directly by calling a service's own deliver method, with the string serialized as an argument.

In addition to serialization and logging code, messages are handled specially by the Mace compiler, as will be described in § 4.2.5. In general, however, special message handling is to leverage the fact that the Mace compiler knows, for each service, the complete set of messages it may send and receive, and so generates appropriate code to handle multiplexing and demultiplexing of the messages efficiently rather than trying to deserialize an arbitrary object, as happens in other languages like Java.

4.2.2 Persistent State

Few distributed systems are stateless. Thus, the Mace specification language has to provide mechanisms for storing the state of each node in the system. I have identified four types of persistent state a service maintains—(1) compile-time *constants*, which support compile-time optimization of the service based on the constant value, (2) run-time constants, which are set during the service initialization by its *constructor's parameters*, (3) high-level *states*, which are used to break-up the implementation into phases, and (4) detailed *state variables*, which let the service maintain specific information in addition to its current phase of operation.

Constants

Syntax:

```
Constants: 'constants' '{' CppInclude(*) Variable(*) '}'
Variable: Type Varname '=' Value ';'

```

The constants block contains two portions. The first is a set of C++ header files that may be needed to define types for constants. To make compilation more

efficient, a file, *ServiceName-constants.h*, is generated for defining the constants, and this file only includes “MaceBasics.h”. Thus, if the type is not defined there, it will need to be `#include-d` *inside* the constants block. Two cases in which this is necessary are constant `std::string` variables and constant `mace::MaceTime` variables. The syntax of a `CppInclude` are the same as in standard C++. The variables themselves are defined as other C++ variables are defined. Each must have a value provided, which will be the compile-time constant used for the variable.

Constructor Parameters

Both constants and constructor parameters are constants at runtime. The distinguishing feature of these is that constructor parameters may be set at runtime rather than compile time, limiting optimizations the compilers may make. Any parameter listed here will be placed, in order, in the initialization function of a service so it may be over-ridden at instantiation time.

Syntax:

```
Constants: 'constructor_parameters' '{' Variable(*) '}'
Variable: Type Varname '=' Value ';'

```

Unlike the constants block, there is no need for C++ includes in this block, as it will enjoy the set of includes the rest of the service code does as well, including any files included from the constants block. Each variable in this block must be given a default value, to support a parameter-less instantiation of the service.

In actuality, two initialization functions will be generated for services that use both lower level services and that have constructor parameters. The first will have the list of services first, and the second will have the list of constructor parameters first. This supports default initialization of either set while only passing in the other. In fact, there will only be one constructor generated, and it will contain no default parameters. This was done for simplicity of code generation and shortness of compilation – all of the default value mapping happens in the initialization functions, which are defined in *ServiceName-init.cc* and declared in *ServiceName-init.h*. As such, no service implementation file depends directly on another service, so when a service is modified, only the initialization functions of dependent services need be recompiled.

States and State Variables

The remaining state is the mutable persistent state. It comes in two types—high level states, and lower level state variables. To understand the distinction, consider a TCP implementation. The TCP implementation defines a state-transition diagram, which includes the states CLOSED, LISTEN, SYN-RCVD, SYN-SENT, ESTAB, FIN-WAIT-1, FIN-WAIT-2, CLOSING, CLOSE-WAIT, TIME-WAIT, LAST-ACK, and CLOSED. These determine the phase of the transport protocol, and determine how to handle incoming packets at a high-level. But nodes also maintain some specific information, such as the congestion window, and the local and remote sequence numbers, which are not encoded in the state-transition diagram. This dichotomy captures exactly the design difference between states and state variables.

Lesson 6: While the high-level state could be captured as an enumeration in a normal state variable, it is more natural to use the states block to define these high-level states. Doing so also exposes to the Mace compiler and tools that these state transitions are more significant to the operation of the service, and this information can be used to better visualize the execution or search the state space in model checking tools.

Lesson 7: Some services, because they provide logical service for multiple purposes, would be better served by a high-level state that is not only service specific, but also specific to a group or registration id. These services currently tend not to use the high-level state variable, to the detriment of automated understanding. We are currently considering new language features to preserve this dichotomy, even for these kinds of services.

States Syntax:

States: `'states' '{' StateName ';' (StateName ';')(*) '}'`

In addition to the states listed as StateName(s), two states are automatically added to every service: *init* and *exited*. The service will begin in the *init* state. When the service is exiting, though calls to the *maceExit* event handler, the service will automatically transition to the exited state. Once in the exited state, no transitions

may occur except *maceExit*, so a service should only transition itself to the exited state if it wishes to prevent all future transitions. Usually this is not done by the programmer, as it is handled automatically on the call to *maceExit*. In each service, the state of the service can be checked and modified through the *state* variable. An enumeration is created from the set of states, so the *state* variable may be compared to the names of the states, or assigned a new value.

To capture the lower-level states of the service, these variables are defined as part of the ‘StateVariables’ block.

StateVariables Syntax:

```
StateVariables: 'state_variables' '{' Variable(*) '}'
Variable: Type VarName TypeOpts(?) ( '=' InitialValue ) (?) ';'
TypeOpts: '__attribute__(( TypeOpt (',' TypeOpt)(*) ))'
TypeOpt: AttributeName '(' SubTypeOpts(?) ')'
SubTypeOpts: SubTypeOpt (',' SubTypeOpt)(*) ';' (?)
SubTypeOpt: (Value | Key '=' Value)
```

These variables generally share the standard C++ syntax for declaring variables, and are generated as private member variables of the service. They may only be accessed or referenced from code within the service specification—such as transitions, routines, or as discussed in auto-type methods—and may not be externally influenced. This supports encapsulation and checking because only by calling a transition may the state of the service be affected. These variables can also be affected by a variety of attribute field options, which are described in detail in § 4.3.1. The Mace compiler does not enforce, but it is strongly urged, that no pointer or reference variables are placed in a state variables block, because it may violate the encapsulation that is assumed by MaceMC and other Mace tools. State variables may be set to an initial value, or will be initialized to *Type()* otherwise. Variables can be of any standard C++ type, a type named in a typedefs block, or an auto-type. They should not, as described earlier, be a message type.

One state variable type in particular deserves special mention, because it is treated specially by the Mace compiler. This is the *timer* type. In its basic form, a user can define a timer as such:

```
timer printer;
```

This creates a new timer, with the name **printer**. Timers are used in Mace to schedule future callbacks as timed events rather than in response to external stimuli. There are two main kinds of timers: plain and multi-timers. Plain timers can be recursive or not, depending on the type options. Recursive timers are simply plain timers that automatically reschedule themselves as they expire. The service must schedule them the first time to start the sequence, and they will fire until cancelled. Multi-timers cannot be recursive, and have the property that they may be scheduled multiple-times simultaneously. Timers may also contain a set of variables that are passed into the schedule methods, and returned during the expire transition. Because of the various specializations, each timer is generated individually.

Example timer definitions are:

```
timer plain;
timer<int, float> plainWithVariables;
timer recursive5sec __attribute__((recur(5000000)));
timer multi __attribute__((multi(yes)));
timer<int, float> multiWithVariables __attribute__((multi(yes)));
```

Timers support these methods:

schedule() The schedule method takes a *uint64_t* parameter that is the number of microseconds until the timer should expire. Additionally, there is a parameter for each value to be stored with the timer, which the timer will make a copy of for safekeeping. In a multi-timer, the schedule method will return the exact timestamp the timer is scheduled to expire, which is used as a key for other methods. In a plain timer, schedule will assert that the timer is not scheduled.

reschedule() This call is only generated for plain timers. It differentiates itself from schedule only by the fact that it will cancel this timer if already scheduled before scheduling it as requested.

isScheduled() This call with no parameters returns a boolean indicating whether this timer is presently scheduled. For multi-timers, one can pass in a specific timestamp, from the return value of a call to schedule, and the timer will tell whether is a timer scheduled with that timestamp. Timers with variables will also implement an *isScheduled* method that takes the variable list, and will return true if there is a scheduled timer with that set of variables.

nextScheduled() This call, for all timers, returns the first time this timer is scheduled to expire.

cancel() The cancel method can be used to cancel timer expirations. The form without parameters, for both plain and multi-timers, cancels all outstanding expirations of the timer. For multi-timers, there will be two additional forms. One takes a timestamp, and cancels only that specific expiration. The other takes the set of timer variables, and cancels all matching timer expirations.

The transitions that occur when a timer expires are discussed in § 4.2.3.

4.2.3 Execution

There are three places in the service where designers can place executable code that executes directly in the scope of the service itself. These are *transitions*, which are event handlers, *routines*, which are private methods of the service, and *detect transitions and triggers*, which are normal transitions, but separated for convenience. It should be stressed that nowhere in Mace execution code within a service context should the thread be allowed to block, as this would prevent other threads from executing their events in the current Mace implementation.

Transitions

Transitions are the key of all Mace services—it is through transitions that all useful work of the service occurs. There are four kinds of transitions: upcalls, downcalls, scheduler, and aspect transitions. Upcalls are transitions that come from lower layers, namely the registered handlers of services in the services block. Downcalls are transitions that come from higher layers: interfaces in the provides statement of this service. Scheduler transitions are expiration transitions of a timer variable, and aspect transitions are those that occur atomically at the end of a transition in which a monitored state variable is modified.

Syntax:

```
Transitions: 'transitions' '{' GuardedTransitionSet '}'
GuardedTransitionSet: 'guard' Guard '{' GuardedTransitionSet '}'
                    | Transition(*)
```

```

Guard: '(' (const boolean expression) ')'
Transition: TransitionType Guard(?) ReturnType(?) TransitionName
            ParameterList MethodOptions '{' CppMethodBody '}'
TransitionType: 'upcall' | 'downcall' | 'schedule' | AspectType
AspectType: 'aspect' '<' Variable (',' Variable)(*) '>'
TransitionName: (function name from the valid interface methods)
ParameterList: (Type)(?) VarName (',' (Type)(?) VarName)(*)

```

In the execution of a service, event processing happens by threads acquiring the privilege to execute a given event, then calling that event handler. Because different threads are competing to fire events, the order of events is unpredictable, and fair-sharing across threads is handled by operating system primitives. Associated with each transition is a stack of guard boolean expressions, which defaults to a single 'true' if empty. This is the stack parsed from the nested GuardTransitionSet blocks into the specific transition Guard. For example, consider this block:

```

transitions {
  downcall maceInit() { /* event body */ }
  guard (state == waiting) {
    upcall (src == me) deliver(src, dest, const JoinRequest& r) {
      //error?
    }
  }
}

```

It shows a *maceInit* event, whose guard stack defaults to *< true >*, and a deliver event for a *JoinRequest* message. The guard stack for the deliver event is *< (state == waiting), (src == me) >*. When a thread receives permission to execute its event, it will find the first matching and enabled transition, and execute that transition. An enabled transition is one whose guard stack returns true. In the case of the deliver event, the guards stack would be true if the state is the waiting state and the source is the local node. If no matching and enabled transition is found, the default behavior from the interface description will be used instead.

Lesson 8: There is functionally no difference between defining multiple transitions with different guards, and having a single transition with an *if/else if/...* The latter can even be more efficient due to limitations of the current implementation. Despite this, the guards are recommended, because they allow the user

to group transitions by something other than transition type, the compiler to better understand how the system is structured, and the analysis tools to make better use of the guard information.

There are a few caveats and exceptions to the default behavior, which have to do with the transition options described in § 4.3.2. Basically, transitions may be declared as ‘pre’ or ‘post’ transitions, and may be marked as only executing once. Marking a transition as execute-once is equivalent to adding a guard that checks if the execution has happened before, and returns false if so. Otherwise, all pre transitions will be considered for execution, based on their guards and will execute in the order they appear in the specification. Then, after all pre transitions execute, the first matching normal transition will execute, followed by all matching and enabled post transitions, in the order they appear. Finally, aspects and deferred actions will be executed. Since these can in turn cause other things to execute by calling other methods, their interaction is not well understood, so care must be taken when combining all of these features.

Transition code is written in C++, and can make calls to other services (using `upcall_` and `downcall_` functions), to routines, or asynchronous, non-blocking libraries. Transitions can also defer certain routines, downcalls, and upcalls until the end of events, by prefixing those calls with ‘defer_’. To defer an ‘upcall_deliver’, for example, the programmer would write ‘defer_upcall_deliver’. Note that no return value may be collected from those calls, as they are not immediately executed. The specification of transition signatures allows most types to be omitted. The return-type may always be omitted, and will be taken from the interface definition. Also, the types of the parameters may be omitted, as long as they are not needed to distinguish which interface the transition should match.

The parameters of a transition must match exactly the type listed in the interface definition. However, there are a few exceptions. First, the *registration_uid* parameter common to upcalls and downcalls may be omitted, though it should not be omitted if the transition is a downcall and the variable needs to be saved or used for an eventual related upcall. Second, the parameters of a scheduler transition are taken from the template parameters of the timer type, except that they are passed

to the scheduler transition as reference parameters. Finally, the parameters of an aspect method are constant reference parameters to variables of the same type as the template variables of the aspect.

An aspect transition is declared as an aspect that monitors the changes of a set of state variables. The compiler checks this list to make sure they are all state variables, and bases the parameters of the transition on the types of the state variables. An aspect will detect when a state variable changes, and if any of the flagged state variables change, the guard will be checked to see if it is enabled. If so, the prior value will be passed in as a parameter to the aspect transition, by constant reference.

Routines

Syntax:

```
Routines: 'routines' '{' (Routine | RoutineObject)(*) '}'
Routine: Method
RoutineObject: ( 'class' | 'struct' ) TypeName '{' CppBody '}' ';'

```

Routines are nothing more than private member functions of a service. Routines may not be called from outside the service, thus entry points into the service are restricted to transitions. Routines are defined using normal C++ syntax for methods, though they may contain `MethodOptions` between the signature and the method body.

As a special case, the routines block may also contain a class or struct, who's implementation will be copied verbatim from the routines block to the service. This use is deprecated, but remains to support defining a functor for use as a secondary sort function on objects that have a different primary sort.

Detect

The detect block is a shortcut for a common setup in distributed systems. The basic idea is that for a node or group of nodes, often a protocol wants to exchange periodic information, and will declare the remote peer failed if there is no response in an appropriate amount of time. This is often implemented as a set of timers, state

used to keep track of the last time a node was heard from, and messages to exchange information. Timers are reset when an appropriate response from a peer is seen. This block was developed in Mace as an experiment to clean the code related to this.

Syntax:

```

Detect: 'detect' '{ DetectSpec(*) }'
DetectSpec: Id '{ DetectBody }'
DetectBody: DWho DTimerPeriod(?) DWait(?) DInterval(?) DTimeout(?)
            DWTrigger(?) DITrigger(?) DTTrigger(?)
            SuppressionTransitions(?)
DWho: ( 'node' | 'nodes' ) '=' state-variable ';'
DTimerPeriod: 'timer_period' '=' Expression ';'
DWait: 'wait' '=' Expression ';'
DInterval: 'interval' '=' Expression ';'
DTimeout: 'timeout' '=' Expression ';'
DWTrigger: 'wait_trigger' '(' varname ')' '{ MethodBody }'
DITrigger: 'interval_trigger' '(' varname ')' '{ MethodBody }'
DTTrigger: 'timeout_trigger' '(' varname ')' '{ MethodBody }'
SuppressionTransitions: 'suppression_transitions' '{'
                        GuardedTransitionSet '}'

```

In the detect block, multiple detect specifications (`DetectSpec`) can be defined. For each specification, there is a name *name* for the specification, and either a node or set of nodes must be defined. Given the name of the specification, a method called *reset_name* is generated, which takes a `MaceKey` parameter. Whenever the method is called, that node is marked as having responded, and the node clock is reset. The node set of nodes listed in the syntax refers to a state variable, and for each node listed, a virtual clock is started. A timer is generated for the detect specification, which expires at the interval specified by *timer_period*. Whenever the timer expires, each node's virtual clock is consulted, and timers are fired as appropriate. The triggers specified are executed according to the chart in Figure 4.1. First, a *wait* period occurs before any trigger. Then, the wait trigger is executed, followed by the first interval trigger. Then, every time the *interval* passes, the interval trigger is executed again. Finally, if *timeout* time elapses on the virtual clock from its start, the timeout trigger is executed.

As a convenience, transitions may be defined as part of the detect specification. These transitions are equivalent to transitions in the transitions block, and will be

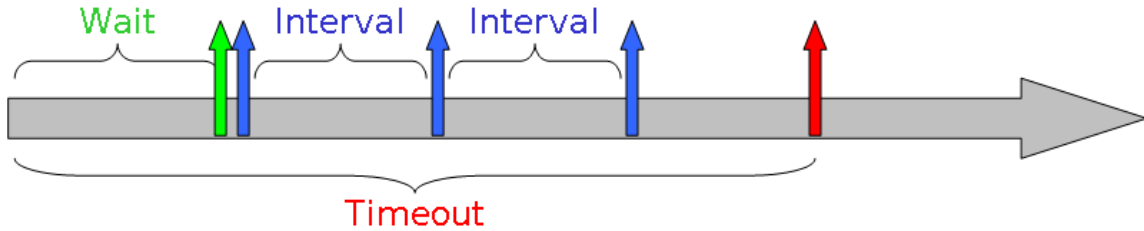


Figure 4.1: This figure shows how timeout periods and triggers interact in detect specifications in Mace. Time is represented by the arrow going to the right. Arrows pointing up indicate triggers executing.

included as though implemented there, though they are automatically marked as pre transitions, to prevent conflicts with transitions in the rest of the specification. The idea is that as a convenience, suppression can often be expressed in separate pre-transitions, to keep these concerns separate.

Many of the detect declarations are optional and have default values. These are currently set as follows:

timer period. By default, the timer will expire every 1 second, measured in microseconds, so the specific value is 1000000.

wait period. The default wait period is 0, meaning the intervals begin immediately.

interval period. The interval period defaults to the setting of the timer period.

timeout period. There is no timeout by default – intervals continue until the peer clock is reset.

Triggers can also be left undefined, causing no action to occur when the timeouts are passed.

4.2.4 Debugging

There are two parts of a Mace specification that are focused on debugging: the definition of properties and structured logging. Properties are conditions for verifying the correctness of a service, and structured logging is the idea of logging by calling

a method rather than writing a text message, which supports more efficient logging and adds structure for automated tools.

Properties

The properties block can be used to define both safety and liveness properties. Safety properties are state conditions that must always be true, at every point during execution. Liveness properties are state conditions that should eventually hold along any execution.

Syntax:

```

Properties: 'properties' '{'
            ( LivenessProperties | SafetyProperties)(*) '}'
LivenessProperties: 'liveness' '{' Property(*) '}'
SafetyProperties: 'safety' '{' Property(*) '}'
Property: Id ':' GrandBExpression ';'
GrandBExpression: BExpression ( Logical BExpression )(*)
BExpression: Equation | BinaryBExpression | BBlock | Quantification |
             '\not' GrandBExpression | Var
Equation: Expression Op Expression
BinaryBExpression: Element ElOp Set | Set SetOp Set
ElOp: \in | \notin
SetOp: \subset | \prosubset | \eq
BBlock: '(' GrandBExpression ')' | '{' GrandBExpression '}'
Quantification: ( '\forall' | '\exists' | '\for{' Op '}' Count '}' )
                VarName '\in' Set ':' GrandBExpression
Op: < | > | \geq | \leq | = | \neq
Logical: \implies | \and | \or | \xor | \iff | \not

```

Each property is given a name, and defined with a 'GrandBExpression'. The idea of the language is to use a latex-like language. In large part, a goal was to be able to copy/paste the property into a latex document, and have it generate the mathematical formula to be tested.

Some common operators and expressions include:

- equality: =
- inequality: \neq
- Element-Set (ElOp): \in , \notin

- Set-Set (SetOp): `\subset`, `\probersubset`, `\eq` (set equality, not C++ equality)
- Numerical: `=` (C++ equality), `<`, `>`, `\geq`, `\leq`
- Logical: `\implies`, `\and`, `\or`, `\xor`, `\iff`, `\not`

All properties presently have to start with a quantification over the nodes (special variable `\nodes`). The quantification options are (n can be any variable name):

- `\forall n \in \nodes`:
- `\exists n \in \nodes`:
- `\for{OP}{COUNT} n \in \nodes`:, Where Op is `<`, `>`, `\geq`, `\leq`, `\neq`, or `=`, and $Count$ is a number.

For example,

`\for{=}{4} n \in \nodes`

would be “for exactly 4 n in nodes”, while

`\for{<}{4} n \in \nodes`

would be “for fewer than 4 n in nodes”.

One can use n as a reference to a node, and use the dot operator to access its state variables, `state`, or a const transition/routine.

Any variable that is a MaceKey can be “promoted” to a node reference. That is, suppose a service had a MaceKey variable `peer` where it stored the address of a peer.

Then the property:

`\forall n \in \nodes: n = n.peer.peer`

says that for each node n , n is the peer of its peer.

You can also compute the closure of a recursion, in a property like this:

`\forall n \in \nodes: n.peer(.peer)* \eq \nodes`

which would be: The set containing $n.peer$ and the closure formed by continually adding each new node's peer to the set, is the set of all nodes.

We use this property to check that successor pointers in Chord form a ring of all nodes. You can also take the cardinality of a set like this:

```
\forall n \in \text{nodes}: |\text{n.peer}(.peer)*| = 4
```

which would test that for each node n , the closure of $n.peer(.peer)*$ has size exactly 4.

You can use multiple quantifications, and parentheses to designate order of operations. This gives you something like:

```
\forall n \in \text{nodes}: (\text{n.state} = \text{init} \text{ or }
                    (\exists m \in \text{nodes}: m.p = n)
                    )
```

Which would say, for all nodes n , either n is in the init state, or there exists a node m such that $m.p = n$. The parentheses may not actually be needed here, but can provide clarity. Additionally, curly braces can be used instead of parentheses for nesting, as desired for clarity.

Each property should begin with a property name and a colon, contain a “grand boolean expression” (GrandBExpression in the grammar), and end with a semicolon.

StructuredLogging

Structured logging is the idea that instead of writing a text log message, such as:

```
maceout << "Received block " << b << " from peer " << peer
        << Log:end;
```

One could just write this as a function:

```
receivedBlock(b, peer);
```

This approach would in theory allow the logging subsystem to understand the structure of the log, rather than just be presented with the human-readable string of a log message. For example, it could know there is a log with name “receivedBlock”

with two parameters, one being a block number and the other a MaceKey called peer, rather than a string with no particular meaning. This allows for shorter logs with lower overhead, while allowing post-processing to reformat logs for human reading or for automated tools to take the fields and process them.

To support this, we first add a structured logs block to the service specification according to this syntax:

```
StructuredLogging: 'structured_logging' '{' MethodDecl(*) '}'
MethodDecl: MethodName '(' CppParameterList ')'
```

This generates a method with the given name, which when called, generates a log entry structured with the fields of the method. The selector string of the method will be `ServiceName::MethodName`. It can be called from any transition or routine of the service.

4.2.5 Miscellaneous

There are two other parts of the Mace specification that don't fit nicely in the rest of the groupings. The first is the method-remappings block, which allows you to remap the signatures and default values of methods in the defined interface, particularly to support serialization and deserialization of parameters. The second is an include-file directive, which allows portions of a Mace service to be provided from separate files.

MethodRemappings

The method remappings block can be used to remap the signatures of either transitions the service *implements*, both upcalls and downcalls, or the interface methods it *uses* (upcall_methodname and downcall_methodname) from the interface of either services in the services block, or the registered handlers of the provided interface.

The syntax breaks these methods into groups to keep their usage straight, since a service might use the same interface it provides, and the remapping might not work in both cases.

```

MethodRemappings: 'method_remappings' '{' (Uses | Implements)(*) '}'
Uses: 'uses' '{' UsesMethodRemap(*) '}'
Implements: 'implements' '{' (Upcalls|Downcalls)(*) '}'
Upcalls: 'upcalls' '{' MethodRemap(*) '}'
Downcalls: 'downcalls' '{' MethodRemap(*) '}'
UsesMethodRemap: ( 'upcall_' | 'downcall_' ) MethodRemap
MethodRemap: MethodName '(' RemapParam (',' RemapParam)(*) ')'
RemapParam: NoRemap | RemapValue | RemapType
NoRemap: Type Name(?)
RemapValue: Type Name(?) '=' Value
RemapType: Type ('<-' | '->') Type

```

The first division within the method remappings block is the division between methods that are *used* and methods that are *implemented*. Used methods refer to ones that this service specification may call from within transition or routine bodies, and that begin with the prefix 'downcall_' or 'upcall_'. These methods will come from either the provided interface of services in the services block, or the handlers defined for the interfaces this service provides. When included in the 'uses' block, a method should include its prefix, to help the Mace compiler distinguish in which interface to search for the method, and to prevent conflict. The `registration_uid.t` parameter may be omitted, though is often included to specify the default service to which to deliver the call. Including the prefix also makes it consistent in how it is shown in this block with how one actually calls the method from service implementation.

Implemented methods are those transitions that are part of the upcall or downcall interface of this service. The 'downcalls' sub-block is for transitions that appear in the provides interfaces of the service being implemented. The 'upcalls' sub-portion is for transitions that appear as handler methods of the used services provided interfaces. For both of these cases, type signatures may be remapped, but remapping default values has no effect, since the caller will be the one supplying the value. As a result, the `registration_uid.t` parameter is often omitted from the implemented methods, though it is not removed from the transition signature, which also has an option of including it as needed.

To remap a value for any parameter, define it with a new value as in a C++ function definition, though the parameter name is optional. You cannot give a default value to a parameter that did not already have a default value. However, you can

let the default value be a variable name, unlike in C++ where the value must be a constant, which allows the default value to change during the execution.

To instead remap the type signature of any parameter, place the type of the actual parameter from the interface definition on the right, and the new type on the left, with an arrow between to indicate in which direction the translation should occur. For a *used* method, the arrow should point to the right, indicating the passed in parameter (on left) should be translated to the type in the actual signature on the right, while for an *implemented* method, the arrow should point left, indicating the calling method will provide a variable of the type on the right, which should be translated to the type on the left before implementing it within the service. At present, only string types may be remapped to other types. A string may either be mapped directly to another type, in which case simple serialization is used, or it may be mapped to the ‘Message’ type, which will cause special handling that will allow remapping to/from any of the expected message types defined for the service, providing support for dispatching the appropriate message.

When remapping, if the type being remapped is a constant reference, it will be translated into/from the string provided, while if it is a non-constant reference, it will be translated both at the beginning and ending of the transition. This allows, e.g. a lower level service to pass in a string to an upcall, which might be remapped to a type, modified in the higher-level service, and then re-serialized into the lower level service’s string as an update.

One special case occurs if in a *used* method a string is remapped from a ‘Message’. In this case, after doing so, the method may be further remapped to provide different default values depending on which Message is passed in.

Also, the special syntax described earlier in § 4.1.1 is sometimes used by interface designers to provide default remappings, so individual service implementers need not worry about them. In particular, the Transport service class provides default remappings that the string in the route and deliver calls should be remapped to a Message. This portion of the interface looks like this:

```
mace services {
  method_remappings {
    uses {
```

```

        downcall_route(const MaceKey&,
                        const Message& -> const std::string&,
                        registration_uid_t);
        downcall_send(const MaceKey&,
                       const Message& -> const std::string&,
                       registration_uid_t);
    }
    implements {
        upcalls {
            deliver(const MaceKey&, const MaceKey&,
                    const Message& <- const std::string&);
            messageError(const MaceKey&, TransportError::type,
                         const Message& <- const std::string&,
                         registration_uid_t);
        }
    }
}

```

This automatically causes services that use a Transport service to have the indicated `method_re mappings` block. This block remaps Message to string in the route and send used calls, and the string back to a Message in the deliver and messageError calls. Note that this can be overridden in a service by defining the same block, but without the remapped types.

```

method_re mappings {
    uses {
        downcall_route(const MaceKey&, const std::string&,
                        registration_uid_t);
        downcall_send(const MaceKey&, const std::string&,
                        registration_uid_t);
    }
    implements {
        upcalls {
            deliver(const MaceKey&, const MaceKey&, const std::string&);
            messageError(const MaceKey&, TransportError::type,
                         const std::string&, registration_uid_t);
        }
    }
}

```

MInclude

Finally, a Mace file may include parts of its specification from other files. The `#minclude` directive may appear anywhere in the service specification after the `ComponentArch` syntax, and outside of any `ServiceBlocks`. Any of the service blocks may be included, even if the block is also defined in the source file². These blocks will be merged with the blocks in the source file, and sorted by order of appearance.

4.3 Options

Many options may be provided at various parts of a Mace specification, which are directives to the Mace compiler for tailoring the generated code for the field, type, or method shown. The type options are generally not shared, and were described above with each type. Below are the field and method options and their descriptions.

4.3.1 Field

Field options are specifically intended to share syntax from the C++ attribute syntax. This syntax is quite ugly, but is used for consistency. Specifically, this syntax is:

```
TypeOpts: '__attribute((' TypeOpt (',' TypeOpt)(*) '))'
TypeOpt: AttributeName '(' SubTypeOpts(?) ') '
SubTypeOpts: SubTypeOpt (';' SubTypeOpt)(*) ';' (?)
SubTypeOpt: (Value | Key '=' Value)
```

What follows is a description of each of the type attributes, and valid sub-type keys and values for fields. With each is also an indication of in what contexts the field option may be applied. Possibilities include the fields of an auto-type, message, or state variable. In the state-variable case, some options are reserved for timer-type variables.

dump (Context: `auto_types`, `message`, `state_variables`) The `dump` option refers to whether to include the value of the field when printing the containing object to a human-readable string. By default, the value is 'yes'. To cause the variable

²Recall that normally a block may appear only once, but included blocks are an exception.

not to be dumped, the value ‘no’ can be provided. In some cases, you may wish to control the standard print method separately from the version that is used by the model checker to print the state of a service. Printing the state should omit things like a timestamp that will vary during model checking but don’t indicate distinct states. To do this, the sub-key ‘state’ can independently set to ‘yes’ or ‘no’.

serialize (Context: `auto_types`, `state_variables`) The `serialize` option refers to whether or not to encode the variable when converting the enclosing object for transmitting in a wire protocol. Valid values are ‘yes’ and ‘no’. This option is not valid in a message field, because it is unreasonable to declare a message field as not serialized. You might not serialize a field if it simply represents local state that is either only relevant locally, or is computed from other values.

recur (Context: `state_variables:timer`) This option marks a timer as a recursive timer. Recursive timers automatically reschedule themselves when they expire. The value passed in should specify the period of the recursion.

multi (Context: `state_variables:timer`) This option marks a timer as a multi-timer. Multi-timers can be scheduled more than once simultaneously, and might store data to return on each expiration of the timer.

reset (Context: `state_variables`) This option is used for advanced model checking. When set on a state variable, it declares that on the `maceReset` method call, this variable should not be reset, perhaps to mimic persistent state or to keep track of state across reboots in the simulated application.

notComparable (Context: `auto_types`) This option can be set on fields of `auto` types. When set, it omits them from the automatically generated comparison functions, which are generated when the `comparable` option is set on the `auto`-type.

4.3.2 Method

Method options can be passed into transitions, routines, or methods of an auto-type. The syntax is:

```
FunctionOptions : '[' (key) '=' (value) ( ';' (key) '=' (value) )* ']'
```

There are only three options presently supported:

merge (Context: transitions) This option tells the Mace compiler to merge this transition in with other transitions in the way specified. The two valid options are 'pre' and 'post'. Specifying the value 'pre' means this transition will be considered before the main dispatch of the transition, and 'post' means it will be considered after the main dispatch of the transition. Merged transitions are considered independently, unlike other transitions where only one may execute.

exec (Context: transitions) This option can take one value, 'once', which tells the Mace compiler that this transition should be executed at most once. It is equivalent to adding a guard that checks this same condition, and then sets a variable when the transition is in fact executed.

trace (Context: transitions, routines, auto_types) The trace option may be specified on transitions, routines, or auto-type methods. This option sets a routine-specific trace level, according to the trace-level semantics described earlier.

4.4 Programming Notes

Now that the Mace specification is fully described, we add a few brief programming notes. First, there are several macros defined in the `mace/lib/mace-macros.h` file that can be used somewhat 'magically' within a Mace specification. These include the logging macros, 'curtime', 'upcallAll', and 'upcallAllVoid'. The mace library is documented using doxygen-style comments. Doxygen allows generation of multiple formats of documentation by following the README in `mace/docs/doxygen/README`.

Without going into the definitions of these macros, a sample transition might look like this:

```

downcall maceInit() {
    maceout << "My service is now being initialized!" << Log::end;
    if (curtime % 2 == 0) {
        macedbg(0) << "The current time " << curtime <<
            " is an even number of microseconds" << Log::end;
    } else {
        ASSERT((curtime-1)%2 == 0);
    }
    upcallAllVoid(initDone, downcall_getLocalAddress());
}

```

In this illustrative initialization transition, the logging macros are being used. Each log message ends with a `Log::endl` object, which tells the logging system to emit the log message without waiting for more text. The `maceout` macro is used to emit a normal logging message, while the `macedbg` macro is being used to print a debug message at log level 0. The `curtime` macro is used multiple times, demonstrating its special property: it works like a time variable, but is only evaluated the first time it is called from any transition. It is only evaluated if needed though, and retains its value for the whole transition, reducing the overall number of calls to `gettimeofday`. Finally, instead of using the `assert` function from the `cassert` header file, the `ASSERT` macro is used, which prints error messages using a variety of mechanisms to effectively be supported by mace tools. `ASSERT` also ensures the error will be printed, and not ignored when logging is disabled. Finally, the `upcallAllVoid` macro is called to deliver the `initDone` upcall to all registered handlers, passing each the return value of the `downcall_getLocalAddress` method, being made to the lower level service. The macro `upcallAllVoid` is used when the upcall has a void return type, or if the return value will be ignored. A separate `upcallAll` macro can be used to aggregate return values across upcalls.

When the actual C++ is generated from this specification, the Mace compiler will insert `#line` directives to tell the C++ compiler where the original source file and line were in case of a compile error, making it easier to fix these problems.

Upon invocation, the Mace compiler needs to be told where to search for interface header files and included mace specification files. This is done through multiple `-I` options, as is done with the `g++` compiler. The build system that `cmake` uses to build Mace will automatically pass in the interfaces directory of the repository,

as well as the current directory. Additional include directories can be defined to tell it to search elsewhere. Finally, upon invoking the Mace compiler, if the environment variable `VERBOSE` is set to 1, it will output more information during the parse about progress and certain masked warnings. Note that setting `VERBOSE` to 1 will also cause the cmake build system to be more verbose.

4.5 Summary

The Mace language has been developed and grown to serve the needs of distributed systems designers, and we have learned many lessons along the way. Some parts of it remain complicated and not well fleshed out, which suggests that more opportunity remains to simplify the language and enhance it further. But already with these few building blocks, we are able to build substantially complicated systems quite simply, and without restricting ourselves to a small subset of distributed systems.

Chapter 5

Mace Modelchecker

Now that we’ve seen how to implement services in Mace, we next describe MaceMC, the Mace model checker, capable of finding liveness violations in unmodified Mace implementations. Hard-to-find, non-reproducible bugs have long been the bane of systems programmers. Such errors prove especially challenging in unreliable distributed environments with failures and asynchronous communication. For example, we have run our Mace implementation of the Pastry [RD01] overlay on the Internet and emulated environments for three years with occasional unexplained erroneous behavior: some nodes are unable to rejoin the overlay after restarting. Unable to recreate the behavior, we never succeeded in tracking down the cause of the error.

Motivated by this and similarly subtle bugs, we turned to model checking to assist us in building robust distributed systems. Unfortunately, existing model checkers able to run on systems implementations (rather than specifications) can only find *safety* violations—counterexamples of a specified condition that should always be true. Simple examples of safety properties are `assert()` statements and unhandled program exceptions. For our target systems however, specifying global *liveness* properties—conditions that should always *eventually* be true—proved to be more desirable. In the above example, we wished to verify that eventually all Pastry nodes would form a ring. Somewhat paradoxically, specifying the appropriate safety property requires knowledge of the nature of the bug, whereas specifying the appropriate liveness property only requires knowledge of desirable high-level system properties.

It is acceptable for a node to be unable to join a ring temporarily, but in our case, the bug made it impossible for a node to ever join the ring, thus violating liveness.

Existing software model checkers focus on safety properties because verifying liveness poses a far greater challenge: the model checker cannot know *when* the properties should be satisfied. Identifying a liveness violation requires finding an *infinite* execution that will not ever satisfy the liveness property, making it impractical to find such violating infinite executions in real implementations. Thus, we set out to develop practical heuristics that enable software model checkers to determine whether a system satisfies a set of liveness properties.

The Mace model checker, MaceMC, is the first software model checker that helps programmers find liveness violations in complex systems implementations. We built our solution upon three key insights:

Life: To find subtle, complicated bugs in distributed systems, we should search for liveness violations in addition to safety violations. Specifying liveness properties frees us from only specifying what ought not happen—that is, error conditions and invariants, which may be hopelessly complicated or simply unknown—and instead let us specify what ought to happen.

Death: Instead of searching for general liveness violations, which require finding violating infinite executions, we focus on a large subset: those that enter *dead* states from which liveness can never be achieved regardless of any subsequent actions. We thereby reduce the problem of determining liveness to searching for violations of previously unknown safety properties. We present a novel heuristic to identify dead states and locate executions leading to them by combining exhaustive search with long random executions.

Critical Transition: To understand and fix a liveness error, the developer must painstakingly analyze the tens of thousands of steps of the non-live execution to find where and how the system became dead. We show how to extend our random execution technique to automatically search for the *critical transition*, the step that irrecoverably cuts off all possibility of ever reaching a live state in the future.

To further help the programmer understand the cause of an error, we developed MDB, an interactive debugger providing forward and backward stepping through global events, per-node state inspection, and event graph visualization. In our experience, MDB, together with the critical transition automatically found by MaceMC, reduced the typical human time required to find and fix liveness violations from a few hours to less than 20 minutes.

Using MaceMC and MDB, we found the Pastry bug described at the beginning of this chapter: under certain circumstances, a node attempting to rejoin a Pastry ring using the same identifier was unable to join because its join messages were forwarded to unjoined nodes. This error was both sufficiently obscure and difficult to fix that we decided to check how FREEPASTRY[fre06], the reference implementation, dealt with this problem. The following log entry in a recent version of the code (1.4.3) suggests that FREEPASTRY likely observed a similar problem: “Dropped JoinRequest on rapid rejoin problem – There was a problem with nodes not being able to quickly rejoin if they used the same NodeId. Didn’t find the cause of this bug, but can no longer reproduce.”

We have catalogued more than 50 bugs using MaceMC thus far across a variety of complex systems, and MaceMC is now an integral part of the testing and development of new systems in Mace. The techniques developed for MaceMC are immediately available to any service implementation in Mace; however, we note that these techniques for finding liveness violations and the critical transition generalize to any state-exploration model checker capable of replaying executions. It should therefore be possible to use this technique with systems prepared for other model checkers by defining liveness properties for those systems. Although our approach to finding liveness violations is necessarily a heuristic—a proof of a liveness violation requires finding an infinite execution that never satisfies liveness—we have not had any false positives among the set of identified violations to date.

5.1 System Model

Software model checkers find errors by exploring the space of possible executions for systems implementations. We establish the MaceMC system model with

our simplified definitions of programs and properties (see [Kin94] for the classical definitions). We then discuss the relationship between liveness and safety properties.

5.1.1 Distributed Systems as Programs

We model-check distributed systems by composing every node and a simulated network environment in a single program (cf. §5.3.1 for the details of preparing unmodified systems for model checking). A program *state* is an assignment of values to variables. A *transition* matches the Mace definition of a transition or an event handler, and maps an input state to an output state. A *program* comprises a set of variables, a set of initial states, and a set of transitions. A *program execution* is an infinite sequence of states, beginning in an initial program state, with every subsequent state resulting from the application of some transition (an atomic set of machine instructions) to its predecessor. Intuitively, the set of variables corresponds to the *state-variables* of the services at each node together with the distributed environment, such as the messages in the network. Thus, a state encodes a snapshot of the entire distributed system at a given instant in time.

Conceptually, each node maintains a set of pending events. At each step in the execution, the model checker selects one of the nodes and an event pending at that node. The model checker then runs the appropriate event handler to transition the system to a new state. The handler may send messages that get added to event queues of destination nodes or schedule timers to add more events to its pending set. Upon completing an event handler, control returns to the model checker and we repeat the process. Each program execution corresponds to a scheduling of interleaved events and a sequence of transitions.

5.1.2 Properties

A *state predicate* is a logical predicate over the program variables or state-variables. Each state predicate evaluates to **TRUE** or **FALSE** in any given state. We say that a state *satisfies* (resp., *violates*) a state predicate if the predicate evaluates to **TRUE** (resp., **FALSE**) in the state.

Table 5.1: Example predicates from systems tested using MaceMC. *Eventually* refers here to *Always Eventually* corresponding to Liveness properties, and *Always* corresponds to Safety properties. The syntax allows a regular expression expansion ‘*’, used in the AllNodes property.

System	Name	Property
Pastry	AllNodes	<i>Eventually</i> $\forall n \in \mathbf{nodes} : n.(successor)^* \equiv \mathbf{nodes}$ Test that all nodes are reached by following successor pointers from each node.
	SizeMatch	<i>Always</i> $\forall n \in \mathbf{nodes} : n.myright.size() + n.myleft.size() = n.myleafset.size()$ Test the sanity of the leafset size compared to left and right set sizes.
Chord	AllNodes	<i>Eventually</i> $\forall n \in \mathbf{nodes} : n.(successor)^* \equiv \mathbf{nodes}$ Test that all nodes are reached by following successor pointers from each node.
	SuccPred	<i>Always</i> $\forall n \in \mathbf{nodes} : \{n.predecessor = n.me \iff n.successor = n.me\}$ Test that a node’s predecessor is itself if and only if its successor is itself.
RandTree	OneRoot	<i>Eventually</i> for exactly 1 $n \in \mathbf{nodes} : n.isRoot$ Test that exactly one node believes itself to be the root node.
	Timers	<i>Always</i> $\forall n \in \mathbf{nodes} : \{(n.state = init) \vee (n.recovery.nextScheduled() \neq 0)\}$ Test that either the node state is <i>init</i> , or the recovery timer is scheduled.
Mace-Transport	AllAked	<i>Eventually</i> $\forall n \in \mathbf{nodes} : n.inflightSize() = 0$ Test that no messages are in-flight (i.e., not acknowledged).
		No corresponding safety property identified.

Safety Property: a statement of the form *always p* where *p* is a *safety (state) predicate*. An execution *satisfies* a safety property if *every* state in the execution satisfies *p*. Conversely, an execution *violates* a safety property if *some* state in the execution violates *p*.

Liveness Property: a statement of the form *always eventually p* where *p* is a *liveness (state) predicate*. We define program states to be in exactly one of three categories with respect to a liveness property: *live*, *dead*, or *transient*. A live state satisfies *p*. A transient state does not satisfy *p*, but some execution through the state leads to a live state. A dead state does not satisfy *p*, and no execution through the state leads to a live state. An execution *satisfies* a liveness property if every suffix of the execution contains a live state. In other words, an execution satisfies the liveness property if the system enters a live state infinitely often during the execution. Conversely, an execution *violates* a liveness property if the execution has a suffix without any live states.

It is important to stress that liveness properties, unlike safety properties, apply over entire program executions rather than individual states. Classically, states cannot be called live (only executions)—we use the term live state for clarity. The intuition behind the definition of liveness properties is that any violation of a liveness state predicate should only be temporary: in any live execution, regardless of some violating states, there must be a future state in the execution satisfying the liveness predicate.

Table 5.1 shows example predicates from systems we have tested in MaceMC. We use the same liveness predicate for Pastry and Chord, as both form rings with successor pointers.

5.1.3 Liveness/Safety Duality

We divide executions violating liveness into two categories: Transient-state and Dead-state. *Transient-state (TS) liveness violations* correspond to executions with a suffix containing only transient states. For example, consider a system comprising two servers and a randomized job scheduling process. Let the liveness property be that eventually, the cumulative load should be balanced between the servers. In one TS liveness violation, the job scheduling process repeatedly prefers one server over the other. Along a resulting infinite execution, the cumulative load is never balanced. However, at every point along this execution, it is possible for the system to recover, e.g., the scheduler could have balanced the load by giving enough jobs to the underutilized server. Thus, all states in the violating execution are transient and the system never enters a dead state.

Dead-state (DS) liveness violations correspond to an execution with any dead state (by definition all states following a dead state must also be dead because recovery is impossible). Here, the violating execution takes a *critical transition* from the last transient (or live) state to the first dead state. For example, when checking an overlay tree (cf. §5.5), we found a violating execution of the “OneRoot” liveness state predicate in Table 5.1, in which two trees formed independently and never merged. The critical transition incorrectly left the *recovery* timer of a node *A* unscheduled in the presence of disjoint trees. Because only *A* had knowledge of members in the other tree, the protocol had no means to recover.

Our work focuses on finding DS liveness violations. We could have found these violations by using safety properties specifying that the system never enters the corresponding dead states. Unfortunately, these safety properties are often impossible to identify *a priori*. For instance, consider the liveness property “AllNodes” for Chord shown in Table 5.1: eventually, all nodes should be reachable by following successor pointers. We found a violation of this property caused by our failure to maintain the invariant that in a one-node ring, a node’s predecessor and successor should be itself. Upon finding this error, we added the corresponding safety property for Chord. While we now see this as an “obvious” safety property, we argue that exhaustively listing all such safety properties *a priori* is much more difficult than specifying desirable liveness properties.

Moreover, liveness properties can identify errors that in practice are infeasible to find using safety properties. Consider the “AllAked” property for our implementation of a transport protocol, shown in Table 5.1. The property is for the test application, which sends a configurable total number of messages to a destination. It states that all sent messages should eventually be acknowledged by the destination (assuming no permanent failures): the transport adds a message to the *inflight* queue upon sending and removes it when it is acknowledged. The corresponding safety property would have to capture the following: “Always, for each message in the *inflight* queue or retransmission timer queue, either the message is in flight (in the network), or in the destination’s receive socket buffer, or the receiver’s corresponding *IncomingConnection.next* is less than the message sequence number, or an acknowledgment is in flight from the destination to the sender with a sequence number greater than or equal to the message sequence number, or the same acknowledgment is in the sender’s receive socket buffer, or a reset message is in flight between the sender and receiver (in either direction), or ...” Thus, attempting to specify certain conditions with safety properties quickly becomes overwhelming and hopelessly complicated, especially when contrasted with the simplicity and succinctness of the liveness property: “Eventually, for all n in nodes, $n.inflightSize() = 0$,” i.e., that eventually there should be no packets in flight.

Thus, we recommend (and ourselves practice) the following iterative process for finding subtle protocol errors in complex concurrent environments. A developer

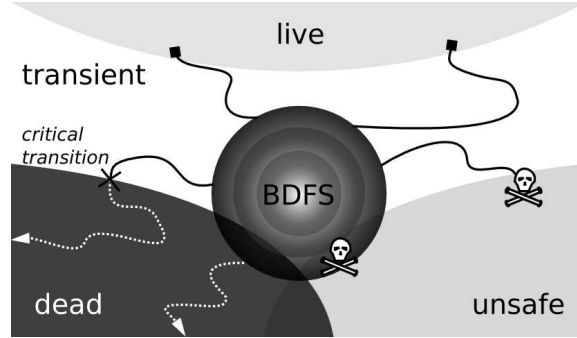


Figure 5.1: State exploration: we perform an iterative, bounded depth-first search (BDFS) from the initial state (or search prefix): most periphery states are indeterminate. We execute random walks from the periphery states and flag walks not reaching live states as suspected violating executions.

begins by writing desirable high-level liveness properties. As these liveness properties typically define the correct system behavior in steady-state operation, they are relatively easy to specify. Developers can then leverage insight from DS liveness violations to add new safety properties. In Table 5.1, we show safety properties that became apparent while analyzing the corresponding DS liveness violations. While safety properties are often less intuitive, the errors they catch are typically easier to understand—the bugs usually do not involve complex global state and lie close to the operations that trigger the violations.

5.2 Model Checking with MaceMC

This section presents our algorithms for finding liveness and safety violations in systems implementations. We find potential liveness violations via a three-step state exploration process. While our techniques do not present proofs for the existence of a liveness violation, we have thus far observed no false positives. In practice, all flagged violations must be human-verified, which is reasonable since they point to bugs which must be fixed. As shown in Figure 5.1, our process isolates executions leading the system to dead states where recovery to a configuration satisfying the liveness state predicate becomes impossible.

Step 1: Bounded depth-first search (BDFS) We begin by searching from an initial state with a bounded depth-first search. We exhaustively explore all executions up to some fixed depth in a depth-first manner and then repeat with an increased depth bound. Due to state explosion, we can only exhaustively explore up to a relatively shallow depth of transitions (on the order of 25-50); as system initialization typically takes many more transitions (cf. Figure 5.3), the vast majority of states reached at the periphery of the exhaustive search are not live. We call these states *indeterminate* because at this point we do not yet know whether they are dead or transient.

Step 2: Random Walks While the exhaustive search is essential to finding a candidate set of liveness violations, to prune the false positives, we must distinguish the dead from the transient states. To do so, we perform long random walks to give the system sufficient time to enter a live state. If the system still fails to reach a live state by the end of the walk, we flag the execution as a suspected liveness violation. Our random walks typically span tens or hundreds of thousands of transitions to minimize the likelihood of false positives.

Step 3: Isolating the Critical Transition The model checker presents the execution exhibiting a suspected liveness violation to the developer to assist in locating the actual error. The programmer cannot understand the bug simply by examining the first states that are not live, as these are almost always transient states, i.e., there exist executions that would transition these initial indeterminate states to live states. Thus, we developed an algorithm to automatically isolate the *critical transition* that irreversibly moves the system from a transient state to a dead state.

5.2.1 Finding Violating Executions

We now describe the details of our algorithms. Suppose that MaceMC is given a system, a safety property *always* p_s , and a liveness property *eventually* p_l .

Our algorithm `MaceMC_Search` (Algorithm 1) systematically explores the space of possible executions. Each execution is characterized by the sequence of choices made to determine the node-event pair to be executed at each step. We iterate over all the sequences of choices of some fixed length and explore the states visited in the

Algorithm 1 MaceMC_Search

Input: Depth *increment*

```

depth = 0
repeat
  if Sequences(depth) is empty then
    depth = depth + increment
  Reset system
  seq = next sequence in Sequences(depth)
  MaceMC_Simulator(seq)
until STOPPING CONDITION

```

execution resulting from the sequence of choices. Consider the set of all executions bounded to a given depth *depth*. These executions form a tree by branching whenever one execution makes a different choice from another. To determine the order of executions, we simply perform a depth-first traversal over the tree formed by this depth bound. *Sequences(depth)* returns a sequence of integers indicating which child to follow in the tree during the execution. It starts by returning a sequence of 0's, and each time it is called it increases the sequence, searching all possible sequences. For each sequence, **MaceMC_Search** initializes the system by resetting the values of all nodes' variables to their initial values and then calls the procedure **MaceMC_Simulator** to explore the states visited along the execution corresponding to the sequence. After searching all sequences of length *depth*, we repeat with sequences of increasing depth. We cannot search extreme system depths due to the exponential growth in state space. While they have not been necessary to date, optimizations such as multiple random walks or best-first search may enhance coverage over initial system states.

Algorithm 2, **MaceMC_Simulator**, takes a sequence of integers as input and simulates the resulting execution using the sequence of choices corresponding to the integers. **MaceMC_Simulator** simulates an execution of up to d_{max} transitions (cf. §5.3.4 for setting d_{max}). At the i^{th} step, **MaceMC_Simulator** calls the procedure **Toss** with i , the sequence, and the number of ready events to determine pending node event pairs to execute, and then executes the handler for the chosen event on the chosen node to obtain the state reached after i transitions. If this state violates the given

Algorithm 2 MaceMC_Simulator

Input: Sequence seq of integers

```

for  $i = 0$  to  $d_{max}$  do
   $readyEvents = \text{set of pending } \langle node, event \rangle \text{ pairs}$ 
   $eventnum = \text{Toss}(i, seq, |readyEvents|)$ 
   $\langle node, event \rangle = readyEvents[eventnum]$ 
  Simulate  $event$  on  $node$ 
  if  $p_s$  is violated then
    signal SAFETY VIOLATION
  if  $i > depth$  and  $p_l$  is satisfied then
    return
  signal SUSPECTED LIVENESS VIOLATION

```

safety predicate, then **MaceMC_Simulator** reports the safety violation. If this state is beyond the search depth and satisfies the given liveness predicate, then the execution has not violated the liveness property and the algorithm returns. Only considering liveness for states beyond the search depth is required to test that *every suffix* of the execution contain a live state. Otherwise, we would be testing that for any execution, a single live state must be reached. If the loop terminates after d_{max} steps, then we return the execution as a suspected liveness violation.

Combining Exhaustive Search and Random Walks

The procedure **Toss** ensures that **MaceMC_Search** and **MaceMC_Simulator** together have the effect of exhaustively searching all executions of bounded depths and then performing random walks from the periphery of the states reached in the exhaustive search. **Toss**(i, seq, k) returns the i^{th} element of the sequence seq if i is less than $|seq|$ (the length of the sequence) or some random number between 0 and k otherwise. Thus, for the first $|seq|$ iterations, **MaceMC_Simulator** selects the $seq[i]^{th}$ element of the set of pending node event pairs, thereby ensuring that we exhaustively search the space of all executions of depth $|seq|$. Upon reaching the end of the supplied sequence, the execution corresponds to a random walk of length $d_{max} - |seq|$ performed from the periphery of the exhaustive search. By ensuring d_{max} is large enough (hundreds

of thousands of transitions), we can give the system enough opportunity to reach a live state. If the execution never enters a live state despite this opportunity, we flag the execution as a suspected liveness violation.

5.2.2 Finding the Critical Transition

If MaceMC reaches the maximum random walk depth d_{max} without entering a live state, we have a suspected liveness violation. The execution meets one of two conditions:

Condition 1 (C1): The execution is a DS liveness violation, meaning the system will never recover. The execution should be brought to the attention of the programmer to locate and fix the error.

Condition 2 (C2): The execution does not reach any live states, but might still in the future. The execution should be brought to the attention of the programmer to determine whether to proceed by increasing d_{max} or by inspecting the execution for a bug.

Before discussing how we distinguish between the two cases, consider an execution that does enter a dead state (meets condition C1). The programmer now faces the daunting and time consuming task of wading through tens of thousands of events to isolate the protocol or implementation error that transitioned the system to a dead state. Recall that while the system may enter a transient state early, typically a much later critical transition finally pushes the system into a dead state. After attempting to find liveness errors manually when only the violating execution was available, we set out to develop an algorithm to automatically locate the critical transition. Importantly, this same procedure also heuristically identifies whether an execution meets C1 or C2.

Algorithm 3 shows our two-phase method for locating the critical transition. It takes as input the execution E from the initial random walk, which from step d_{init} onwards never reached a live state even after executing to the maximum depth d_{max} . The function `Recovers(E, i, k)` performs up to k random walks starting from the i^{th} state on the execution E to the depth d_{max} and returns `TRUE` if *any* of these walks

Algorithm 3 FindCriticalTransition

Input: Execution E non-live from step d_{init} to d_{max}
Input: Number of Random Walks k
Output: (Critical Transition d_{crit} , Condition C1 or C2)

```

1: {Phase 1: Exponential Search}
2: if not Recovers( $E, d_{init}, k$ ) then return ( $d_{init}, C2$ )
3:  $d_{curr} = d_{init}$ 
4: repeat
5:    $d_{prev} = d_{curr}$ 
6:    $d_{curr} = 2 \times d_{curr}$ 
7:   if  $d_{curr} > d_{max}/2$  then return ( $d_{curr}, C2$ )
8: until not Recovers( $E, d_{curr}, k$ )
9: {Phase 2: Binary Search}
10: { $d_{prev}$  is highest known recoverable}
11: { $d_{curr}$  is lowest believed irrecoverable}
12: loop
13:   if ( $d_{prev} = d_{curr} - 1$ ) then return ( $d_{curr}, C1$ )
14:    $d_{mid} = (d_{prev} + d_{curr})/2$ 
15:   if Recovers( $E, d_{mid}, k$ ) then  $d_{prev} = d_{mid}$ 
16:   else  $d_{curr} = d_{mid}$ 

```

hit a live state, indicating that the i^{th} state should be marked transient; and **FALSE** otherwise, indicating that the i^{th} state is dead. In the first phase, MaceMC doubles d_{curr} until **Recovers** indicates that d_{curr} is dead. d_{max} and the resulting d_{curr} place an upper bound on the critical transition, and the known live state d_{prev} serves as a lower bound. In the second phase, MaceMC performs a binary search using **Recovers** to find the critical transition as the *first* dead state d_{crit} between d_{prev} and d_{curr} . If we perform k random walks from each state along the execution, then the above procedure takes $O(k \cdot d_{max} \cdot \log d_{crit})$ time (Note that $d_{crit} \leq d_{max}$).

In addition to the full execution that left the system in a dead state and the critical transition d_{crit} , we also present to the programmer the event sequence that shares the longest common prefix with the DS liveness violation that ended in a

live state. In our experience, the combination of knowing the critical transition and comparing it to a similar execution that achieves liveness is invaluable in finding the actual error.

Two interesting corner cases arise in the `FindCriticalTransition` algorithm. The first case occurs when Phase 1 cannot locate a dead state (indicated by $d_{curr} > d_{max}/2$ in line 7). In this case, we conclude that as the critical transition does not appear early enough, the system was not given enough opportunity to recover during the random walk. Thus, case C2 holds. The developer should raise d_{max} and repeat. If raising d_{max} does not resolve the problem, the developer should consider the possibility that this execution is a TS liveness violation. To help this analysis, MaceMC provides the set of live executions similar to the violating execution, but the developer must isolate the problem. In the second case, we find no live executions even when in the initial state (line 2); either the critical transition is at d_{init} (the initial state), or, more likely, we did not set d_{max} high enough. The programmer can typically determine with ease whether the system condition at d_{init} contains a bug. If not, once again we conclude that case C2 holds and raise d_{max} and repeat Algorithm 1.

5.3 Implementation Details

This section describes several subtle details in our MaceMC implementation. While we believe the techniques described in Section 5.2 could be applied to any state-exploration model checker capable of replaying executions, MaceMC operates on systems implemented using the Mace compiler and C++ language extensions described in § 3. Leveraging Mace code frees us from the laborious task of modifying source code to isolate the execution of the system, e.g., to control network communication events, time, and other sources of potential input. Thus, using Mace-implemented systems dramatically improves the accessibility of model checking to the typical programmer.

5.3.1 Preparing the System

To model check a system, the user writes a driver application suitable for model checking that should initialize the system, perform desired system input events, and

check high-level system progress with liveness properties. For example, to look for bugs in a file distribution protocol, the test driver could have one node supply the file, and the remaining nodes request the file. The liveness property would then require that all nodes have received the file and the file contents match. Or for a consensus protocol, a simulated driver could propose a different value from each node, and the liveness property would be that each node eventually chooses a value and that all chosen values match. In the case of MaceMC, the simulated application is implemented as a C++ object which provides two methods, one which returns whether an application event is waiting, and another which simulates the given application event. It is generally implemented within Mace as a service which provides the `SimApplication` service class. The user also must implement a function which configures all the nodes in the system, each of which may execute different code. The MaceMC application links with the simulated driver, the user’s compiled Mace object files, and Mace libraries. MaceMC simulates a distributed environment to execute the system—loading different simulator-specific libraries for random number generation, timer scheduling, and message transport—to explore a variety of event orderings for a particular system state and input condition.

Non-determinism

To exhaustively and correctly explore different event orderings of the system, we must ensure that the model checker controls all sources of non-determinism. So far, we have assumed that the scheduling of pending $\langle \text{node}, \text{event} \rangle$ pairs accounts for all non-determinism, but real systems often exhibit non-determinism *within* the event handlers themselves, due to, e.g., randomized algorithms and comparing timestamps. When being model checked, Mace systems automatically use the deterministic simulated random number generator provided by MaceMC and the support for simulated time, which we discuss below. Furthermore, we use special implementations of the Mace libraries that internally call `Toss` at every non-deterministic choice point. For example, the TCP transport service uses `Toss` to decide whether to break a socket connection, the UDP transport service uses `Toss` to determine which message to deliver (allowing out-of-order messages) and when to drop messages, and the application simulator uses `Toss` to determine whether to reset a node. Thus, by systematically

exploring the *sequences* of return values of `Toss` (as described in `MaceMC_Search` in the previous section), MaceMC analyzes all different sequences of internal non-deterministic choices. Additionally, this allows MaceMC to deterministically replay executions for a given sequence of choices.

One special note about non-determinism is that certain forms of it are outside the control of MaceMC, and must be avoided if model checking is to be used. One example of this is uninitialized stack variables. Since these may contain different values each time the method is called, this will cause non-determinism leading the executions to behave differently, preventing the model checker from properly detecting problems. Other things which would cause this include a wide variety of memory bugs as well as using the c-library directly to access time and random number functions. The model checker will detect non-determinism if different calls are made to the random number generator, and will report it to the user (though debugging it is challenging).

Time

Time introduces non-determinism, resulting in executions that may not be replayable or, worse, impossible in practice. For example, a system may branch based on the relative value of timestamps (e.g., for message timeout). But if the model checker were to use actual values of time returned by `gettimeofday()`, this comparison might always be forced along one branch as the simulator fires events faster than a live execution. Thus, MaceMC must represent time abstractly enough to permit exhaustive exploration, yet concretely enough to only explore feasible executions. In addition, MaceMC requires that executions be deterministically replayable by supplying an identical sequence of chosen numbers for all non-deterministic operations, including calls to `gettimeofday`.

The options of returning a random, constant or monotonically increasing value for all calls to `gettimeofday` would result in both the incorrect pruning of valid executions (thus missing bugs), and the unnecessary exploration of executions that cannot occur in practice. The option of replacing all calls to `gettimeofday` with a call to `Toss` to generate an arbitrary 64-bit integer would result in an explosion in the space of sequences to be searched, rendering MaceMC useless for finding bugs.

We observed that systems tend to use time to: (i) manage the passage of real time, e.g., to compare two timestamps when deciding whether a timeout should occur, or, (ii) export the equivalent of monotonically increasing sequence numbers, e.g., to uniquely order a single node’s messages. Therefore, we address the problem of managing time by introducing two new Mace object primitives—**MaceTime** and **MonotoneTime**—to obtain and compare time values. When running across a real network, both objects are wrappers around `gettimeofday`. However, MaceMC treats every comparison between **MaceTime** objects as a call to **Toss** and implements **MonotoneTime** objects with per-node counters. Developers concerned with negative clock adjustments (and more generally non-monotone **MonotoneTime** implementations) can strictly use **MaceTime** to avoid missing bugs, at the cost of extra states to explore. Compared to state of the art model checkers, this approach frees developers from manually replacing time-based non-determinism with calls to **Toss**, while limiting the amount of needless non-determinism.

5.3.2 Mitigating State Explosion

One stumbling block for model-checking systems is the exponential explosion of the state space as the search depth increases. MaceMC mitigates this problem using four techniques to find bugs deep in the search space.

1. Structured Transitions

The event-driven, non-blocking nature of Mace code significantly simplifies the task of model-checking Mace implementations and improves its effectiveness. In the worst case, a model checker would have to check all possible orderings of the assembler instructions across nodes with pending events, which would make it impractical to explore more than a few hundred lines of code across a small number of nodes. Model checkers must develop techniques for identifying larger atomic steps. Some use manual marking, while others interpose communication primitives. Non-blocking, atomic event handlers in Mace allow us to use event-handler code blocks as the fundamental unit of execution. Once a given code block runs to completion, we return control

to MaceMC. At this point, MaceMC checks for violations of any safety or liveness conditions based on global system state.

2. State Hashing

When the code associated with a particular event handler completes without a violation, MaceMC calculates a hash of the resulting system state. This state consists of the concatenation of the values of all per-node state variables and the contents of all pending, system-wide events. The programmer may optionally annotate Mace code to ignore the value of state variables believed to not contribute meaningfully to the uniqueness of global system state, or to format the string representation into a canonical form to avoid unneeded state explosion (such as the order of elements in a set). `MaceMC.Simulator` checks the hash of a newly-entered state against all previous state hashes. When it finds a duplicate hash, MaceMC breaks out of the current execution and begins the next sequence. In our experience, this allows MaceMC to avoid long random walks for 50-90 percent of all executions, yielding speedups of 2-10.

3. Stateless Search

MaceMC performs backtracking by re-executing the system from an initial state, following the sequence of choices used to reach an earlier state, similar to the approach taken by Verisoft [God97]. For example, to backtrack from the system state characterized by the sequence $\langle 0, 4, 0 \rangle$ to a subsequent system state characterized by choosing the sequence $\langle 0, 4, 1 \rangle$, MaceMC reruns the system from its initial state, re-executing the event handlers that correspond to choosing events 0 and 4 before moving to a different portion of the state space by choosing the event associated with value 1. This approach is simple to implement and does not require storing all of the necessary state (stack, heap, registers) to restore the program to an intermediate state. However, it incurs additional CPU overhead to re-execute system states previously explored. We have found trading additional CPU for memory in this manner to be reasonable because CPU time has not proven to be a limitation in isolating bugs for MaceMC. However, the stateless approach is not fundamental to MaceMC—we are presently exploring hybrid approaches that involve storing some state such as

sequences for best-first searching or state for checkpointing and restoring system states to save CPU time.

4. Prefix-based Search

Searching from an initial global state suffers the drawback of not reaching significantly past initialization for the distributed systems we consider. Further, failures during the initial join phase do not have the opportunity to exercise code paths dealing with failures in normal operation because they simply look like an aborted join attempt (e.g., resulting from dropped messages) followed by a retry. To find violations in steady-state system operation, we run MaceMC to output a number of live executions of sufficient length, i.e., executions where all liveness conditions have been satisfied, all nodes have joined, and the system has entered steady-state operation. We then proceed as normal from one of these live prefixes with exhaustive searches for safety violations followed by random walks from the perimeter to isolate and verify liveness violations. We found the Pastry bug described in the introduction using a prefix-based search.

5.3.3 Biasing Random Walks

We found that choosing among the set of all possible actions with equal probability had two undesirable consequences. First, the returned random walks on error paths had unlikely event sequences that obfuscated the real cause of the violation. For example, the system generated a sequence where the same timer fired seven times in a row with no intervening events, which would be unlikely in reality. Second, these unlikely sequences slowed system progress, requiring longer random walks to reach a live state. Setting d_{max} large enough to ensure that we had allowed enough time to reach live states slowed `FindCriticalTransition` by at least a factor of ten.

We therefore modified `Toss` to take a set of weights corresponding to the rough likelihood of each event occurring in practice. `Toss` returns an event chosen randomly with the corresponding probabilities. For example, we may prioritize application events higher than message arrivals, and message arrivals higher than timers firing. In this way, we *bias* the system to search event sequences in the random walk with

the hope of reaching a live state sooner, if possible, and making the error paths easier to understand.

Creating the weighted version of `Toss` had another side effect—making the state space smaller, while retaining application logic. Consider a system which has as logic “70 percent of the time do X ”. The application logic requires this to be coded as `if(Toss(10) < 7)X`. But this would present MaceMC with 10 options. The weighted version instead presents MaceMC with only 2 options, but tells it what weight they shall occur in real execution (and therefore random walks). The weighted version of `Toss` can also be used in Mace services by developers to inject random actions with weight 0, meaning they would only occur during exhaustive search, and not during random walks. This is used to prevent timeouts in testing a user-level transport.

Figure 5.2 shows that the average depth before reaching a live state from an unmarked state using unbiased random walks is larger than appropriately chosen biased walks. This figure plots the cumulative distribution of steps to live paths for a simple 2-node overlay tree (see § 5.5 for protocol details).

Biasing the random walks to common sequences may run counter to the intuition that model checkers should push the system into corner conditions difficult to predict or reason about. However, recall that we run random walks only after performing exhaustive searches to a certain depth. Thus, the states reached by the periphery of the exhaustive search encompass many of these tricky corner cases, and the system has already started on a path leading to—or has even entered—a dead state. Other mechanisms may be used to push the system into bad states, but the biased random walks are specifically to test that a dead state has not been entered.

One downside to this approach is that the programmer must set the relative weights for different types of events. In our experience, however, every event has had a straightforward rough relative probability weighting. Further, the reductions in average depth before transitioning to a live state and the ease of understanding the violating executions returned by MaceMC have been worthwhile. If setting the weights proves challenging for a particular system, MaceMC can be run with unbiased random walks.

In one special case, an application wished to set different weights for different application events, and so the `eventsWaiting` call was modified to allow setting the

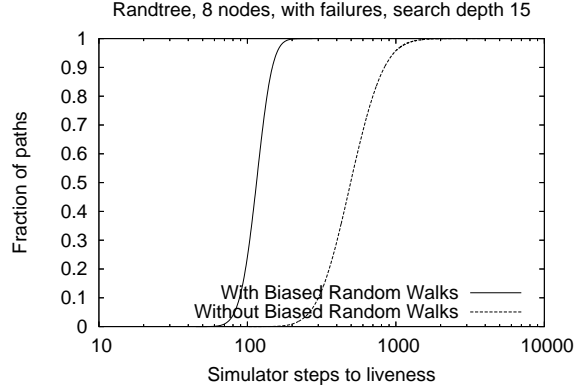


Figure 5.2: Compared performance of random walks with and without biasing.

event weight. This was to bias the walk away from restarting a node after a reboot. At first glance, an application could just return that no events were waiting for some period of time. However this does not work (and should not be used in practice), because if no *other* events are waiting, the path will abruptly end. Rather, set a low event weight, and restarting will simply be less likely.

We admit that a particular weight assignment may in fact bias the simulator away from a sequence of events, however unlikely, that would be able to transition the system back to a live state. In this case, MaceMC would report false positives because it would confirm the presence of a liveness violation that could have been invalidated by pure random walks. However, by setting d_{max} to be large enough, these false positives can be eliminated, and in practice, have never been encountered.

Fairness

Liveness properties are traditionally checked over fair system executions. For example, in a messaging system where we wish to check that all messages are eventually acknowledged, the liveness property may be violated along an execution where the receiving node is never scheduled and thus may never send its ACK, or along another execution where the network drops every ACK packet. Classically [CGP99], the user prohibits the model checker from considering such executions by specifying *fairness constraints* that force the model checker to only consider those executions

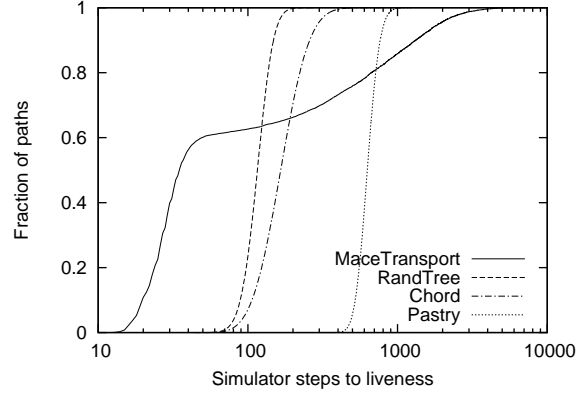


Figure 5.3: CDF of simulator steps to a live state at a search depth of 15.

where each node gets to execute infinitely often and messages are not dropped infinitely often. Unfortunately, these constraints are hard to specify and enforce for real systems implementations. Our random-walk-based approach eliminates the need for explicitly specifying the fairness constraints but nevertheless, with high probability, restricts the checker to consider only those executions that meet the fairness constraints. We intuitively achieve this by running the walk long enough to ensure that the probability of an unfair execution drops to zero, as over time all possible transitions get selected often enough. As a result, the random walks ensure that MaceMC finds violations that satisfy the implicit fairness constraints and thus correspond to genuine liveness bugs.

5.3.4 Tuning MaceMC

In addition to event weights discussed above, MaceMC may be tuned by setting d_{max} (random walk depth), k (number of random walks), and a wide variety of knobs turning features on and off. Feature knobs include whether to test node failures, socket failures, UDP drops, UDP reordering, and the number of simulated nodes, and are generally easy to set based on the target test environment.

Setting k is a bit more complex. k represents the tradeoff between the time to complete the critical transition algorithm and the possibility that the reported critical transition is before the actual critical transition. This occurs when k random

executions of d_{max} steps did not satisfy liveness, but some other path could have. We informally refer to this occurrence as “near dead”. In our tests, we generally use k between 20 and 60. At 60, we have not observed any prematurely reported critical transitions, while at 20 we occasionally observe the reported critical transition off by up to 2 steps. To tune k , the programmer considers the output critical transition. If it is not obvious why it is the critical transition, the programmer can increase k and re-run to refine the results.

Finally, we discuss how to set d_{max} . We ran MaceMC over four systems using random walks to sample the state space beyond an exhaustive search to 15 steps. Figure 5.3 plots the fraction of executions that reached the first live state at a given depth. What we observe is that in these four systems, since all sample executions reached a live state by 10,000 steps, a random execution that takes 80,000 steps to reach a live state would be a significant outlier, and likely somewhere along the execution it became trapped in a region of dead states. Setting d_{max} too low generally leads to the critical transition algorithm reporting condition C2, which is what we treat as the signal to increase d_{max} .

Figure 5.3 also illustrates that the depths required to initially reach a live state are much greater than what can be found with exhaustive search. MaceMC found only 60% of executions reached a live state for MaceTransport after considering 50 steps (the edge of what can be exhaustively searched using state-of-the-art model checkers), less than 1% of executions for RandTree and Chord, and none of the executions for PASTRY.

5.3.5 Diverging Event Handlers

Inside `MaceMC_Simulator`, we have assumed that each event handler terminates and returns control to the outer loop so that the next node event pair can be executed. However, a buggy Mace event handler may not terminate and return control to the simulator, as it may, for example, deadlock or enter an infinite loop. Inside the procedure `MaceMC_Simulator`, we set a separate timer before invoking each event handler. If the timer fires before the event handler completes, we signal a *divergence*, a liveness violation where the system enters a state where it cannot make forward

progress because a single node remains stuck in an event handler. Though a divergence could just be due to a complex computation or blocking call that took too long, we still report a bug since experience with Mace has shown that events lasting that long can have a significant impact on the throughput of the system. However, since there could be a system which makes use of such code, MaceMC can be configured to use different timeout intervals, or ignore the divergence timer altogether.

5.4 MaceMC Debugger

Although MaceMC flags violating executions and identifies the critical transition that likely led the system to a dead state, the developer must still understand the sequence of events to determine the root cause of the error. This process typically involves manually inspecting the log files and hand-drawing sketches of evolving system state. To simplify this process, we built MDB, our debugging tool with support for interactive execution, replay, log analysis, and visualization of system state across individual nodes and transitions. MDB is similar in function to other work in distributed debuggers such as the WiDS Checker [LLPZ07] and Friday [GAM⁺07]. MDB allows the programmer to: (i) perform single step system execution both forward and backward, (ii) jump to a particular step, (iii) branch execution from a step to explore a different path, (iv) run to liveness, (v) select a specific node and step through events only for that node, (vi) list all the steps where a particular event occurred, (vii) filter the log using regular expressions, and (viii) *diff* the states between two steps or the same step across different executions by comparing against a second, similar log file.

MDB also generates event graphs that depict inter-node communication. It orders the graph by nodes on the x-axis and simulator steps on the y-axis. Each entry in the graph describes a simulated event, including the transition call stack and all message fields. Directional arrows represent message transmissions, and other visual cues highlight dropped messages, node failures, etc.

MDB recreates the system state by analyzing detailed log files produced by MaceMC. While searching for violations, MaceMC runs with all system logging disabled for maximum efficiency. Upon discovering a violation, MaceMC automatically replays the path with full logging. The resulting log consists of annotations: (i) writ-

```

$ ./mdb error.log
(mdb 0) j 5
(mdb 5) filediff live.log
...
localaddress=2.0.0.1:10201
out=[
-   OutgoingConnection(1.0.0.1:10201, connection=ConnectionInfo(cwnd=2, packetsSent=2, ack-
sReceived=1, packetsRetransmitted=0),
-       inflight=[ 6002 → MessageInfo(seq=6002, syn=0, retries=0, timeout=true) ],
-       rtbuf=[ ], sendbuf=[ ], curseq=6002, dupacks=0, last=6001)
+   OutgoingConnection(1.0.0.1:10201, connection=ConnectionInfo(cwnd=1, packetsSent=1, ack-
sReceived=0, packetsRetransmitted=0),
+       inflight=[ 6001 → MessageInfo(seq=6001, syn=1, retries=0, timeout=true) ],
+       rtbuf=[ ], sendbuf=[ MessageInfo(seq=6002, syn=0, timer=0, retries=0, time-
out=true) ], curseq=6002, dupacks=0, last=0)
]
in=[ ]
- timer<retransmissionTimer>([dest=1.0.0.1:10201, msg=MessageInfo(seq=6002, syn=0, re-
tries=0, timeout=true)])
+ timer<retransmissionTimer>([dest=1.0.0.1:10201, msg=MessageInfo(seq=6001, syn=1, re-
tries=0, timeout=true)])
...

```

Figure 5.4: MDB session. Lines with differences are shown in italics (– indicates the error log, + the live log), with differing text shown in bold. The receiver is IP address 1.0.0.1 and the sender is 2.0.0.1.

ten by the programmer, (ii) generated automatically by the Mace compiler marking the beginning and end of each transition, (iii) produced by the simulator runtime libraries, such as timer scheduling and message queuing and delivery, and (iv) generated by the simulator to track the progress of the run, including random number requests and results, the node simulated at each step, and the state of the entire system after each step. For our runs, logs can span millions of entries (hundreds to thousands of megabytes).

To demonstrate the utility of our debugging tools for diagnosing and fixing errors, we consider a case study with a bug in MACETRANSPORT: a reliable, in-order, message delivery transport with duplicate-suppression and TCP-friendly congestion-

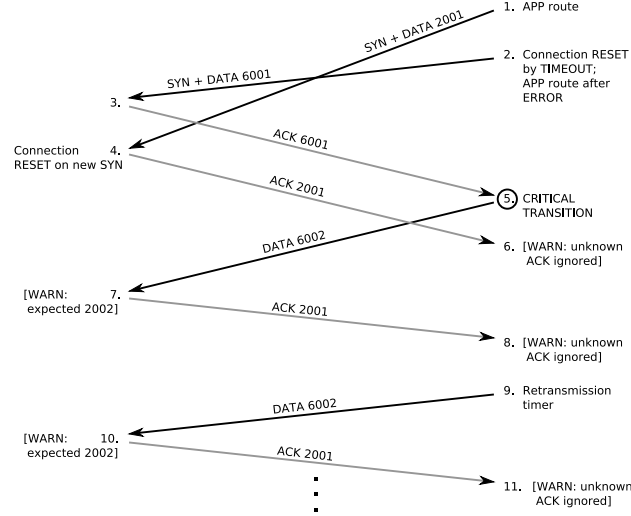


Figure 5.5: Automatically generated event graph for MACETRANSPORT liveness bug.

control built over UDP. Unlike TCP, MACETRANSPORT is fundamentally message- rather than stream-oriented, making it a better match for certain higher-level application semantics. As such, rather than using sequence numbers to denote byte offsets as with TCP, MaceTransport assigns an incrementing sequence number to each packet. To obtain lower-latency communication, MACETRANSPORT avoids a three-way handshake to establish initial sequence numbers. A key high-level liveness property for MACETRANSPORT is that eventually every message should be acknowledged (unless the connection closes).

MaceMC found a violating execution of the “AllAked” property in Table 5.1, where a sender attempts to send two messages to a receiver. Figure 5.5 shows a pictorial version of the event graphs automatically generated by MDB; the actual event graph is text-based for convenience and contains more detail. In Step 1, the sender sends a data packet with the SYN flag set and sequence number 2001. In Step 2, the retransmission timer causes the connection to close and MACETRANSPORT signals an error to the application. The application responds by attempting to resend the packet, causing MACETRANSPORT to open a new connection with sequence number 6001. At this point, both the old “SYN 2001” and the new “SYN 6001” packets are in flight. In Step 3, the network delivers the packet for the new 6001 connection, and the receiver replies by sending an “ACK 6001” message. In Step 4, the network

delivers the out-of-order “SYN 2001” message, and the receiver responds by closing the connection on 6001, thinking it is stale, and opening a new incoming connection for 2001.

Unfortunately, in Step 5 (the critical transition) the sender receives the message “ACK 6001.” Believing the 6000-sequence connection to be established, the sender transmits “DATA 6002,” at odds with the receiver’s view. From here on, the execution states are dead as the receiver keeps ignoring the “DATA 6002” packet, sending ACKs for the 2001 connection instead, while the sender continues to retransmit the “DATA 6002” packet, believing it to be the sequence number for the established connection.

We illustrate a portion of an MDB session analyzing this bug in Figure 5.4. We load the error log in MDB, jump to the critical transition step (5), and *diff* the state with the live path with the longest shared prefix (output by MaceMC while searching for the critical transition (see §5.2.2)). The excerpt shows the state for the sender node. The key insight from this output is that in the live execution (lines indicated with +), the retransmission timer is scheduled with “SYN 6001,” meaning that the packet could be retransmitted and the receiver could become resynchronized with the sender. Comparing the differences with the violating execution (lines indicated with −), where 6001 has been removed from the *inflight* map and timer because of the ACK, allows us to identify and fix the bug by attaching a monotonically increasing identifier in the SYN packets, implemented using a `MonotoneTime` object. Now, when the receiver gets the “SYN 2001” message out of order, it correctly concludes from the identifier that the message is stale and should be ignored, allowing acknowledgment of the “DATA 6002” message.

5.5 Experiences

We have used MaceMC to find and catalog safety and liveness bugs in a variety of systems implemented in Mace, including a reliable transport protocol, an overlay tree, Pastry, and Chord. Now, MaceMC is an integral part of our implementation and testing, and we use it daily to debug implementations of Paxos, a distributed shared memory service, a community information synchronization service, and all

our implementations. For the reliable transport protocol, overlay tree, and Pastry, we ran MaceMC over mature implementations manually debugged both in local- and wide-area settings. MaceMC found several subtle bugs in each system that caused violations of high-level liveness properties. All violations cataloged (save some found in Chord, see below) were beyond the scope of existing software model checkers because the errors manifested themselves at depths far beyond what can be exhaustively searched. We used the debugging process with Chord as control—we first performed manual debugging of a new implementation of Chord and then employed MaceMC to compare the set of bugs found through manual and automated debugging.

Table 5.2 summarizes the bugs cataloged with MaceMC. This includes 52 bugs found in four systems. Spanning the three mature systems, the 33 bugs across 1500 lines of Mace code correspond to one bug for every 50 lines of code. MaceMC actually checks the generated C++ code, corresponding to one bug for every 250 lines of code. In the only comparable check of a complex distributed system, CMC found approximately one bug for every 300 lines of code in three versions of the AODV routing protocol [MPC⁺02]. Interestingly, more than 50% of the bugs found by CMC were memory handling errors (22/40 according to Table 4 [MPC⁺02]) and all were safety violations. The fact that MaceMC finds nearly the same rate of errors while focusing on an entirely different class of liveness errors demonstrates the complementary nature of the bugs found by checking for liveness rather than safety violations. To demonstrate the nature and complexity of liveness violations we detail two representative violations below; and include a detailed discussion of each bug we cataloged in § 5.6.

Typical MaceMC run times in our tests have been from less than a second to a few days. The median time for the search algorithm has been about 5 minutes. Typical critical-transition algorithm runtimes are from 1 minute to 3 hours, with the median time being about 9 minutes.

5.5.1 RandTree

RandTree implements a random overlay tree with a maximum degree designed to be resilient to node failures and network partitions. This tree forms the backbone

Table 5.2: Summary of bugs found for each system. LOC=Lines of code and reflects both the Mace code size and the generated C++ code size.

System	Bugs	Liveness	Safety	LOC
MaceTransport	11	5	6	585/3200
RandTree	17	12	5	309/2000
Pastry	5	5	0	621/3300
Chord	19	9	10	254/2200
Totals	52	31	21	

for a number of higher-level aggregation, streaming, and gossip services including our implementations of Bullet [KRAV03] and RanSub [KRA⁺03]. We have run RandTree across emulated and real wide-area networks for three years, working out most of the initial protocol errors.

RandTree nodes send a “Join” message to a bootstrap node, who in turn forwards the request up the tree to the root. Each node then forwards the request randomly down the tree to find a node with available capacity to take on a new child. The new parent adds the requesting node to its child set and opens a TCP connection to the child. A “JoinReply” message from parent to child confirms the new relationship.

Property. A critical high-level liveness property for RandTree (and other overlay tree implementations) is that all nodes should eventually become part of a single spanning tree.

We use four separate Mace liveness properties to capture this intuition: (i) there are no loops when following *parent* pointers, (ii) a node is either the root or has a parent, (iii) there is only one root (shown in Table 5.1), and (iv) each node N ’s parent maintains it as a child, and N ’s children believe N to be their parent.

Violation. MaceMC found a liveness violation where two nodes A, D have a node C in their child set, even though C ’s parent pointer refers to D . Along the violating execution, C initially tries to join the tree under B , which forwards the request to A . A accepts C as a child and sends it a “JoinReply” message. Before establishing the connection, C experiences a node reset, losing all state. A , however, now establishes the prior connection with the new C , which receives the “JoinReply” and ignores it (having been reinitialized). Node C then attempts to join the tree but this time is routed to D , who accepts C as a child. Node A assumes that if the TCP socket to

C does not break, the child has received the “JoinReply” message and therefore does not perform any recovery. Thus, C forever remains in the child sets of A and D .

Bug. The critical transition for this execution is the step where C receives the “JoinReply” from A . MDB reveals that upon receiving the message, C ignores the message completely, without sending a “Remove” message to A . Along the longest live *alternate* path found from the state prior to the critical transition, we find that instead of receiving A ’s join reply message, C gets a request from the higher-level application asking it to join the overlay network, which causes C to transition into a “joining” mode from its previous “init” mode. In this alternate path, C subsequently receives A ’s “JoinReply” message, and correctly handles it by sending A a “Remove” message. Thus, we deduced that the bug was in C ’s ignoring of “JoinReply” messages when in the “init” mode. We fix the problem by ensuring that a “Remove” reply is sent in this mode as well.

5.5.2 Chord

Chord specifies a key-based routing protocol [SMK⁺01] with guarantees that a message can be routed in $O(\lg N)$ hops using only $O(\lg N)$ space at each node. The overlay network is self organizing, has a low constant probe overhead, and is resilient to node failures. Chord structures an overlay in a ring such that nodes have pointers to their successor and predecessor in the key-space. To join the overlay a new node gets its predecessor and successor from another node. A node inserts itself in the ring by telling its successor to update its predecessor pointer, and a stabilize procedure ensures global successor and predecessor pointers are correct through each node probing its successor.

Property. We use a liveness property to specify that all nodes should eventually become part of a single ring (see Table 5.1). This minimal correctness condition guarantees that routing reach the correct node.

Violation. MaceMC found a liveness violation in the very first path it considered. This was not unexpected, given that Chord had not been tested yet. This was, in fact, the second bug MaceMC found, the first being a safety violation for a timer firing earlier than expected, which was quickly fixed. In this bug found, the critical transition

algorithm returned transition 0 and condition C2, implying that the algorithm could not determine if the path had run long enough to reach liveness.

Looking at the event graph, we saw the nodes finished their initial join quickly (step 11), and spent the remaining steps performing periodic recovery. This process suggested that the system as a whole was dead, since reaching a live state would probably not require tens of thousands of transitions when the initial join took only 11.

MDB showed us that mid-way through the execution, client0's successor pointer was client0 (implying that it believed it was in a ring of size 1), which caused the liveness predicate to fail. The other nodes' successor pointers correctly followed from client1 to client2 to client0. We believed the *stabilize* procedure should correct this situation, expecting client2 to discover that client0 (its successor) was in a self-loop and correct the situation. Looking at this procedure in the event graph, we saw that there was indeed a probe from client2 to client0. However, client2 ignored the response to this probe. We next jumped to the transition in MDB corresponding to the probe response from the event graph. In fact, client0 reported that client2 was its predecessor, so client2 did not correct the error.

Starting at the initial state in MDB we stepped through client0's transitions, checking its state after each step to see when the error symptom occurs. After 5 steps, client0 receives a message that causes it to update its predecessor but *not* its successor, thus causing the bug.

Bug. This problem arose because we based our original implementation of Chord on the original protocol [SMK⁺01], where a joining node explicitly notified its predecessor that it had joined. We then updated our implementation to the revised protocol [SMLN⁺03], which eliminated this notification and specified that all routing state should be updated upon learning of a new node. However, while we removed the join notification in our revisions, we failed to implement the new requirements for updating routing state, which we overlooked because it concerned a seemingly unrelated piece of code. We fixed the bug by correctly implementing the new protocol description.

We found the same bug using manual testing, but we expended a great deal more effort. We found the following aspects of using MaceMC particularly valuable:

(i) automatically determining that a bug had occurred, (ii) running separate nodes in one process with unified logging, (iii) matching message sending with message receiving, (iv) snapshots of the system state, (v) using the critical transition to rule out the possibility of a concurrency problem, and (vi) automated tools for visualizing, examining, and stepping through the execution.

Overall, both our manual testing and model checking approaches found slightly different sets of bugs. On the one hand, manual testing found many of the correctness bugs and also fixed several performance issues (which cannot be found using MaceMC). Manual testing required that we spend at least half of our time trying to determine whether or not an error even occurred. A single application failure may have been caused by an artifact of the experiment, or simply the fact that the liveness properties had not yet been satisfied. Because of these complexities, identifying errors by hand took anywhere from 30 minutes to several hours per bug.

On the other hand, MaceMC did find some additional correctness bugs and moreover required less human time to locate the errors. MaceMC examines the state-snapshot across all nodes after each atomic event and reports only known bugs, thereby eliminating the guesswork of determining whether an error actually occurred. Furthermore, the model checker outputs which property failed and exactly how to reproduce the circumstances of the failure. MaceMC also produces a verbose log and event graph, and in the case of liveness violations, an alternate path which would have been successful. These features make it much easier to verify and identify bugs using MaceMC, without the hassle of conducting experiments that require running many hosts on a network. We spent only 10 minutes to an hour using MaceMC to find the same bugs that we painstakingly identified earlier with manual testing; and we found the new bugs (those not caught with manual testing) in only tens of minutes.

5.6 Bugs

For every system we apply MaceMC to, we have found errors. In this section, we discuss each of the bugs cataloged with MaceMC. The errors we describe are quite complex, and indicate both bugs in the implementation of the systems as well as holes in the *specification* of some well-known protocols like Pastry and Chord. We

now briefly describe each of these protocols and the bugs that MaceMC has found in them.

5.6.1 RandTree

RandTree is an overlay tree implementation available in Mace which builds a random tree. Its key features include robustness to failures, including failure of the root of the tree, and the ability to recover from network partitions by periodic probing of nodes (called a peer set) passed into the tree on *joinOverlay()*.

It is perfectly valid for RandTree to temporarily form multiple different trees, since the root is elected, and trees are expected to merge over time. Accordingly, it is not clear what safety properties to write for RandTree. We do specify that loops are invalid, but other safety properties are not well understood. However, we can write a variety of liveness properties generic to all overlay trees, which specify that the nodes do eventually organize into a spanning tree. These properties were described in § 5.5.1.

The join procedure for RandTree is to contact a member of the peer set. (If that times out, round-robin to the next member of the peer set). When contacted, a node will pass along a join request to the root of the tree. The root of the tree then either accepts the new child, or pushes it down if it doesn't have capacity for a new child. As a special case, if the joining node has a numerically lower IP address than the root, the root will rejoin under that node instead, making it the new root. This mechanism is designed to let nodes deterministically agree on a root node.

Periodically, every node probes members of its peer set (in round robin fashion) to make sure they are in a tree which shares the same root. If the roots do not match – corrective action is taken, asking the larger IP root to join under the smaller IP root.

We found seventeen bugs in RandTree, each of which is discussed below. With RandTree, as with other protocols, the methodology was to run MaceMC until a bug is found, then to fix the bug and restart. This is necessary because once an initial bug is found, the MaceMC cannot just 'ignore' the bug and go on, since there may be many instances of the same bug along different paths. However, once the rate of

finding new bugs decreases, we do sometimes branch out, running the same protocol under different configurations to find additional bugs simultaneously.

RT1 This first bug was a divergence. A node’s join timer fired, which normally causes it to initiate a new join request to the next member of the peer set. However, it skips over any members of the peer set which are its children. However, if it has already accepted its entire peer set as children, but receives a join request from another node with a lower IP address, by the protocol it will attempt to join under them. If that join times out, it would then enter an infinite loop trying to find a suitable member of the peer set, but finding none because it had already accepted all specified peers as children. As a fix, in the join deliver handler, in addition to sending a join message to the new node, RandTree was modified to add that new node to the peer set.

RT2 As first written, a RandTree node, when asked for its parent, would return the null address if it was the root of the tree. Some time later, we modified the API to require that tree-roots return their own address as their parent (a self-loop), to distinguish them from nodes which are in the process of joining, and therefore have no parent. As a result, RandTree appeared to never form a spanning tree – since no node ever claimed to be the root by that API call.

RT3 The recovery timer (which is what periodically checks to make sure all potential peers share the same root to merge trees) was being manually rescheduled when it fired. However, in the Mace specification for RandTree, the timer firing would only be processed when the node was in the joined state. So if the timer ever fired when the node was in the joining state, no event handler would execute, and it would never again be scheduled. As a result – when peers connect to each other and form distinct trees, they would never merge, and RandTree would not form a single spanning tree.

RT4 When the root of a tree changes, a *new_root* message is propagated down the tree to inform peers of a new root. Each node maintains the root of its tree, to compare with probes from other nodes to see if they are members of the same tree. However, in cases where the join timer fired, and as a result multiple peers

(temporarily) accepted the same node as a child, that child node may receive a *new_root* message from a node other than the one which would eventually become its actual parent. However, no check was being performed on the *new_root* message, and it would update its root field on any receipt. As a result, when later being probed by members of the other tree, it would mistakenly believe they were all part of the same tree, preventing proper recovery.

RT5-9 Five more instances of RT1 were discovered. These were other cases where a RandTree node decided to join a node other than those listed in its peer set, and when they were all children, it reverted to the same infinite loop. Upon finding the second path with the same root cause, a find was done on the code to fix the remaining four cases without finding them independently.

RT10 If a node initially accepts another as a child, and then the child later asks to be removed, but the parent node receives the remove message while it is busy joining under another node, it will ignore the remove message. In this case, though the child node will no longer recognize it as a parent – the parent node will forever believe it is a child (short of a network failure or node departure), causing an inconsistent network view.

RT11 RT11 is a safety property violation. The property is that for each node, either the node is in state RandTree `_init`, or the recovery timer is scheduled (exclusively). This property was added after discovering RT3, when we realized that this was a safety property whose violation can lead to liveness violations.

When a node has had `maceInit` called on it, and receives a probe message from another node, it may attempt to join that node, transitioning to the joining state. This effectively prevents the recovery timer from ever being called, as it was supposed to be scheduled later during the `joinOverlay` call. This is another instance of the programmer not being prepared events to occur in the brief time between node initialization and when the application calls `joinOverlay`. The fix was to ignore probe messages until a node is no longer in the `init` state.

RT12 RT12 is a liveness bug. It was caused by a *remove* message being dropped by the sending transport, because the transport buffer was full. Though the send

error was immediately reported by the transport, no reaction was implemented or taken by the sending node. As a result, the recovery message was never received, and the kid state never matched the parent state.

In isolating this bug, the most useful tool was the `FindCriticalTransition` technique. It correctly reported the problem was step 68. In the end, I had to go to the log, rather than event graph, to discover the problem, because the graph did not report the send error, and displayed only the attempt to send the remove message. This also caused us to update MDB to show the matching of message sends and receives, and graphically display when messages encounter errors.

Two possible fixes for `RandTree` present themselves – (1) on an enqueue failure, call a method to close and reset the connection, removing the peer from our own state. (2) queue the send for later sending. In practice, `RandTree` is generally run without buffer limits on control messages, preventing this problem.

Starting with RT12, we modified MaceMC to output statistics about the search, so we could report its performance. The search for this bug was executed using our automated tool for running the search, followed by graph generation and critical transition search where appropriate. The search phase of this took 9.5 hours, searching 2.85 million paths, searching in the end at a depth of 35. `FindCriticalTransition` ran in 1.5 minutes.

Also noteworthy, though the critical transition was random number 158 at a simulator step of 68, the first 35 random numbers were a significant contributor to finding this bug, since they caused a large number of messages to be queued between the source and destination, which at a higher depth caused the dropped message. I believe this would have been more unlikely/harder to find without using exhaustive searching, because random walks will tend not to explore this kind of extremity as reliably.

RT13 RT13 is the bug described in § 5.5.1. The specific liveness bug occurs when a node *A* sends a join message to another node *B*, who forwards the join to node *C* (since in `RandTree`, joins all get forwarded to the root of the tree). Then, before node *C* responds to node *A*, node *A* reboots. The socket from node *A* to *B* is broken, but that does not help node *C*, who has not yet opened its

socket to *A*. If the timing is such that the response from node *C* to node *A* arrives between *A* being initialized and having `joinOverlay` called, the message is ignored as *A* is in the init state. In this case, *A* joins a fourth node *D*, and the child pointer from *C* to *A* is never fixed. It is not clear whether the presence of the fourth node is necessary.

The bug was found in path number 383365, working on a search depth of 15 (about the 280000th path searched at depth 15). `FindCriticalTransition` correctly identified the critical transition as position 18, in which the `joinReply` from *C* to *A* is delivered. In the closest live path, instead, `joinOverlay` is called on *A*, preventing the problem.

RT14 RT14 is a safety bug. Specifically, it is a violation of the timers property, which asserts that at all times, if a node is not in the init state, its recovery timer is scheduled. As with RT11, this bug occurs as a result of transitioning out of the init state based on receiving a message. In this case, it was not the probe message, but instead the `probe_join` message, which is sent to a root node to inform it of another root node to merge two trees. This error is not seen without node failures because a `probe_join` message would only be sent about a live node. A node failure resets this, and so the bug is now found.

This bug was found as path number 1350198, at a search depth of 15, but an actual random depth of 22 (simulator depth of 21). Step 22/21 is also the critical transition (though `FindCriticalTransition` is not functional for safety property violations), because if instead `joinOverlay` had been called before receiving the message, things would have worked fine.

RT15-17 RT15 is another safety bug. In this case, after forming a full tree of 6 nodes, the root node dies, followed by node 3. When node 2 gets the network error from node 0, it tries to join node 1 (the new root by minimum IP address). However, before node 1 responds, the join timer fires on node 2. It then tries to join node 3 (because it is looking round-robin in its `joinSet`), the newly restarted node 3 responds to node 2 with a response of `FORCE_ROOT`, which means that node 3 will soon be joining under node 2, as 3 was a root, but 2 has a smaller address. In the `JoinReply`, node 3 lists node 2 as the root, since it would be

the new root in a tree. This causes node 2, who at the time believes node 0 is still root, to experience a failed assertion, since it asserts that `msg.root` (which should be node 2 by the protocol) matches node its own root value. To fix this bug, we modify `RandTree` to set the root back to itself on a join timer firing – since that falls back to old join methods.

RT15 was found using the prefix search. First, I ran `MaceMC` on `RandTree` with `PRINT_SEARCH_PREFIX` set to 1, and `MAX_PATHS` set to 20. This caused the `MaceMC` to output 20 path prefixes which lead to live states. I then restarted the `MaceMC`, using `prefix1.path` as the search prefix. This searched 535 paths before finding this scenario, which required 4 nodes, two of which restarted (or at least one restarted, and the other suffered a socket error). Once I had the error path, our tools created `error.log` and `error.graph`. I used `MDB` and the event graph to figure out why the assertion had fired. The event graph was the primary tool in observing what happened, while `MDB` was used to confirm details of the failure, ensuring I understood specifically what was happening. The whole process took less than 30 minutes.

RT16 and RT17 are also safety bugs, in the same vein of RT15. The fix in RT15 is not sufficient to prevent cases where we get a force root message and the roots do not match. Instead, we now reset the root any time our parent dies. This may prevent some shortcuts during tree recovery, but should be more safe. In this 6 node case, several nodes died, and a node which did not die sent a message to another node, which eventually got forwarded through another couple of hops to a newly restarted node with a higher address, and a root force message was sent to the joiner. One key in this scenario is the root itself has failed. This bug was found using the same prefix as RT15, but was found as path 38365, still at the search depth of 5. The appearance of these two bugs also led us to write a safety property that ties together the joining state to the state of the root pointer.

5.6.2 Pastry

Pastry [RD01] is a distributed algorithm for node location and overlay routing. Every node joining the overlay becomes responsible for receiving and handling messages destined for a range of a 160-bit integer space, localized around an identifier chosen for the node¹. The node location and routing are determined using a decentralized protocol where each node keeps track of a logarithmic subset of the total set of nodes in the system.

Joining requires contacting a bootstrap peer and routing a message to the node presently managing the identifier space containing the new node's identifier, with responses that bootstrap the joining node's own state. Joining the overlay is only complete when the new node receives routing table entries for each row of its routing table, and the leaf set from its immediate neighbor. After the node has finish joining, it informs members of its leaf set about its presence, and enters the normal operating mode of the protocol, wherein it performs periodic probes and updates to ensure that every node's routing table is correct.

Pastry correctness properties are defined in terms of eventual correctness of the overlay routing structure, and in particular the ring structure formed by the nodes. It is very similar to Chord in this sense.

We have cataloged 5 bugs in our Pastry implementation using MaceMC:

Pastry1 The first Pastry bug found was a combination of several factors, which led to the third node of the simulated 3 nodes being unable to join. First, a node joining sends a message to a bootstrap node, who forwards it on towards the present owner of the address. In addition to forwarding it on each node along the path sends the number of rows to the joining node to which it shares in common a common prefix. The final destination node sends a copy of all remaining routing table rows, as well as the leaf set.

Commonly, transport services are configured with a limited buffer size, to help flow-control services sending data to slow receivers. In the Pastry implementa-

¹The identifier is frequently chosen by taking the SHA1 hash of the IP address or hostname, but the method of selecting node identifiers isn't so important as the fact that they are generally "evenly" dispersed in the 160-bit address space.

tion, nodes are not required to check that each sent message fit in the buffer—instead, the assumption is that the joining node will timeout a join attempt eventually, and re-initiate it. When it does so, the joining node sends the first row it needs, to prevent duplicate sending of rows already received. However, a bug in the forwarding node caused it to forward the wrong start row, instead always forwarding the number 1. Coupled with the fact that 40 row messages need to be received, and that a buffer might be limited to 20 messages at a time, each time the destination node received the Join message, it began with row 1, failing at by row 21. As a result, the joining node could never successfully join the overlay network.

Pastry2 The second Pastry bug illustrates a case where the Pastry specification does not fully explain how to handle the situation. When a node briefly becomes disconnected from its peers (or more generally, a network partition), and all sockets between the nodes break, on each side of the partition all references to the other side are removed, with the intention of re-establishing peering relationships through standard periodic maintenance. However, in the case of a network partition, each partition “forgets” completely about the other one, and periodic maintenance does not resolve it. As a result, in a 3 node system, MaceMC found a case where all three nodes form Pastry rings of size 1.

Pastry3 This third bug in Pastry was discovered in a system of 8 nodes, setting the leafset size to 4. It was a liveness failure – namely that the leafset relationship was not reflexive. `FindCriticalTransition` pinned it down to step 355, in which node A received an inform message from node B. However, upon adding node B to A’s leafset, `Lmax`, which is supposed to be the hash id of the furthest node away in the right half of the leafset, was improperly set to the other node in the right half of the leafset. This occurred because of a bug in `MaceKeyDiff`’s \leq operator, which was returning the wrong value. Specifically, if the difference between the two keys was greater than half of the address space, it was incorrectly treated as an overflow. As it turns out, this same bug had appeared in basic testing (without the modelchecker), and had been fixed for the $<$ operator, but not applied to the other.

Though `FindCriticalTransition` correctly returned the right step, it a bit more challenging than usual to determine the problem. In addition, we used the event graph and MDB, even directly viewing the logs. The need to look at the logs was to be able to compare two states to each other, a feature we added to MDB after finding this bug.

Below is the detailed procedure we used to find this bug:

- 1 `FindCriticalTransition` is run automatically on any non-live path
- 2 The event graph is automatically generated for both the error path and the live path found
- 3 The event graph summarizes the property failure, and can be used to see at a glance the bad step (as well as what worked)
- 4 Switch to MDB, where I skip ahead toward the end to see the details of the failed property – find the nodes which failed the property. I note that one of the two nodes happens to be the one event 355 (from `FindCriticalTransition`) occurred on.
- 5 Going back to step 355, I see that its the receipt of an inform message. I suspect that maybe the problem is these two nodes got confused about each other, and one doesn't share the other. Looking back at a later state, I realize that the sender was not one of the two nodes with a problem, though it does still appear in the other node's leafset.
- 6 I decide I now need to figure out the correct leafset. I go to the log (future mdb) to see all nodes values of myhash, and sort them manually. Upon further inspection, all nodes have their correct leafset, except the one event 355 occurs on. I conclude something must have happened in step 355 which locked that node into its leafset. I use mdb to reinspect 355, to see both the logs for that event, as well as the state of the node at the end.
- 7 I briefly go back to the logfile, where its easier to compare the difference in this node's state between two steps. (354 and 355). Not seeing anything particularly out-of-the ordinary in this exercise, I look at the state of the node at 355 in more detail. This is the point where I notice that Lmax

and Lmin state variables are strangely asymmetric – Lmin is the further away of two nodes, but Lmax is the closer of them. Inspecting the code, I determine this to be a bug.

- 8 Looking at the logs, I examine the code within `recompute_leaf_bounds`, the function which is supposed to update Lmax. I conclude the most likely scenario is that something must be broken about comparing MaceKeyDiff objects. This code was not overly well tested, so that makes sense. Careful inspection (and the logs already generated by the run) allow me to determine the problem – that the comparison is incorrect in these values due to incorrect belief about overflow.
- 9 Finally, analysis of the suspected bug matches what actually happens, affirming the work done thus far. Specifically, with Lmax containing the wrong value, the larger element of the leafset is never replaced, because no node is closer than Lmax, and if its further than Lmax it isn't considered. This would cause the leafset to be permanently incorrect. Considering its severity, it can only happen when a node's initial leafset is pretty bad – its nearest neighbor, and one further than half the keyspace away. It is therefore less likely in scenarios with more nodes, and with larger leafsets. However, the bug in MaceKeyDiff could have many other subtle effects, which this fixes.

Stats for this bug. It was found in path 5 (the first 4 of which were abandoned as duplicates), therefore at a search depth of 5. The last nail step was 355, with the last nail random position at 373.

Pastry4-5 The fourth Pastry bug found was another violation of a liveness property. In this case, we developed the prefix search, where MaceMC performs its exhaustive search, not from the initial state, but at the end of a fixed execution prefix. First, we run MaceMC, and tell it to output the paths as it completes them. We let it run for a few paths, generating potential execution prefixes that end in live states. We then re-run the model checker, instructing it to run the execution prefix before beginning its search. This allows the modelchecker to get past the joining phase of the protocol, where, since it takes an 8-node

Pastry ring on the order of 700-800 steps to reach liveness, where exhaustive search would not reach.

The specific bug found was one where not all nodes find a path to a joined state, despite starting in one. In the violating path, node 5, after restarting, cannot rejoin. Node 0 (the bootstrap node) sees the socket error from node 5, but before 5's second join attempt begins, it receives state from another node (in this case a leafset info message) which mentions node 5. Though 5 will not be added to node 0's leafset (it only adds known live nodes to a leafset), it is added to node 0's routing table. Later, when node 5's join is received by node 0, node 0 forwards it back to node 5, preventing it from being able to join.

One heavyweight solution to this problem would seem to be to probe all nodes before adding them to the routing table, and only inserting them if they respond that they are alive and running. This, however, was determined to be prohibitively expensive, and so was ignored as a solution. (It also only works if nodes maintain a connection to each peer in their routing table.)

Looking beyond that solution, at first approximation, the problem is that a node forwards the join message back to the joining node. Accordingly, our initial naive implementation was to check when forwarding a Join message that we were not forwarding the message back to the joining node. While this attempted solution did in fact prevent the specific case observed using MaceMC, it did not solve the problem. When running MaceMC again, we found bug PASTRY5, another variation of the same problem.

In the new violating execution, two nodes failed simultaneously. Though their socket error removes them from node 0's state, subsequent periodic updates cause them to be re-added to the routing table for node 0. Although the new code prevents their join messages from being routed back to them, in this case, each node's join message is forwarded to the other node (also not joined). The actual solution implemented to fix this bug was to respond to messages received before join completion with a note to the sending node to remove the joining node from their routing table. This way, the node state can be cleared, for future join attempts to succeed.

5.6.3 MaceTransport

MACETRANSPORT is an implementation in Mace of a (user-level) transport service providing reliable, in-order, message delivery with duplicate-suppression and TCP-friendly congestion-control. MACETRANSPORT is implemented using a basic unreliable UDP transport.

The simulated application for the MaceTransport sends a configurable number of messages to remote destinations, restarting if the connection socket is broken before the transmission is complete. The liveness properties for this service are that eventually the messages should be delivered, and the safety properties are that each message should be delivered at most once on any connection. Allowing restart means the simulated application can test cases with node failures. The simulated application keeps track of whether a complete transmission is received during the lifetime of a node, so even if the node reboots, it need not retransmit all the messages.

MaceTransport1 In the first MaceTransport bug found by MaceMC, a message sent by the simulated application was delivered to the receiver twice. The specific steps leading to the bug were: 1) The source sends the first message in a new connection (syn flag is set). 2) The receiver receives and delivers it and sends an ack. 3) Before the source receives the ack, the retransmission timer fires, so the source retransmits the message. 4) The receiver receives the duplicate. Because the syn flag is set, the receiver concludes that the source died and came back, so it erases all state associated with the connection, creates new connection state, and then delivers the duplicate.

This exercises a corner case, where a retransmitted message contains a syn flag, confusing the destination. To prevent the error (and the duplicate delivery, additional state is kept at the receiver about the initial sequence number, to support special duplicate suppression of the data+syn packet.

MaceTransport2 The second bug found in MaceTransport manifested itself as a failed map lookup, where state that had been deleted was referenced. The problem was that the retransmission timer was not canceled for all inflight messages associated with a connection that was being closed. When a connection is

closed, all of its state is erased. A new connection to the same destination was then opened. When the retransmission timer fired for an old message, it actually executed because the guard thought that the connection was still present as it had been re-opened. The timer then tried to reference the deleted state and caused an assertion to fail.

The fix is that all timers for inflight messages must be canceled when closing the connection. We also added a new safety property that would have caught this problem in the first place.

MaceTransport3 MACETRANSPORT3 is the bug described in § 5.4. For MACE-TRANSPORT3, when syn packets were delivered out-of-order, the receiver may expect different sequence numbers than the sender and no data will be sent or delivered on the connection. This situation could be corrected by the retransmission timer closing the connection with an error, so this is best considered a performance bug. MaceMC however found it using a liveness property, because the weight of a timeout event was selected to be 0, preventing it from occurring during random execution. The implication of this decision is to assert that the MaceTransport should be able to recover from any problems without reverting to a timeout, in the absence of node and network failures.

Setting this weight of a timeout to be non-zero would have allowed the timeout to occur, and there would have been no bug found by MaceMC. However, the techniques described in § 6 would have isolated it as a performance bug. This demonstrates that many of our systems are robust enough to mask even their own bugs, making them seem to be simple performance problems. Interestingly, this complicates the debugging process as well, as the problems are masked from the tester as well.

The fix is to have each sender send a monotonically increasing identifier (implemented as a fine-grained timestamp) in the syn packet. The receiver maintains the most recent identifier from each sender and ignores syn packets with an identifier that is old.

The modelchecker also discovered two errors in the initial attempted fixes. The first was that we omitted a necessary check in an if-else clause which caused an

out-of-order syn identifier to still be processed. This was fixed by adding the check and clause. The second was that the identifier was initially stored in the incoming connection state, which was lost when the incoming connection was closed or reset. The fix for this was to store the identifier in a separate state variable map that persists across connections.

MaceTransport4 Out-of-order packet delivery to the transport caused an internal assertion to fire that prevented duplicate message delivery to the application. The fix was to increment a counter when delivering buffered messages that arrived out-of-order.

MaceTransport5 Out-of-order packet delivery caused the transport to send an acknowledgment for a future sequence number, which should never happen, instead of the most recently received packet. The fix was correctly setting the ack sequence number when packets arrive out of order.

MaceTransport6 Out-of-order packet delivery caused the transport to continue to retransmit packets that had already been acknowledged. On receipt of an acknowledgment with a sequence number not found in the inflight message map (ie, the sequence number is greater than any inflight message), the transport would previously do nothing. The fix was to check if the sequence number is greater than all inflight messages, and, if so, have the transport mark all inflight messages as acknowledged.

MaceTransport7 Out-of-order packet delivery caused a receiver to delete buffered data upon receipt of an ack when the data will need to be retransmitted. A somewhat complicated series of events need to occur to demonstrate this performance/bounded liveness bug (note that the retransmission timer will fix this condition).

Essentially, the bug occurs because when a reset is received, both incoming and outgoing state get erased. This is problematic when resets are received out-of-order, as the incoming connection state is lost, even though the sender does not realize this.

The fix is to close the connection in only a single direction. On a reset, close the outgoing connection. On a syn, close the incoming connection. On a timeout, close both directions.

MaceTransport8 Out-of-order plus duplicate packet delivery caused the transport to attempt to deliver fragmented message before all the fragments had been received. The fix is to check if an out-of-order received message is already buffered, and, if so, do not process it again.

MaceTransport9 In another performance bug, the transport was sending an acknowledgment for the most recently received packet, rather than the cumulative last in-order packet. The fix is to check for out-of-order buffered messages before sending the acknowledgment. MaceMC did not directly find this bug (there were no assertions for this property), but rather it was discovered while inspecting the event graphs for other runs.

MaceTransport10 The Clear-To-Send timer was not being canceled when the connection signaled an error, causing too many CTS callbacks to be made to the application. This was caught by a safety property. This bug did not manifest itself until testing with 3 nodes, since with 2 nodes the liveness properties were being satisfied before the extra CTS callbacks could occur.

MaceTransport11 Another performance/bounded liveness bug. Out of order message delivery can cause the sender and receiver expected sequence numbers to differ. The fix is track the number of duplicate out-of-order buffered messages received. If this value exceeds some threshold, then reset the connection (“fast reset”).

5.6.4 Chord

MaceMC has also been used to model check our implementation of the DHT routing protocol Chord [DZD⁺03]. The Chord protocol is very similar in concept to the Pastry protocol, but varies in some significant technical details. However, our experimentation has shown that Chord suffers from the same troubles as Pastry when it comes to recovering from network partitions.

In testing Chord, we first tested it by hand, initially with just 2 and 3 nodes, to verify basic functionality without failure. Then, we conducted Bamboo-style tests [RGRK04a], conducting lookups on a churning network with 300 nodes always alive. Multiple identical lookups are executed simultaneously, to determine if the network is performing consistent lookups. Between these two tests, we debugged our implementation over the course of several weeks. Once the Chord implementation achieved an acceptable performance of lookups, we returned to the modelchecker from the initial version to compare the bug finding techniques. We ran the modelchecker for 5 nodes, allowing network failures, UDP drops, and node failures.

Both approaches found a slightly different set of bugs. Using the manual testing approach we found many of the known correctness bugs, but additionally fixed a number of issues with performance which can not be found using MaceMC. However, the modelchecker found all the correctness bugs of the lookup tests, and also some additional ones. Furthermore, it did so while requiring less human time. When conducting manual testing, once past the basic sanity tests, a large portion of time was spent even trying to determine whether or not bugs even occurred. After all, a single inconsistent lookup is only an indication of a problem, but could also be explained by a node failing while handling the lookup, or non-synchronized lookups which get *correctly* delivered to different nodes based on arrival or departure of the destination node. Because of this, identifying bugs manually took from 30 minutes to several hours per bug.

Instead, MaceMC can examine the state-snapshot across all nodes after each atomic event, and can report only known bugs. Furthermore, the modelchecker reports which property failed, and exactly how to reproduce the circumstances of the failure. It also produces a verbose log and event graph, as well as in the case of liveness bugs an alternate path which would have been successful. These features make it much easier to verify and identify bugs using MaceMC, and without the hassle of conducting experiments which require many hosts and a modelnet environment. The same bugs found by MaceMC took only 10 minutes to an hour to find. Bugs which were not found by manual testing might take longer to isolate, but in practice these too took only a tens of minutes to find.

Chord 1 At the beginning of testing bugs, the Chord implementation was untested.

As a result, the model checker quickly found its first bug – a case where the `fix_fingers` timer fired, and $(next == me)$ (which was asserted to be false). This tests that the next hop towards the finger we are trying to fix is not the local node. In this early version of Chord, if we are the successor for a key, we go ahead and route the message locally, instead of, as is required, to the strict predecessor for a key. To solve this, we added a flag for *isPred*, which would be true if we were the successor or predecessor for a key. The assertion then became *ASSERT(isPred || next != me)*, and if *isPred*, we would route to our predecessor. (Safety violation at step 143, on path number 9 [the first 8 terminated early due to duplicate detection], modelchecking time 4.7 seconds, search depth 5)

Chord 2 (This bug is described in § 5.5.2.) After fixing the first bug, the MaceMC quickly found its second bug. This one however, was a liveness bug, but the critical transition was the first transition. This case raised condition C2, so we had to determine whether the system was completely dead, or whether the random walk simply hadn't been given enough time. The condition violated was one where from each node, following the successor pointers should encompass all nodes in the Chord ring. Looking at the event graph, the only thing immediately obvious was that the nodes finished their initial startup events quickly (step 11), and spent the remaining steps performing periodic recovery. This suggests that probably the system as a whole was dead, since if it could have recovered, it likely would have given the pattern of recovery steps. Next step was to look at MDB, which allowed me to see that at step 10000 (just to pick a random step) showed that client0's successor pointer was client0. Client1 had client2 as its successor, and client2 had client0 as its successor, so only client0 seemed to be problematic. At this point we wanted to determine 2 things. First, how did it get in this state in the first place, and second, why was the periodic recovery not correcting it? Starting with the second question, we looked at the code, and discovered that the *stabilize* timer is supposed to correct this situation, expecting client2 to probe client0, and discover that its predecessor

is incorrect, and tell it to insert itself in the after client2. Looking at this recovery in the event graph, we see client2 send a *get_pred* message to client0, who responds with a *get_pred_reply* message. So far so good. But client2, upon its receipt, does nothing. Looking further into this, we jump to transition 10014 in MDB, which is a *get_pred_reply* receipt. However, this shows us that client0 is in fact reporting that client2 is its predecessor, which is why client2 thinks everything is okay. With this new information about why the problem persists, we now turn our attention to where this first happened. In MDB we jump back to the initial state, and start stepping through client0, checking its state after each step to see when this occurs. 5 steps into client0, in system step 12, client0 receives an *update_pred* message which causes it to update its predecessor, but it does not update its fingers (and in particular its successor), which leads to the bug. A bit more checking indicates that this problem arose because the original MACEDON implementation of Chord was based on the conference paper [SMK⁺01], where the TON paper [SMLN⁺03] was used in the revisions for Mace. The solution was to update the fingers when receiving an *update_pred* message. (Liveness violation, on path number 6 [first 5 exercised duplicate states], critical transition at step 0, model checking time 5.9 seconds, last nail ran in 70 seconds, search depth 5)

Chord 3 This bug was one which pointed out a downcall which was not implemented. The downcall was required to test the properties, and therefore caused an assertion to fail. The solution was to implement getNextHop(). This also demonstrated a problem in our event graph generation tool, which had assumed it need not worry about incomplete events. We fixed the event graph generation to handle such errors more gracefully. (Safety violation at step 102, model checking time unknown, path number around 235000, search depth 15).

Chord 4 The first three bugs were found before with MaceMC configured not to fail nodes or sockets. Now, with failures turned on, the first bug found was one we had noticed during the port, but left in the code to verify the modelchecker could find it. The bug was in handling errors; the code there incorrectly updated the fingers and successor, and the result was a liveness violation found by the

modelchecker, with a critical transition at step 9, where client1 rebooted. To solve this problem, we simply enabled the fix we had put in place during the port. (Liveness violation, on path number 1022, critical transition at step 9, model checking time 11.2 seconds, last nail ran in 4 minutes, search depth 10).

Chord 5 This bug was another one we had anticipated, where a partition in the Chord network causes two distinct rings to form. In this case, a 3 node network, where one node was partitioned from the other two formed a ring of one, while the other two created a ring of two. This behavior was expected since we had seen the analog of this bug while testing our Pastry implementation. Neither system contains a description of how to solve this problem in their protocol description. To fix this for Chord, we added a recovery timer, and periodically execute "find_pred" to your bootstrap peers with your own key. If you get a find_pred_reply, that means recovery is needed. You consider adding the respondent to your routing, and then you tell their predecessor to consider adding you. (Liveness violation, on path number 1021, critical transition at step 9, model checking time 9.5 seconds, last nail ran in 3.5 minutes, search depth 10)

Chord 6 After the initial fix for Chord 5, an exception was being thrown due to a null predecessor after error. We added a new safety property to check for this, as well as some code to fix it. (Safety violation in step 46, model checking time 12 seconds, path number 1905, search depth 10)

Chord 7 A liveness bug found when bootstrap nodes are chosen randomly from the set of nodes previously initialized instead of always as client0. This bug occurred where nodes formed distinct rings due to a mistake in find_pred when the message is initially sent to a node which is the destination (and not the predecessor). Solving this bug involved modifying the technique to route strictly to a predecessor when executing find_pred. Upon finding this bug, MaceMC had found all the bugs observed in manual testing of Chord. It did not isolate any of the additional performance bugs found in manual testing, because as it has no notion of how long anything takes, performance bugs cannot be found. (Liveness

violation, on path number 58720, critical transition at step 36, model checking time 8 minutes, last nail ran in 9 minutes, search depth 10)

Chord 8 Another liveness bug where a node bootstraps off of a node which doesn't realize this node rebooted. The joining node gets itself back as successor, and updates its successor state to point to itself, despite having a predecessor other than itself. Further, the `fix_fingers` timer returns if `pred == self` or `succ == self`, since the ring is empty, and therefore ignores this case. The solution for this bug involved updating the joining path to update fingers when getting your initial predecessor. Thus, at least the node gets a successor even though it is told it is its own successor. Also, since this was the second time we had seen problems when the successor variable was a self-reference while the predecessor variable referenced another node, we added a new safety property that ($pred = me \Leftrightarrow succ = me$), which we call *SuccessorPredecessor*. (Liveness violation, on path number 5,118,422, critical transition at step 37, model checking time 10 hours, last nail ran in 3.5 minutes, search depth 15)

Chord 9 After adding the new safety property, the model checker began to find other cases where this occurred. This occurred in reboot scenarios and in conjunction with the recovery timer. When a peer recovery occurred and we got back a `find_pred_reply`, we were updating the fingers with the node we discovered, but did not update our predecessor if it was better than our own. This was only a problem when we thought we were our own predecessor. The solution was to check this case and update our predecessor in these cases. (Safety violation in step 35, model checking time 58 seconds, path number 8086, search depth 10)

Chord 10 A liveness bug was seen in which a node *A*, while joining, may have its own, or or another node's join attempt routed to them, while this *A*'s join request was being routed to that node. This problem we note also was present in our Pastry implementation as well. Here we were able to solve the problem by adding a *remove* message, which is sent if you detect that another node has you in their routing table, and you are presently joining. That message causes them to remove you from their routing state. Over time, this clears this problem and allows nodes to join again. (Liveness violation, path number

3,969,436, critical transition at step 33, model checking time 7.1 hours, last nail ran in 6.5 minutes, search depth 15)

Chord 11 A new execution which violated the *SuccessorPredecessor* property. This was simply a case where the solution for Chord 9 added a new case where the predecessor was set, but not the successor. (Safety violation at step 41, on path number 13,425,634, model checking time 28.8 hours, search depth 20)

Chord 12 A new execution which also violated the *SuccessorPredecessor* property, and was a corner case forgetting to set the predecessor. This case was found using the prefix search technique, which started each execution prefixed with an initial execution to a live state. (Safety violation at step 161, on path number 4990, model checking time 5 minutes, search depth 5)

Chord 13 A new execution which also violated the *SuccessorPredecessor* property. In this case, upon getting a find_pred message when a node has no predecessor and no peers, it was updating the finger table, but not setting its predecessor. (Safety violation at step 179, on path number 554906, model checking time 8.5 hours, search depth 10). This bug was found using the prefix search technique, which started each execution prefixed with an initial execution to a live state.

Chord 14 A new execution which also violated the *SuccessorPredecessor* property. In this case, applying the fix for Chord 13 opened up a new case to violate the property which was exercised because a range check was being applied to the wrong identifier: it should have checked the actual identifier instead of the identifier plus one. (Safety violation at step 39, on path number 2, checking time 7.6 seconds, search depth 5). After fixing this bug, we next merged the two versions of Chord, the one we fixed by hand-testing, and the one we fixed using MaceMC. This of course introduced new bugs.

Chord 15 One difference in the two versions had been that in the hand-tested version, for efficiency's sake, when a node's predecessor dies, you do not modify your node's id space, but instead clear your predecessor to indicate you are waiting for a node to fill it. In the merge, the keyrange was not properly updated in some cases when setting the predecessor. This was a liveness violation that all

nodes could not be reached through the closure of *getNextHop* called on each of the node's identifiers. The incorrect keyrange affected the routing to prevent its success. This bug took a significantly longer period of time to diagnose compared with the other Chord bugs, because although MaceMC provided the last nail, event graphs for live and dead paths, and mdb to consider both logs, it took a long time to figure out exactly what should have been happening for the routing to work properly. In this case, though it took a long time, it was especially valuable to have MDB to step through and compare states from the live and dead executions, and the event graphs to see at a glance what happened in both cases. This bug was also interesting in the sense that MaceMC stumbled on it mostly by chance, in the first path searched, when the critical transition was not even in the first 50 steps. (Liveness violation, path number 1, critical transition was step number 89, model checking time 25.3 seconds, last nail ran in 1.4 hours—the machine was thrashing with other jobs while running this last nail)

Chord 16 We found an execution which fired an assertion because the constant version of the *getHashIdPlusOne()* method was called when the finger's *hashId* was null. This occurred as a result of the decision that in the merged version, on failure the predecessor would be cleared rather than set to another node. Then, when processing a *find_pred_reply*, to consider whether to add as the predecessor, we tried to see if it was in range. Under the new paradigm, we simply check to see if the predecessor is empty, and set it automatically then. Additionally, with this bug we updated the properties to allow null predecessors to imply that a node's successor is itself. (Safety violation at step 72, path number 6472, model checking time 81.5 seconds, search depth 10)

Chord 17 After fixing Chord 16, there were circumstances where a node gets a *find_pred_reply* when its predecessor is itself. Thus, if a node is about to add a peer to its finger routing table, and its predecessor is itself, the code must update it. This caused a violation of the updated *SuccessorPredecessor* property. (Safety violation at step 93, path number 35556, model checking time 7.6 minutes, search depth 10).

Chord 18 Previously, if a strict predecessor could not be found when one was requested, and the local node’s predecessor is not set, then a null key was returned from `nexthop`. But the `find_pred` message is dropped if null is returned, meaning nodes cannot join in an empty ring. To solve this problem, we modified `nexthop` to return *me* instead of null, correcting this problem. (Liveness violation, path number 907856, critical transition at step 24, model checking time 3.8 hours, last nail ran in 2 minutes)

Chord 19 This bug was a liveness violation caused by a faulty state guard on the transition to send “remove” messages to nodes who had bad routing table entries (of joining nodes). The broken guard did not send the message if it was the first hop of the “find_pred” message (assuming it was an attempted join). However, other first hop “find_pred” messages exist, and the failure to respond to these caused a bug where some nodes could not rejoin after rebooting. Finding this bug within MaceMC took about an hour. Fixing this bug involved adding a “joining” flag to “find_pred” messages, to distinguish between these messages to allow the join-cache to remain. (Liveness violation, path number 3638, critical transition at step 169 (random position 193), model checking time 5 minutes, last nail ran in 20.8 minutes) Last nail took as long as it did due to the large number of dead executions it visited.

5.7 Summary

The most insidious bugs in complex distributed systems are those that occur after some unpredictable sequence of asynchronous interactions and failures. Such bugs are difficult to reproduce—let alone fix—and typically manifest themselves as executions where the system is unable to *ever* enter some desired state after an error occurs. In other words, these bugs correspond to violations of *liveness* properties that capture the designer’s intention of how the system should behave in steady-state operation. Though prior software model checkers have dramatically improved our ability to find and eliminate errors, elusive bugs like the subtle error we found in Pastry have been beyond their reach, as they only find violations of *safety* properties.

We have described techniques that enable software model checkers to heuristically isolate the complex bugs that cause liveness violations in systems implementations. A key insight behind our work is that many interesting liveness violations correspond to the system entering a *dead* state, from which recovery to the desired state is impossible. Though a safety property describing dead states exists mathematically, it is often too complex and implementation-specific for the programmer to specify without knowing the exact bug in the first place. Thus, we have found that the process of finding the errors that cause liveness violations often reveals previously unknown safety properties, which can be used to find and fix more errors. We have used MaceMC to find and catalog 31 liveness (and 21 safety) errors in Mace implementations of four complex distributed systems. We believe that our techniques—a combination of state-exploration, random walks, critical transition identification, and MDB—radically expand the scope of implementation model checkers to include liveness violations, thereby enabling programmers to isolate subtle errors in systems implementations.

5.8 Acknowledgement

Chapter 5 is an updated and revised copy of the paper by Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat, titled “Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code,” as it appears in the proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation 2007. The dissertation author was the primary researcher and author of this paper.

Chapter 6

Analysis of Performance

Using MaceMC (just described in § 5) we can find violations of liveness properties by systematically exploring the space of possible executions from a fixed starting point, then testing each state we encounter to see if it is a dead state using long random executions that only terminate when some high-level liveness property eventually becomes true. While this approach has been successful at helping us discover and fix corner-case executions that cause our systems to fail in deployed scenarios, it is incapable of helping us detect and solve bugs that affect the performance of our system, because the tests only care *if* the liveness condition is ever satisfied, and not *when*.

In fact, after performing extensive model checking of the random tree protocol, which serves as the backbone for many of our higher-level services, we had deemed it to be mostly correct, only to have users complain that in practice, some nodes were never joining the tree, and therefore were never downloading a file that BulletPrime, a file distribution protocol which makes use of a tree service, was trying to disseminate.

When we used MDB to explore the logfiles to see what was happening, it became apparent that the nodes were in fact forming independent trees, despite all trying to bootstrap off of the same peer. We hypothesized that after forming disconnected trees, the recovery timer, whose responsibility is to correct network partitions, should correct the problem. But the recovery timer had only been scheduled to expire every hour, since network partition events are expected to be infrequent, and in practice it was not ever expiring during a 10 minute experiment. To test the hypothesis, we

adjusted the recovery timer to fire every 20 seconds, and re-ran the experiment. This time, though multiple trees did initially form, over the period of about a minute, they did merge into a single tree.

Practically, it does not make sense to fire the recovery timer every second, which was the interval required for the join process to not impact the results of the file distribution, a matter of only 2 minutes. Furthermore, there is the question of how the tree became disjoint in the first place. The recovery timer was correctly fixing partitions, but the programmer had believed partitions would only occur in the presence of network errors or node churn.

In manually tracking down this bug, we learned an important lesson—*the robust nature of distributed protocols can actually mask bugs, converting correctness bugs into performance bugs*. Furthermore, since these bugs are not readily observed by correctness testing, they often persist in deployed mature code, never being found, instead being felt as occasional performance glitches.

Our first approach to fixing these bugs in our random tree protocol was to disable the recovery timer, and then also configure MaceMC to not explore executions that have node or socket failures. This approach did allow us to fix several problems in our RandTree implementation, but unfortunately was not satisfying, as it may not always be practical to separate functionality like the recovery timer paired with model checking search parameters. Even in the case of RandTree, we would not find any performance bugs caused by problems in handling node or socket failures. Accordingly, we sought out to develop an extension to MaceMC that can automatically explore the execution space, isolating performance and correctness bugs.

While ensuring the correctness of distributed systems is a necessity, finding performance problems is arguably even more challenging. Relatively rare interactions among individual nodes in the distributed system or the inconvenient delay or drop of a single message can lead to poor performance. These performance anomalies are typically rare and also difficult to reproduce because of the high expense associated with logging overall system behavior on multiple distributed nodes. Even once the conditions leading to a performance bug can be isolated, the process of diagnosing the actual error is currently tedious and error prone. Individual programmers are left with the daunting task of wading through hundreds of megabytes of log files

and manually tracing communication and control flow across nodes in the distributed system.

In this chapter we describe tools to: (i) automate the process of finding performance bugs in distributed systems and (ii) to isolate the root cause of the error to assist the programmer in fixing the performance issue. To be widely applicable, we believe any such tools must operate on the unmodified source code of the distributed system. Further, the tools cannot require running the system across the network as a part of the debugging process. Acquiring the necessary computation and network resources to run a distributed system and then collecting and parsing individual log files from multiple nodes are sufficient impediments to limit many debugging efforts even with appropriate tools.

The key insight behind our work is that we can employ guided search using techniques similar to model checking to compare the performance of distributed systems implementations under a range of random executions. The key distinction between such guided search and traditional model checking techniques is that the search must maintain a realistic notion of time and that the exploration of the search space cannot be exhaustive because of the infinite space of possible timing interactions, when compared to the merely very large space of possible event orderings for traditional model checking.

This chapter makes three principal contributions. First, we show how to build a system to performance check a distributed systems implementation. We train the system to follow distributions of per-event execution times measured from simulated executions of the actual, unmodified system code. We then explore a variety of random event orderings and timings to develop a profile of expected system performance. This allows us to flag executions that significantly deviate from the average case performance as demonstrating a performance anomaly. These poorly performing executions often produce log files consisting of hundreds of megabytes of data with tens of thousands of events spread across one hundred or more nodes.

Manually inspecting these log files to verify, isolate, and fix a performance error is a challenging task. Thus, our second contribution is an algorithm to automatically characterize the *divergence point* in a given execution. That is, we isolate the particular point in the execution after which it likely becomes impossible for any

possible future event ordering to ever achieve good performance. Prior to this divergence, some alternate path would still achieve acceptable performance. Equipped with both the divergence point and an alternate execution that achieves good performance after the divergence, we find programmers are able to much more quickly diagnose and fix performance errors.

Finally, we present the results of applying our techniques to four real, complex distributed systems. We discuss in detail the application of our tool to these systems, the subtle performance bugs we found, and our solutions. Importantly, all of our implementations were mature and have been run across the Internet for several years. Further, while we had been aware of intermittent performance issues and had attempted to diagnose them on multiple occasions, we were unable to do so without the techniques presented in this paper.

6.1 Background

6.1.1 From Model Checking to Performance Testing

The design of this tool is heavily inspired by MaceMC, and in fact shares some implementation code. However, the differences are substantial enough that it has not been possible to have a single implementation perform both tasks. At a high level, the biggest difference in the two systems is that the new tool must track the time at each node, which forces the implementation to be different in several ways. For example, consider the event processing loop: MaceMC would enumerate all possible events and randomly pick one, doing a ‘late-binding’ of the event ordering, while the new tool randomly schedules future events, doing an ‘early-binding.’ These, and other subtle differences lead to different design decisions, and the new design is described in the following sections. One of the real surprises for us was that the critical transition technique used in MaceMC could be adapted to help isolate the divergence point in a poorly performing execution.

6.1.2 Basic System Model

As with MaceMC, a distributed system is mapped into a single process by running each node as a separate object in the process and connecting the nodes together through a simulated in-memory network. The addition of application and timer simulators and a modified random number generator allow deterministic replay and search of system executions.

The implementation of a distributed node consists of a set of service objects that implement event handlers. When not in an event handler, a node blocks waiting for another event. Services implement three kinds of event handlers as state *transitions*: application, network, and timer transitions. An *event* occurs by calling a handler on one of the service objects at the node. It may in turn execute an arbitrary piece of non-blocking code and invoke an arbitrary set of transitions on other service objects in the same node. The event executes atomically to completion, modifying local variables and generating new events to deliver later.

The *state of a node* is the set of all variables of all service objects at that node. The state of all nodes together, combined with the network, timer, and application simulator state comprise the *system state*.

An *execution* of the distributed system is an *initial system state* and an ordered set of pairs:

$$\langle node, event \rangle$$

At each simulator step, the simulator simulates the given *node*, executing the unmodified transition corresponding to the event *event*. After each event, the system is in a new state. The system can then be tested for violations of state conditions satisfaction of the *stopping condition*, which tells the model checker to move on to the next execution.

6.1.3 Dealing with Time

Thus far, we have completely ignored the importance of time in the system model. While ignoring time is standard practice in model checking for correctness, maintaining a notion of time is required in nearly all analyses of system performance.

In an event driven system, there are two basic ways time affects system execution. The first is the schedule of when events should occur at individual nodes. This accounts for both the latency of network messages and the delay caused by scheduling a timer to fire later. The second is the execution time of the event code itself. While our system model requires event handlers to be non-blocking, and therefore fast, in practice they take a non-negligible amount of time. To be consistent, a node's time must be updated according to both of these factors.

Thus, taking time into account, we describe an *execution* as an initial system state and an ordered set of tuples:

$$\langle node, event, start, duration \rangle$$

At each simulator step, the model checker simulates the first event in the list (ordered by *start*). It first updates the system clock for the simulated node to be the maximum of its current time and the time the event was scheduled. The simulator then calls the transition for the event and updates the node time by the amount of the *duration* when the event completes.

As part of the system state, we will now include a clock for each node. At the end of each event, the time at each node is consistent with the $(start + duration)$ of the last event at that node. We then define the *average system time* as the average time across nodes. By subtracting the system start time from the average system time, we can measure how long the execution has been running. We define the *average execution duration* as the length of the execution when the system state satisfies the *stopping condition*.

6.2 Model Checking

In this section we describe our algorithms for finding performance-oriented bugs in unmodified systems code. Our model checking process occurs in two main phases. In the first phase, we simulate the execution of a system as described in Section 6.1.2 to produce a set of executions suspected to have performance bugs. In the second phase, we compare these anomalous executions to non-anomalous ones to

aid the programmer in finding the cause of the bugs. Our overall system architecture can be seen in Figure 6.1.

At the core of our techniques are two main algorithms, presented as Algorithm 4 (*EventSimulator*) and Algorithm 5 (*Search*). *EventSimulator* implements the simulator as described in Section 6.1.3. This algorithm takes three parameters: the system we wish to model check, the stopping condition, and a set of event timing distributions (further described in Section 6.2.1). The algorithm constructs three queues: (i) an *events* queue initialized with a small set of events to bootstrap the system—typically an application “init” event on every node, (ii) an empty *timedEvents* queue, and (iii) an empty *realTimes* queue.

The algorithm pops the first event, which has the smallest start time, off the queue. The node running the chosen event sets its clock to the maximum of its current clock value and the start time of the event. Next, we record the real time and execute the event, potentially causing other events to be added to the queue (e.g., in the case where a new timer or network event is scheduled). After execution, we retrieve the timing distribution corresponding to the event (this will be explained in § 6.2.1 as the event duration distribution, or EDD) and evaluate it at a randomly chosen percentile to capture variations in the execution of individual events. The value returned is assigned the event’s *duration*.

The node advances its clock by this duration and adds a new $\langle \text{node}, \text{event}, \text{start}, \text{duration} \rangle$ tuple to *timedEvents*. An entry containing the event’s ID, a string representation of an event, and the event’s actual execution time is added to *realTimes*. Finally, we test the stopping condition, and if it is met, return *timedEvents* and *realTimes*. This process continues until we meet the stopping condition or the system has no more events to process. However, the latter case indicates an error.

Note that our approach to timing events assumes events are independent and hence uncorrelated with one another. While not true in practice (e.g., a particularly slow node is likely to have multiple consecutive slow events), this approach has been sufficient to explore naturally occurring variations in event orderings and timings, enabling us to find a number of interesting performance bugs. We leave modeling potential correlations among event and network timings to future work, though we

believe our initial simpler approach will, in general, suffice to find a range of interesting performance problems.

Algorithm 4 EventSimulator

Input: System S

Input: Stopping condition C

Input: Set EDD of timing distributions

$events = \text{Queue of } \langle node, event, start \rangle$

$timedEvents = \text{Queue of } \langle node, event, start, duration \rangle$

$realTimes = \text{Queue of } \langle eventId, realTime \rangle$

$S.initEvents(events)$

while not $events.empty()$ **do**

$\langle node, event, start \rangle = events.pop()$

$node.time = \max(node.time, start)$

$startTime = RealTime()$

 Simulate $event$ on $node$

$endTime = RealTime()$

$realTime = endTime - startTime$

$dist = EDD[event.getId()]$

$duration = dist[rand()]$

$node.time = node.time + duration$

$timedEvents.push(\langle node, event, start, duration \rangle)$

$realTimes.push(\langle event.getId(), realTime \rangle)$

if S satisfies C **then**

return $\langle timedEvents, realTimes \rangle$

signal ERROR

Algorithm 5, **Search**, simply calls **EventSimulator** in a loop, implementing a guided search of the execution space. This algorithm takes the system to evaluate, a stopping condition, a set of event timing distributions, and an integer N as parameters and calls **EventSimulator** N times. It appends the *execution* object returned from each execution to the *execs* queue and flattens the *times* queues into a single

eventTimes queue, returning both *execs* and *eventTimes* upon termination. Between each invocation of **EventSimulator**, the algorithm resets the system state, which includes tasks such as clearing the simulated network of all messages, instantiating new nodes for the next execution, removing all scheduled timers, and resetting the random number generator state.

Algorithm 5 Search

Input: System S

Input: Stopping condition C

Input: Set EDD of timing distributions

Input: Integer N

$execs = \text{Queue of } \langle execution \rangle$

$eventTimes = \text{Queue of } \langle eventId, realTime \rangle$

for $i = 1$ to N **do**

 Reset system

$\langle ex, times \rangle = \text{EventSimulator}(S, C, EDD)$

$execs.\text{push}(ex)$

 Add all elements in $times$ to $eventTimes$

return $\langle execs, eventTimes \rangle$

6.2.1 Searching

The first phase of performance model checking is the *search* phase. As mentioned above, the goal of the search phase is to find executions with suspected performance bugs. Doing so requires characterizing a “normal” execution. Because we are primarily concerned with the average execution duration, it is paramount that our simulation of time be as accurate as possible. Although it is feasible to simply time the execution of each event on each node and use these values to advance the node’s clock, we chose a more flexible abstraction.

We represent the timings of each event with a distribution generated from actual executions of the system. We call these the *event duration distributions* or *EDD*. To collect these distributions, we pick a value N such that N random executions

of the system should generate all possible events. We define an arbitrary EDD that maps all events to a constant value. Next, we execute **Search** on the system, the stopping condition, our contrived EDD, and N . When the search is complete, we can discard the *execs* queue, because we are only interested in the actual time each event took. By looking at the *eventTimes* queue, we can compile distributions for each event, using *eventId* as the identifier for the distribution.

Letting the simulator draw from timing distributions has three main advantages over using raw execution times for each run:

Repeatability: After each event, the simulator determines the amount of time taken by evaluating the event’s distribution at a randomly chosen percentile. Since the sequence of random numbers is already recorded, the execution can be replayed and have the exact same execution time, regardless of how long it actually takes. A side-effect of this property is that the code can even be modified—perhaps to add logging, perform more time-intensive testing, etc.—but still behave as if the system were running the original, unmodified code.

Coverage: By combining timing values across multiple executions, it is possible to produce an execution with timing values never seen in a single run. This allows the simulator to explore more variations in timing behavior, including potentially hard-to-find corner cases.

Malleability: With a timing distribution for each event, it is easy to explore “what if” scenarios with event times. One can simply change the distribution for an event or events in question to see how the average execution duration would be affected. This could be useful for evaluating the effects of a potentially time-consuming algorithmic improvement, buffering strategy, etc., without actually implementing anything.

It is also possible to construct similar distributions for network latency/bandwidth observed on a particular real-world topology. Our simulator does not currently do this; rather, we take a simpler approach that uses randomized latency values. For more details, see Section 6.3.3.

When we have a suitable EDD for our target system, the next step is to quantify what should be deemed an anomaly by using the EDD to supply event

timings. Once again, we use the **Search** algorithm, but this time we pass the system, the stopping condition, our new generated EDD, and another value of N . Using the *execs* queue, we compute the values of the first and third quartiles of the average execution time of each execution. We then compute a lower bound, defined as $Q_1 - 1.5 \times (Q_3 - Q_1)$, and an upper bound, defined as $Q_3 + 1.5 \times (Q_3 - Q_1)$. Execution times outside of this range represent “mild outliers” [MM99] and are deemed anomalous.

The final step in the first phase finds anomalous paths. To do so, we run **Search** with the same parameters as before, except we watch for executions that fall outside our bounds determined in the previous step. In this case, it is reasonable to set N to a fairly large number and terminate the search when we find an anomalous path.

6.2.2 Analysis

Once the model checker identifies an execution with anomalous performance, which often consists of tens of thousands of events, the programmer faces the daunting task of determining the cause of the anomaly. In this section, we discuss our strategies for analyzing executions and techniques we have implemented in the model checker to help focus our search efforts.

One of the key properties of our simulator is that it provides deterministic replay of executions. By recording each event tuple during the search phase, the model checker has a path describing the complete execution, and can later replay this path by executing each event on the appropriate node at the time indicated in the event tuple. To provide consistent executions when replaying a path, the model checker must control all sources of non-determinism. We address non-determinism in event orderings by using a simulated source of time, as discussed above. In addition to the event orderings, real systems often make use of non-determinism *within* the event handlers themselves, such as using randomized algorithms. When executing in the simulator, Mace systems automatically use the deterministic simulated random number generator provided by MaceMC.

By leveraging deterministic replay, the model checker can provide the programmer with a verbose log file containing the event details and system state after

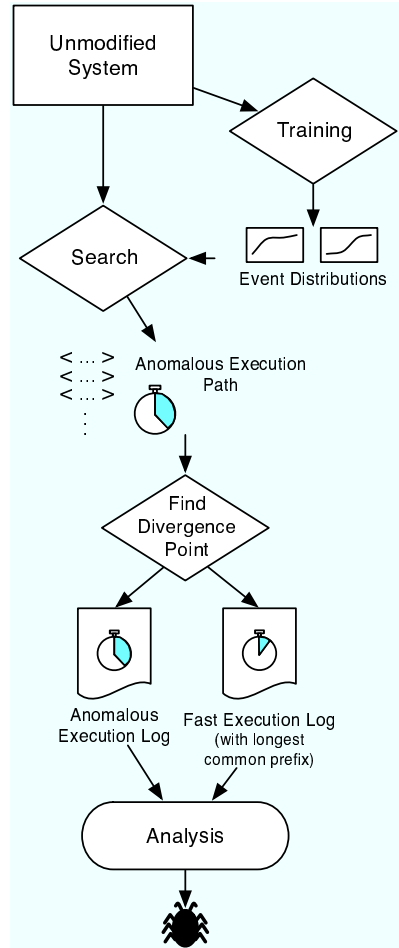


Figure 6.1: System Architecture: first, the unmodified system is run through a set of training runs to determine a set of event duration distributions (EDD). Next, the system, along with the EDD, is fed into the **Search** algorithm, which will produce one or more anomalous paths. We then use the ExecutionSearch tool to analyze the anomalous path, giving us the most similar “normal” path. At this point, any debugging tool can be used to compare the two executions until the source of the bug is found.

every event, a valuable starting point for debugging the execution. While searching for anomalous executions, the simulator runs with all system logging disabled for maximum efficiency. When the model checker finds a violation, it automatically replays the path with full logging. The resulting log contains annotations (i) written by the programmer, (ii) generated automatically by the Mace compiler, which mark the beginning and end of each transition, as well as details of the event, such

as a method invocation call stack with parameters, (iii) produced by the simulator runtime libraries, such as message queuing and delivery and timer scheduling, and (iv) generated by the simulator tracking the execution’s progress, including random numbers generated and the complete state of the simulated node after the each event execution. The verbose logs for the systems we have examined span tens of thousands of events and are hundreds to thousands of megabytes in size.

Verbose log files enable generic debugging, allowing different approaches for analyzing the execution results. For example, one of the authors used a set of scripts he had written for debugging and analyzing live *BulletPrime* runs, after splitting the simulated log file into per-node log files. In contrast, one of the other authors preferred using MDB, the interactive debugger designed for MaceMC, which can navigate long log files and isolating events of interest. Having the simulator generate structured log files along with the custom logging added by the programmer gives the developer the flexibility to choose the most appropriate tools for a given analysis.

Although verbose log files provide a foundation for analyzing an execution, due to the length of many anomalous execution paths and the size of their corresponding log files, our tools are often insufficient for quickly finding the source of a performance problem. Thus, we have sought better techniques for isolating the cause of the performance problems. One important contribution of this work is a technique to automatically determine where in the execution the performance problem becomes apparent. This powerful technique both allows the programmer to essentially ignore a potentially large portion of the log and points to the part of the execution where they should focus their attention. We call the execution event that first exhibits the anomalous performance to be the *divergence point*, and we automatically identify the divergence point as described below.

The insight behind our algorithm is as follows. Up to some prefix point in an anomalous execution, the overall performance of the system is likely fine and exhibits similar overall behavior to other runs with acceptable performance. Even if individual anomalous events take place, later system behavior is sufficient to recover. However, there is some point in the anomalous execution beyond which it becomes impossible to ever match the performance of the good executions. Our approach is to conduct guided random walks from various points in the anomalous execution to determine

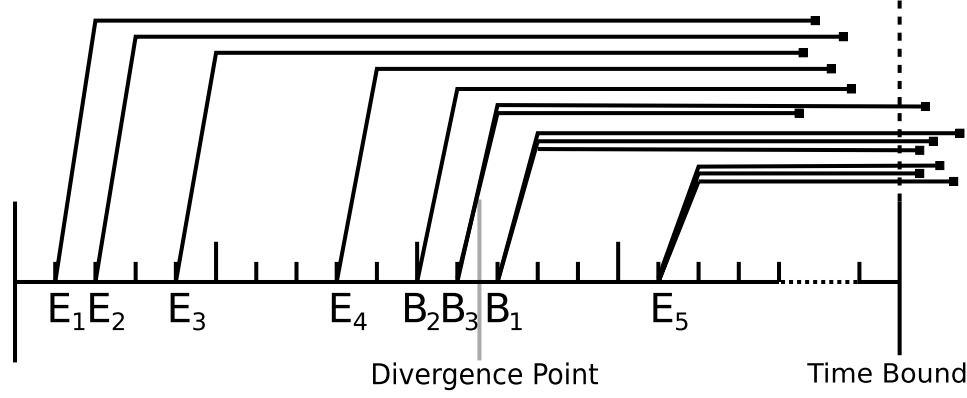


Figure 6.2: We first perform an exponential search ($E_1 - E_5$) to determine bounds for the divergence point, then a binary search ($B_1 - B_3$) to isolate the divergence point.

whether the completion time of all random walks from this point remains anomalous or falls within the good range exhibited by most runs. If a sufficient number of random walks from a given prefix all wind up with an anomalous execution, we have found an upper bound on the prefix. That is, the system diverged to a state where it likely became impossible to ever achieve good performance somewhere before that prefix. By conducting multiple such guided searches as described below, we can precisely bracket this *divergence point* in the execution, tremendously easing the programmer's job in locating the root cause of the performance bug.

We employ the deterministic replay capability of the model checker as well as random executions to perform a search along the anomalous path. As illustrated in Figure 6.2, we begin by initializing the *prefix* length to one event, replaying the portion of the path overlapping with the prefix, and then performing up to k random walks. If one of the random walks satisfies the timing constraint, then we double the prefix length and repeat. Eventually, we reach a part of the path where none of the k random executions satisfy the timing constraint, and thus we have lower and upper bounds on the divergence point. The algorithm then begins the next phase, in which we perform a binary search between the upper and lower bounds. For each round of this search, we once again perform up to k random walks, and if any satisfy the timing constraint then we increase the lower bound, and if none of them satisfy the constraint, then we decrease the upper bound. The algorithm terminates when the

bounds converge on the divergence point—the step indicating the longest common prefix between the anomalous execution and a different execution that will satisfy the timing constraint if a different choice had been made in the divergence point.

6.3 Implementation Details

This section describes several challenges we encountered while adapting MaceMC to search for performance bugs. Although we believe the techniques described in this chapter can be applied to any systems implementation, our performance checker works with systems implemented using the Mace source-to-source compiler and C++ language extensions (§ 3). Recall that Mace structures a system component as a state machine with messages, C++ state variables, and atomic event handlers—implemented as methods in C++—that process the receipt of messages, timers, and application requests. Mace furthermore provides syntax to compose reusable and interchangeable system components into complex distributed systems. The Mace compiler generates C++ code ready to run on live networks by constructing classes and methods to handle event dispatch, serialization, callbacks, timers, etc. By using Mace, we avoid the painstaking task of modifying the implementation to identify blocks of code representing discrete events (e.g., the processing of a message).

6.3.1 Preparing the Test

Users write a simulated driver application to prepare their system for performance testing. The simulated application should initialize the system, perform desired system input events, and check high level system progress with the stopping condition. For example, to look for performance anomalies in a file distribution system, the test driver could have one node publish the file and the other nodes request the file, and use the stopping condition that all nodes have finished downloading the file. The compiled simulated application links with the compiled Mace system object files and Mace libraries. The performance tester executes the system by starting the simulated application to drive the system forward and simulating a distributed environment, loading simulator-specific, deterministic libraries for message queuing and

delivery, supplying the current system time, timer scheduling, and random number generation.

6.3.2 Pending Event Lists

At each step of the simulated execution, the simulator picks an event to fire from a set of *pending events*. In MaceMC, we would construct the set of pending events at every step by querying the system components—the simulated application, simulated network, and simulated scheduler—for their respective subsets of ready events. The model checker would randomly choose the event for the current step from the combined set of all ready events and then discard the set because each system component maintained its events in its state. We chose this approach because of its simplicity—each simulated component constructs the set of pending events from its state at the time when the simulator requests them, which ensures the accuracy of the set of events. We contrast this with the complexity of the alternative we ultimately implemented for our new performance tester, described below.

When we began our performance debugging work, we initially tried using our prior model checker, and we found it was far too slow to allow us to search long enough to discover any performance bugs. When we initially wrote the prior model checker, we knew the approach was not the most efficient, but it worked sufficiently well for the size of the systems we model checked, which usually had less than ten nodes. With the performance tester, we needed to support configurations with hundreds of nodes, and the event picking algorithm was so slow we were not willing to wait to search more than a single path. Thus, we implemented our current approach, in which we assign each event a *start time* and have the simulator maintain a list of pending events each simulated component must update as appropriate. The advantage of this approach is efficiency—before, computing the set of events for the simulated network required performing an $O(n^2)$ operation, while the new algorithm is an $O(1)$ operation—pick the event with the earliest start time. The drawback of the new approach is additional complexity, that mainly stems from the need to remove events from the pending list. For example, consider the example of a service canceling a timer. With the old approach, the timer would be removed from the scheduler state,

so when the simulator queried the scheduler for the set of events, the scheduler would just return all currently scheduled timers. With our current scheme, the scheduler needs to actually remove the canceled timer event from the pending list. As another example of additional complexity, the simulated network needs to remove all pending messages if a socket error occurs. However, the new approach of maintaining an incrementally updated pending event list is two orders of magnitude faster than our old algorithm, and the changes allow our new tool to search executions fast enough to be practical for the systems we need to check.

6.3.3 Simulated Network

Since our performance tester simulates the execution of an entire distributed system on a single machine, it is necessary to simulate a network interconnecting the system. As mentioned in Section 6.2.1, one possible way of doing this would be to gather distributions for all-pairs latency/bandwidth measurements from real runs of the system on a real network. Then, when determining the arrival time for a message, the proper distribution could be consulted.

However, we use a simpler network abstraction in our system. The computed arrival time of each message to be delivered is the sum of three factors: (i) a minimum latency intending to model the delay induced by the speed of light, currently 1ms, (ii) the propagation delay due to constrained outgoing bandwidth between nodes and the number of simultaneous “flows”, and (iii) a random delay taken from a Pareto distribution.

More specifically, we model the bandwidth between all pairs of nodes as 8000 Kbps arbitrarily. To simulate the common case where a node’s first-hop link is the bottleneck, we count the number of active outgoing flows the node has to all other nodes above a certain size, and use this value to divide up the available 8000 Kbps. Thus, if a node has two flows, the latency generated for a new packet in part (ii) above will be based on the time it takes to send the message over a 4000 Kbps link. We assume here all flows with a certain threshold number of bytes in transmission are receiving their “fair share” from TCP. In particular, we make sure to exclude “light”

flows from this equation, because we do not want to equally penalize these flows. In our current implementation, this threshold is set to 300 bytes.

Although our simulation of network timing does not perfectly mirror an 8000 Kbps interconnected topology, we have found that the values it generates are generally suitable for our needs.

6.3.4 Large Log Files

In Section 6.2.2, we discussed the verbose log files the simulator automatically generates after finding an anomalous execution by replaying the path with full system logging enabled. The log files resulting from the performance tester presented us with more scaling challenges in several respects. First, log files required much more storage largely because there were far more nodes in the system. One change we made to help reduce log size was to only log the state of the node that just executed the current step, whereas previously we would log the state of all the nodes, which made inspecting the system state convenient when there were only a handful of nodes, but was too inefficient for larger systems.

Second, in our prior model checking work, a property violation and error typically occurred near the beginning of a path, almost always within the first few thousand steps. As a result, we could truncate the path when generating the log file, which would help keep it a more manageable size. With the performance bugs we have found, this is not the case, in that the divergence point may be anywhere in the execution. Thus, we adapted our log analysis tool to be able to ignore a prefix portion of the log, as well as a truncated part of the log, so we could focus on the portion of the log just surrounding the divergence point. Finally, we modified our tool for generating a graphical representation of the events in the execution to make better use of screen real estate by condensing the graphical portion of the log into a small column, and using the remaining space for text describing the event. This change allowed our event graph to scale from five to ten nodes to up to one hundred.

6.4 Experiences

We ran our performance tester on implementations of BulletPrime [KBK⁺05], Pastry [RD01], Chord [SMK⁺01], and a random tree protocol (RandTree) as described below, and found significant, previously unsolved performance issues with each. In each example, we describe the system being executed, the process we took, and the stopping condition we used for the systems. We then describe the performance issues we found using our tool and the steps we took to improve them.

All the systems we evaluated have been extensively tested in live runs, and in the case of Pastry, Chord, and RandTree, were tested for correctness during our prior work. BulletPrime, though not run through MaceMC, has been tested the most extensively for performance.

In each of the experiments below (except RandTree, which was tested at small scale and resulted in fast executions), the performance tester was configured to use 40-100 nodes. Note that it is impractical to perform any kind of exhaustive search using standard model checking techniques on systems of this size. Executions run in the simulator took anywhere from 8 seconds to 3 minutes. The execution length in real time is sensitive to the complexity of the stopping condition, because it is executed after every step. In the case of Pastry, for example, the stopping condition is $O(n^2 \log n)$ for n nodes in the worst case, which makes testing it very slow for larger n . In general, training the performance tester took between 5-30 minutes for each system.

The time spent searching for anomalous executions varied depending on how hard the simulator had to search, but the bugs we found appeared relatively quickly. The longest part of the performance tester execution was the ExecutionSearch step, which has to run $O(k \log s)$ steps, where k is the number of random paths at each step, and s is the output step. ExecutionSearch ranged from 1 hour to 22 hours to run (in the latter search, 630 executions were performed at an average of 130 seconds per execution). Though 22 hours is perhaps excessive, the search was unattended, and did allow us to ignore nearly 62% of the 22000 events in the anomalous execution.

6.4.1 BulletPrime

BulletPrime is a mesh-based, peer-to-peer file distribution protocol similar in functionality to Bittorrent [bit]. Each node contacts a source node, receives a set of initial peers to join, and then begins downloading a file in parallel from other nodes. Before discussing the bugs we found, there are a few relevant implementation details to discuss. First, BulletPrime uses two transports – one for data and a second for control messages. The data transport is configured to only buffer one block’s worth of data at a time, while the control transport is configured to buffer an unbounded number of messages. However, “Diff” messages, or those that inform a peer about blocks a node possesses, are sent over the data transport. Another necessary component of the protocol is the method for which nodes learn about new peers in the system. BulletPrime is structured on top of RanSub [KRA⁺03], a gossip protocol that periodically delivers a changing random subset of mesh participants to each node.

Stopping Condition. The stopping condition for BulletPrime is straightforward – it is simply the condition that all nodes have finished downloading the file. To test this, we check whether each node has downloaded all of the blocks it expects.

Anomalies. In the first experiment with BulletPrime, we used a setup of 100 nodes downloading a 20MB file. The performance tester found the first anomaly after 134 executions, finding an 11 second execution, which exceeded the 9.5 second upper bound (as computed in Section 6.2.1). Each execution took approximately 18 seconds of real time, and terminated in between 56000-57000 simulator steps.

After examining the times that each individual node took in this execution, we determined there was only one slow node. Once we used the ExecutionSearch tool, we found that the discrepancy was based on the timing of one particular message the slow node sent to one of its peers. Upon further investigation, we realized that in the “good” execution, the slow node’s message caused a Diff to be sent to it successfully, which happened to contain information about two blocks that were never successfully sent by any other node. The slow node was then able to request these blocks from this peer and complete the file download. In the anomalous execution, the message was timed such that the Diff message sent from the slow node’s peer was dropped by the transport because it was already full. As a result, the slow node never learned

about the two blocks, leaving it stuck since it did not know about any peers who had the missing blocks. Eventually, RanSub delivered a new set of candidate nodes, and the slow node was able to join one of them and retrieve the missing blocks. However, waiting on RanSub was responsible for the few second delay, causing the anomalous execution.

The second experiment with BulletPrime used only a 2MB file, but the performance tester found the second anomaly in less than 10 executions. In this case, the anomalous execution took around 5.5 seconds, whereas a normal execution took no more than 2 seconds of simulated time. To debug this case, we once again looked at individual node completion times and found that only a few nodes were slow. This time, we used “traditional” debugging techniques to look for certain things in the log file. The first thing that was obvious from inspecting the slow nodes’ logs was that they did not receive any blocks until approximately 5 seconds into the run, and then quickly retrieved all the blocks. We found that the node did not acquire any peers for the first 5 seconds. The slow node did receive a list of candidate peers from the source when it joined. However, its attempts to join each of them failed because all of the candidates were full. As a result, the node had to wait for RanSub to deliver it a set of new peers, which did not occur for 5 seconds.

Improvements. To fix our first anomalous condition, we simply changed Diff messages to be sent over the control transport instead of the data transport. This eliminated the problem of Diffs being dropped by the transport, and ensured that all nodes received all intended Diff messages.

The second performance problem was based on the fact that when a node is rejected by potential peers, it could have to wait for RanSub to deliver it new ones. To overcome this idle time, we changed the JoinReject message to contain the list of the rejecting node’s peers. Then, when a node receives the JoinReject message, it has a set of other nodes it can try to join.

Note that in both cases, the overall system execution was *correct*. However, just as corner cases in execution can lead to errors, similar corner cases can lead to unexpectedly slow performance. Model checking techniques appear to be well suited to automatically find both types of conditions.

6.4.2 Pastry

Pastry enables nodes to self-organize into a ring structure. Each node in the ring takes an address in a circular address space, and becomes responsible for the address space in the immediate vicinity of its own address. The Pastry protocol organizes the ring to enable routing to any address using a path of no more than $\log(n)$ hops.

The primary functionality we are concerned with is the stabilization of the Pastry network. Pastry’s performance can be measured by how long it takes nodes to finish their self-organization protocol. This protocol includes two basic components. The *active* join component allows a joining node to connect to an existing node and follows a protocol to find out where in the network it should insert itself. This protocol is correct as long as only one node is joining at a time, and in the absence of departing nodes.

To handle multiple simultaneous node joins and departures, a second protocol component involves periodically exchanging information with peers to ensure that each node’s state is accurate. This protocol component can correct a wide variety of errors, mistakes, and dropped messages, and therefore mask many bugs and opportunities for performance improvements.

Stopping Condition. We implement the stopping condition for Pastry to be the time all nodes’ routing information has stabilized to a consistent state. This involves checking that both the leafset information is correct at each node, which is important for correctness and fault tolerance, and that the routing distance between all pairs of nodes is logarithmic in the number of nodes.

Anomalies. During the training of Pastry anomaly conditions, we stopped the training tool early. This was because in the first 6 paths, the average execution times for the 40 pastry nodes (in seconds) were:

$$(60.0073, 60.0061, 60.0085, 40.0051, 20.0369, 20.0313)$$

Variations of this magnitude were immediate indicators of a performance problem. We re-ran the performance tester having it save any execution that took longer than 50 seconds. 22 of the first 40 paths it ran took longer than 50 seconds, the longest taking 100.0044 seconds in 8848 simulator steps. All executions were within a second

of a multiple of 20 seconds, a clue that the execution time was being dominated by the behavior of the system following a timer that fires every 20 seconds.

Anomalies such as these were not unexpected. In earlier experiments, we had observed that our Pastry implementation required either a long stabilization period after first being started, or an extended stagger-start period where nodes are slowly started over the first 30 seconds of the experiment. But our earlier model checker did not detect these as problems because Pastry was still executing correctly, and we did not attempt to extensively debug this performance problem in live executions because of the difficulties of debugging a distributed system, and the knowledge that we could just wait for it to stabilize before conducting other experiments using Pastry.

The ExecutionSearch tool indicated the performance was not terminally bad before step 5681. In both fast and slow executions, all nodes are trying to join at once. This causes nodes to initially believe there are only a few nodes in a small ring, and then they all nearly simultaneously announce themselves as new members of this small ring, largely oblivious to the other arrived nodes. In the ensuing mayhem of the active join protocol, the nodes closest in the address space to the bootstrap node are successful, while the state of other nodes further away depends on the precise order in which nodes are added and removed from the leafset of the bootstrap node.

Unlucky nodes take one or more executions of the periodic protocol to finally correct their state, each time learning about nodes closer to their final position in the overlay.

Improvements. The basic problem is that waiting 20 seconds for each execution of the periodic protocol is a huge performance penalty in the ring construction time. Scheduling the protocol more frequently would help, at the added cost of higher overhead in the common case—such fixes are only required during times of high node churn. An adaptive timer could be used, though its design would likely involve difficult and network-specific tuning parameters.

After exploring a range of options, we settled on the following solution. The problems essentially occur for nodes who are replaced in the bootstrap leafset but know nothing about nodes near them in the address space. Thus, we notify a node with the current leafset when *removing* it from the leafset. When it receives the leafset, it will be informed of several closer nodes to its correct place in the ring.

We note this is also the same information it would receive later when its timer fires, but then it will be more out-of-date and less relevant. It is important the node get this particular version of the leafset, because the later version will only contain nodes close to the peer, not to the evicted node.

After making these improvements, nearly all paths complete in almost exactly 20 seconds, eliminating most of the extra delays. During analysis of this performance problem, we also noted that one of the causes of stabilization latency was due to the fact that we were testing Pastry under a configuration where the transport will only buffer a small amount of network data, resulting in dropped messages. Reconfiguring the transport to use a large buffer, all execution times varied in length from 1.5-2 seconds.

6.4.3 Chord

Chord is an overlay network protocol that provides the same interface as Pastry, though through a slightly different design and implementation. The performance metrics of interest are the same in both Chord and Pastry.

Stopping Condition. The stopping condition of Chord is conceptually the same as for Pastry.

Anomalies. Overall, executions in Chord were faster, in both simulated time and real time, than in Pastry, taking a typical 2.2-3.3 simulated seconds for 80 nodes to join. Our analysis determined that any execution taking longer than 5.1 seconds was abnormally high. On the 7th path searched the performance tester reported an execution that took 7.2 seconds. ExecutionSearch ran for 16 hours and eventually reported an event which, under careful scrutiny, forced an alternate ordering of two messages. When these messages were received in one order, a node A learned about another node B from node C , which it needed to learn about to stabilize its position in the ring. In the other order, node C discards information about node B , and does not include it in its message to A . As a result, node B goes through a lengthy iterative process to learn about node C . This process involves the periodic repair protocol executing, each time bringing node B one step closer to its final place in the

ring. Although the stabilization timer was scheduled for every 500ms, this process had to be repeated about 10 times, causing the overall delay.

Improvements. This example demonstrates how even making a periodic protocol more frequent can lead to anomalous executions. We implemented a similar modification to Chord as we did to Pastry, where any time a node gets a new predecessor, it alerts its old predecessor about its new predecessor. This operation is more efficient than its Pastry counterpart, because it distributes less data. After applying this modification, the new executions now typically take only 1.2 to 2.0 seconds, with only a few longer executions taking 2.7 to 3.0 seconds.

6.4.4 RandTree

RandTree is a random tree protocol we use as a component in many other distributed systems (including BulletPrime). Nodes in RandTree self-organize to form a tree structure, which is used by services such as multicast.

Stopping Condition. We set the stopping condition to be that the nodes have formed a single spanning tree. Each node tracks its parent and children in local state variables, so the property requires that these are consistent across the network and that from all nodes the parent pointer can be followed to the single root of the spanning tree.

Anomalies. We had previously observed through manual inspection of a distributed experiment that under some circumstances, the initial tree joining protocol left the system in a state with multiple disjoint trees, which is only corrected after executing a periodic recovery protocol.

Manual inspection of the logs from the live run did not, however, provide enough detail to fully diagnose what had happened, nor how to fix it. Running RandTree in our original model checker did not solve this problem either, because each execution did *eventually* lead to a consistent spanning tree.

With our new performance tester, however, we were able to identify four types of performance anomalous executions:

- Re-routed messages cause an earlier *remove* message to be received after a later *join* message, with the result that the node trying to join is unsuccessful and moreover is not aware of the failure.

- *join* messages can be received before the tree is ready to process them. Since there is no network error, the sender is unaware the message was ignored.
- *join* messages are redirected from the root of one tree to the root of another tree and handled incorrectly because the root of the tree handles messages specially.
- *join_reply* messages received in different orders lead to different numbers of trees being initially created.

Improvements. For each of these types of anomalies, we were able to develop a modification to RandTree to overcome these performance limitations, and after doing so, users of the application using RandTree report that it is no longer a performance bottleneck for their system.

These improvements were:

- Distinguishing old from new joining attempts by associating each with a join sequence number. Nodes discard any messages from earlier sequences.
- Handling early join messages by storing them in a deferral queue until the node is ready to process them.
- If one root sees a *join* message forwarded through another tree, forward it back to the root for that tree. In the meanwhile, the node that was inconsistent will have corrected the problem preventing a re-occurrence.
- Whenever a node receives a *join_reply* message and it is not already joined, check to see if the root matches that node's own view of the root. If not, that node pro-actively initiate the recovery protocol to merge the two trees.

6.5 Limitations

While this approach to finding performance bugs has been successful and seems quite promising, it is not a panacea. We describe a number of limitations with our approach.

First, the system can only consider the performance of system conditions developed by the programmer. That is, we are in effect running the real system code

with real input. While the system can find corner cases in the performance of the system for a particular set of input conditions, the coverage of the system is only as good as the test cases provided by the developer.

Second, because the performance tester is generating executions based on realistic distributions of event timings, it is less likely to cover as much of the code paths as a traditional model checker. This means it may have to be run separately for different environmental conditions it may be run under (e.g., different network conditions or different distributions of processor speeds, etc.). The alternative approach of considering all possible performance conditions is infeasible. As future work however, we are considering the benefits of additional heuristics to inform the relative performance and distribution of individual events.

Third, while the performance tester is good at finding anomalous executions, the ExecutionSearch technique will be less applicable if there is no clear divergence point. That is, if execution times are evenly spread on the timeline, the divergence point may be only a small delta from the anomalous execution—a similar execution with a small change may perform just enough better to not be considered a mild outlier. To combat this, we use the quartile, and not the outlier bound for the ExecutionSearch execution, but this utility is not yet well-explored.

Fourth, if the system being checked has an algorithmic deficiency that exhibits itself on all executions, this performance tester will not locate it through anomaly detection, since all executions will seem “average.”

Finally, our present implementation has not considered complicated execution search strategies, correlations between individual event timings, network topology models, or event timing distribution methods, because they have not yet been needed. The design supports these possibilities, and we anticipate that additional performance bugs may be isolated with additional detail.

6.6 Summary

We demonstrate how to use model checking techniques to automatically find and isolate performance bugs in unmodified distributed systems code. Our approach involves employing an automated search of a subset of the execution space looking

for runs that perform abnormally with respect to typical executions. Implementing the simulator to faithfully track the time of each node in the system by training it with timing characteristics of actual executions is key to being able to identify these anomalous executions.

Further, once the performance tester has found an anomalous execution, we show how to perform a systematic search for the most similar execution that does not exhibit the performance bug. These two runs, along with an automatically identified divergence point—the step after which it becomes impossible for the execution to achieve acceptable performance—serves dually to direct the developer to a portion of the execution believed to contain the bug, and to attest that the bug does not occur before the divergence point. We have applied this performance tester to four mature systems, finding long-outstanding performance bugs in each. Relative to running experiments on nodes spread across the Internet, or even on a local-area network emulator, we have found that our performance tester significantly simplifies and speeds the task of performance debugging.

Chapter 7

Conclusions and Future Work

Currently, the primary challenge to developing distributed systems is that it is difficult for developers to internally conceive of and manage the complexity of a distributed system. Sources of complexity in understanding include:

- The magnitude and complexity of the implementation code.
- The need to blend algorithmic and mechanical implementation code.
- The concurrent events within and across nodes.
- The uncertainty at each node of the status of peer nodes.
- The dynamic and heterogeneous environment across participants and networks.
- The challenge of multi-node execution replay and analysis.
- The challenge of tracing the causal flow of distributed control paths.
- The challenge of distinguishing between random bad luck, and incorrect or sub-optimal behavior.

This dissertation presents the Mace language, runtime, and toolkit supporting the development of correct, high performance distributed systems. Mace achieves a balance between performance and expressiveness, enabling human and computer analysis of distributed systems without sacrificing performance, proving the hypothesis of this dissertation.

7.1 Contributions

This dissertation makes the following specific contributions:

- Semantic implementation of distributed systems.** Mace supports the development of distributed system through the use of *service objects*, *events*, and *aspects*. What separates these notions from the simply using standard object-oriented programming with aspect-oriented programming is the *distinction* made by the programmer between semantically different things. Object-oriented programming would allow you to say something about all objects, or all objects that derive from a specific type, or all public methods of an object. But there are many different objects, with different inheritance diagrams, and all are not created equal. Similarly, events are like methods, but should be treated specially because they are events of a service object, rather than generic methods. Finally, aspects are important, but when should they be considered? Not at the end of every method, but rather only at the event boundaries. It is the combination of these three programming techniques with the semantic meaning the designer gives them through the language which makes them so useful for building distributed systems in a structured way.
- Detecting and isolating liveness violations.** Prior to MaceMC, it was not known how to test liveness properties in unmodified systems implementations, as their state space was simply too complex. MaceMC presents a novel technique for testing encountered states to see if they are dead states, making it possible to use dynamic model checking to find and isolate violations of liveness that enter a dead state. This technique uses long random executions from each state encountered to ensure that executions make progress towards satisfying liveness properties, and in the absence of progress, isolate the specific step, which we call the critical transition, that prevents progress from being made.
- Automated performance-anomaly testing.** Much effort is expended in logging and more recently structured logging, as a technique for reflecting on problems in observed executions. But less effort is spent on being able to automatically search for and detect performance anomalies. We demonstrate that

an interesting class of bugs, which would cause correctness errors in the absence of a robust implementation, can be observed, automatically detected, and isolated automatically by combining the techniques of a model checker with the reality of a traditional simulator.

- **Practical implementations.** In addition to its research contributions, Mace represents 4 years of practical implementation effort. The Mace toolkit is publicly available, and has been so for several years. It is used by researchers at several universities and labs around the world. The Mace compiler, by itself, generates much of the mechanical code that users would otherwise have to write, and all too often get wrong, and what is more, new features in the compiler affect all Mace implementations. These contributions by themselves make Mace an excellent tool for the development of distributed systems. Additionally, Mace comes with an implementation of the model checker, which easily works with nearly any service implemented in Mace. The Mace development effort also comes with a highly optimized library for logging and networking, and a set of pre-implemented services, including Chord, Pastry, Bamboo, RandTree, Scribe, SplitStream, and a whole host of others. We regularly evaluate new tools and services implemented in Mace to see if they are ready for public release, and do so as soon as it is sensible.

7.2 Future Work

As we develop new distributed systems, we do occasionally run into a class of tedious code that a compiler could simplify for us, and in such occasion, we debate whether to extend the language. But in general, the Mace language has largely stabilized, and does not change often. However, much of the work described in this thesis could be extended in a wide variety of ways. Four of these are summarized in this section.

7.2.1 Static Analysis

Currently, Mace treats event handlers as black-box C++ code. Mace does not try to parse the code, or understand the specific side effects it has. As a result, the Mace implementation is inefficient in many ways, because event-boundary code—e.g. property testing, aspect monitoring, and evaluation of transition guards—must be evaluated at *every* event boundary. Similarly, MaceMC cannot prune the search space or guide the search based on the actual code it is testing. Using static analysis, even a limited amount, could prove useful in increasing the efficiency of a Mace implementation of model checking in MaceMC. Specifically, if we could determine which state variables are referenced and written by each transition, this would allow a variety of compile-time optimizations.

7.2.2 Finding Liveness-Violating Executions

The techniques in MaceMC have been successful at finding violating executions, and in particular are good at isolating the critical transition once a violating execution is found. But, after testing any system for a period of time, the easy-to-find bugs are exhausted, and users have to determine how best to find new bugs. That process may involve changing the system configuration, the test driver, using a prefix execution, or allowing failures further in the execution. The reality is that bounded-execution-depth searching is limited and only finds problems within a certain area of the state space. We are interested in finding new techniques to help the model checker find property violations. One promising approach is to use a different bounding model. For example, we can use bounded-failures or bounded-context-switches, which might be adapted from CHESS [MQ07] to work with causal-paths. Alternately, new techniques that can discover code-coverage may help guide the model checker search to get better coverage of the state space. Finally, the model checker may need to be adapted to support more complex events, such as network partitions, which it cannot effectively explore using exhaustive techniques in exponential state spaces.

7.2.3 Machine-Learning Based Performance Testing

The performance tester has the ability to generate a large quantity of data about different random simulated executions, and to generate an endless supply of such executions. It is therefore an interesting problem to consider how data mining or machine learning might be used to create correlations between what happens during an execution and the eventual performance metric. One example would be that a tree protocol may be particularly sensitive to the root of the tree dying, which could be learned by detecting the delay in forming a spanning tree if a node reset event occurs on the same node that a “notify root” event occurs. The Mace performance tester is an ideal environment for this, because executions can be replayed as necessary, with additional logging to generate the right data. Further, the ability to generate new random executions, and guide them on a similar path to other existing executions will present an opportunity to automate a feedback loop for validating hypotheses generated during the analysis.

7.2.4 Online Monitoring and Debugging

Since we do not expect to be able to fix all bugs and problems before deployment, we want to be able to support automated monitoring and debugging of Mace implemented services. Since the Mace compiler understands the structure of each service, the set of correctness properties defined for the service, and moreover, the structure of most of the log messages for services, it actually could be made to support live debugging of distributed systems through (1) automated light-weight messaging between nodes in the system to evaluate properties and general expressions, and (2) the conceptualization of each node’s structured logfile as a database of tables, and the whole system as a streaming database. Being able to support queries across a distributed system, and also queries that support a hybrid XML-structure/table-structure model will make it much easier to monitor and detect what is happening in a deployed system, while also being more efficient, as structured data will use less disk space and bandwidth, and alleviate string processing.

7.3 Summary

Mace is presently a hotbed of new research. For new systems, this encompasses everything from development of a data center replicated storage system to high-bandwidth video streaming systems. New Mace tools are also being built to ease the challenge of understanding and monitoring distributed systems, and existing tools are being enhanced to make them more effective and efficient. New language features are also being considered to further simplify the construction of distributed systems or otherwise enhance the ability of supporting tools. The security of distributed systems is another area in which Mace research might continue, leveraging compiler support to enhance the ability to thoroughly detect vulnerabilities in distributed systems.

Bibliography

- [ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model checking for real-time systems. In *IEEE Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, Pennsylvania, June 1990. IEEE Computer Society Press.
- [ACH⁺95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
- [AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 74–106. Springer-Verlag, 1992.
- [AHM⁺98] Rajeev Alur, Thomas A. Henzinger, Freddy Y.C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In A.J. Hu and M.Y. Vardi, editors, *International Conference on Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, Vancouver, Canada, June 1998.
- [AMW⁺03] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, New York, October 2003.
- [And88] David P. Anderson. Automated protocol implementation with rtag. *IEEE Transactions on Software Engineering.*, 14(3):291–300, 1988.
- [And05] Jenny Anderson. Malfunction briefly halts trading on big board. The New York Times, June 2nd, 2005. <http://www.nytimes.com/2005/06/02/business/02nyse.html>.
- [Ara07] Villu Arak. What happened on august 16. Skype.com, August 20th, 2007. http://heartbeat.skype.com/2007/08/what_happened_on_august_16.html.

- [BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks*, 14, 1987.
- [BD87] Stanislaw Budkowski and Piotr Dembinski. An introduction to estelle: A specification language for distributed systems. *Computer Networks*, 14:3–23, 1987.
- [BIMN03] Paul T. Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003.
- [bit] Bittorrent. <http://bitconjurer.org/BitTorrent>.
- [BLL⁺96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *DIMACS/SYCON workshop on Hybrid systems III: verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
- [BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Symposium on Principles of Programming Languages*, pages 97–105, 2003.
- [BR00] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR Technical Report 2000-14, Microsoft Research, 2000.
- [BR02] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. In *Symposium on Principles of Programming Languages*, pages 1–3, 2002.
- [Car07] Edward Carpenter. Computer malfunction blamed for caltrain delay. The Examiner, April 24th, 2007. http://www.examiner.com/a-691220~Computer_malfunction_blamed_for_Caltrain_delay.html.
- [CDH⁺00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM.
- [CDK⁺03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony I. T. Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Symposium on Operating Systems Principles*, pages 298–313, Bolton Landing, New York, October 2003.

- [CE81] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, Lecture Notes in Computer Science 131, 1981.
- [CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. Mit Press, 1999.
- [CKF⁺02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *International Conference on Dependable Systems and Networks*, Bethesda, Maryland, June 2002. IEEE Computer Society.
- [CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, 2004.
- [CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 415–426, Ottawa, Canada, June 2006.
- [CW02] Hao Chen and David Wagner. Mops: An infrastructure for examining security properties of software. In Vijayalakshmi Atluri, editor, *ACM Conference on Computer and Communications Security*, pages 235–244, Washington, D.C., November 2002. ACM.
- [DCKM04] Frank Dabek, Russ Cox, M. Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM Conference*, pages 15–26, Portland, Oregon, August 2004.
- [DDHY92] David Dill, Andreas Drexler, Alan Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, Cambridge, Massachusetts, October 1992.
- [DKK⁺01] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Symposium on Operating Systems Principles*, pages 202–215, Banff, Canada, October 2001.
- [DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68, Berlin, Germany, June 2002. ACM.
- [DZD⁺03] Frank Dabek, Ben Zhao, Peter Drushcel, John Kubiawicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In

M. Frans Kaashoek and Ion Stoica, editors, *International Workshop on Peer-to-Peer Systems*, Berkeley, California, February 2003. Springer Lecture Notes in Computer Science 2735.

- [ECCH00] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating System Design and Implementation*, pages 1–16, San Diego, California, October 2000.
- [EL02] David Evans and David Larochelle. Improving security using extensible light-weight static analysis. *IEEE Software*, 19(1):42–51, 2002.
- [FBB⁺97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, Saint Malo, France, October 1997.
- [FPK⁺07] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Symposium on Networked System Design and Implementation*, Cambridge, Massachusetts, April 2007.
- [fre06] Freepastry: An open-source implementation of pastry intended for deployment in the internet. <http://freepastry.rice.edu>, 2006.
- [GAM⁺07] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Symposium on Networked System Design and Implementation*, Cambridge, Massachusetts, April 2007.
- [Gar05] Simson Garfinkel. History’s worst software bugs. wired.com, November 8th, 2005. <http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=2>.
- [GAW07] Simon Goldsmith, Alex Aiken, and Daniel Shawcross Wilkerson. Measuring empirical computational complexity. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 395–404, Dubrovnik, Croatia, September 2007.
- [GLvB⁺03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.
- [God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

- [GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *International Conference on Computer-aided Verification*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
- [GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Spin Model Checking and Software Verification*, volume 2648 of *Lecture Notes in Computer Science*, Portland, Oregon, May 2003.
- [HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
- [HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
- [Hol97] Gerard J. Holzmann. The spin model checker. *IEEE Transactions on Software Engineering.*, 23(5):279–295, 1997.
- [Hol00] Gerard J. Holzmann. Logic verification of ansi-c code with spin. In *Spin Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, California, August 2000.
- [Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer And Reference Manual*. Addison-Wesley, 2003.
- [HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *Software Tools for Technology Transfer*, 2(4):72–84, 2000.
- [JGJ⁺00] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James O’Toole Jr. Overcast: Reliable multicasting with an overlay network. In *Symposium on Operating System Design and Implementation*, pages 197–212, San Diego, California, October 2000.
- [JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, San Diego, California, August 2004.
- [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 Annual Technical Conference*, Anaheim, California, April 2005.
- [Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), 1996.

- [Kin94] Ekkart Kindler. Safety and liveness properties: a survey. *The Bulletin of the European Association for Theoretical Computer Science*, 1994.
- [KKM99] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable tcp in the prolac protocol language. In *SIGCOMM Conference*, pages 3–13, Cambridge, Massachusetts, August 1999.
- [KMC⁺00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
- [KRA⁺03] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhurud, and Amin Vahdat. Using random subsets to build scalable network services. In *USENIX Symposium on Internet Technologies and Systems*, Seattle, Washington, March 2003.
- [KRAV03] Dejan Kotic, Adolfo Rodriguez, Jeannie R. Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Symposium on Operating Systems Principles*, pages 282–297, Bolton Landing, New York, October 2003.
- [Lam02] Leslie Lamport. *Specifying Systems: The Tla+ Language And Tools For Hardware And Software Engineers*. Addison-Wesley, 2002.
- [LCH⁺05] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Symposium on Operating Systems Principles*, pages 75–90, Brighton, United Kingdom, October 2005.
- [LLPZ07] Xuezheng Lui, Wei Lin, Aimin Pan, and Zheng Zhang. Wids checker: Combating bugs in distributed systems. In *Symposium on Networked System Design and Implementation*, Cambridge, Massachusetts, April 2007.
- [Lop96] Crista Lopes. *D: A Language Framework For Distributed Programming*. PhD thesis, Northeastern University, 1996.
- [Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [Maz01] David Mazières. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*, pages 261–274, Boston, Massachusetts, June 2001.
- [McM00] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37((1–3)):279–309, 2000.

- [ME04] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Symposium on Networked System Design and Implementation*, pages 155–168, San Francisco, California, March 2004.
- [Mil89] Robin Milner. *Communication And Concurrency*. Prentice-Hall, 1989.
- [MM99] David S. Moore and George P. McCabe. *Introduction To The Practice Of Statistics*. W.H. Freeman, New York, 3rd edition, 1999.
- [MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.
- [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. In *Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.
- [MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455. ACM, 2007.
- [PACR02] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. In *Workshop on Hot Topics in Networks*, Princeton, New Jersey, November 2002.
- [PLS07] Scott Patterson, Aaron Luccetti, and Randall Smith. After a rough morning, a data backup jolts the blue-chip average. The Wall Street Journal, February 28th, 2007. http://online.wsj.com/public/article/SB117262349120221401-d72GPnU5YK6mXwLtk4z2rnPzCV0_20080228.html?mod=rss_free.
- [Pre03] Associated Press. L.a. hospital computer malfunction disrupts trauma center. ChannelWeb, April 22nd, 2003. <http://www.crn.com/it-channel/18822459>.
- [Pre06] Associated Press. Selling stampede shuts tokyo stock market. MSNBC, January 18th, 2006. <http://www.msnbc.msn.com/id/10900819/>.
- [Pre07] Associated Press. Faa: Computer malfunction causes hundreds of flight cancellations. FoxNews.com, June 8th, 2007. <http://www.foxnews.com/story/0,2933,279613,00.html>.

- [QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
- [RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *International Middleware Conference*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350, Heidelberg, Germany, November 2001. Springer.
- [RGRK04a] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht (awarded best paper!). In *USENIX Annual Technical Conference*, pages 127–140, Boston, Massachusetts, June 2004.
- [RGRK04b] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht (awarded best paper!). In *USENIX Annual Technical Conference*, pages 127–140, Boston, Massachusetts, June 2004.
- [RKB⁺04] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostic, and Amin Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Symposium on Networked System Design and Implementation*, pages 267–280, San Francisco, California, March 2004.
- [RKCD01] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. scribe: The design of a large-scale event notification infrastructure. In *Workshop on Networked Group Communications*, London, United Kingdom, November 2001.
- [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 *Symposium on Networked System Design and Implementation*, San Jose, California, May 2006.
- [SBN83] Michael D. Schroeder, Andrew Birrell, and Roger M. Needham. Experience with grapevine: The growth of a distributed system (summary). In *Symposium on Operating Systems Principles*, pages 141–142, Bretton Woods, New Hampshire, October 1983.
- [SG03] Hemantkumar Sivaraj and Ganesh Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. *Electronic Notes Theoretical Computer Science*, 89(1), 2003.
- [SLW⁺05] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis.

- Acute: High-level programming language design for distributed computation. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (IFCP)*, Tallinn, Estonia, September 2005.
- [SMK⁺01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM Conference*, pages 149–160, San Diego, California, August 2001.
- [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 Transactions on Networking*, 11(1):17–32, 2003.
- [USW01] Jeffrey Foster Umesh Shankar, Kunal Talwar and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, Washington, D.C., 2001.
- [VYW⁺02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeffrey S. Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.
- [WCB01] Matt Welsh, David E. Culler, and Eric A. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, Banff, Canada, October 2001.
- [Wes86] Colin H. West. Protocol validation by random state exploration. In *IFIP WG 6.1 International Workshop on Protocol Specification, Testing, and Verification*, 1986.
- [XA05] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In *Symposium on Principles of Programming Languages*, pages 351–363, 2005.
- [YTEM04] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors (awarded best paper!). In *Symposium on Operating System Design and Implementation*, pages 273–288, San Francisco, California, December 2004.