UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Query-based Debugging of Distributed Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Ryan Evans Braud

Committee in charge:

Professor Amin Vahdat, Chair
Professor Alin Deutsch
Professor George Papen
Professor Ramesh R. Rao
Professor Alex C. Snoeren

2010

DEDICATION

To my parents, who have always been there for me.

## EPIGRAPH

*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*
—Brian W. Kernighan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Amin Vahdat, for taking me on as his student. I have learned a lot from him over the years I have been here, including how to look at the glass as half full and not half empty. I appreciate his willingness to stick with me and see me through.

Second, I want to thank all of the friends I have made here. I am not going to name names, but you should know who you are. They have definitely made graduate school a more enjoyable experience. Without them, I don't think I would have been able to stick with it.

Also, I've had the great opportunity to work with some of the brightest people I've ever known while here. Thanks to my co-authors of papers both published and unpublished. I have learned much from each of you as well.

Finally, I'd like to thank the wonderful members of CSE help, the sysnet admins, and the rest of the staff. They have made dealing with building issues, hardware issues, and bureaucratic issues much less painful. I hope over the years UCSD keeps its high standards for its members of all three of these areas.

Chapter 1, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 2, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 3, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 4, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 5, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 6, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

Chapter 7, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

## VITA

| | |
|---|---|
| 2004 | B. S. in Computer Science *cum laude*, University of Maryland, College Park |
| 2004 | B. S. in Mathematics, University of Maryland, College Park |
| 2007 | M. S. in Computer Science, University of California, San Diego |
| 2010 | Ph. D. in Computer Science, University of California, San Diego |

## PUBLICATIONS

Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala, "Finding Latent Performance Bugs in Systems Implementations," *Foundations of Software Engineering (FSE)*, 2010.

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

Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat, "Remote Control: Distributed Application Configuration, Management, and Visualization with Plush," *Large Installation System Administration Conference (LISA)*, 2007.

Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat, "Mace: Language Support for Building Distributed Systems," *Programming Languages Design and Implementation (PLDI)*, 2007.

Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud, "Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse," *Computer and Communications Security (CCS)*, 2005.

Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat, "Maintaining High Bandwidth under Dynamic Network Conditions," *USENIX Annual Technical Conference*, 2005.

Suman Banerjee, Seungjoon Lee, Ryan Braud, Bobby Bhattacharjee, and Aravind Srinivasan, "Scalable Resilient Media Streaming," *International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2004.

Rob Sherwood, Ryan Braud, and Bobby Bhattacharjee, "Slurpie: A Cooperative Bulk Data Transfer Protocol," *INFOCOM*, 2004.

ABSTRACT OF THE DISSERTATION

**Query-based Debugging of Distributed Systems**

by

Ryan Evans Braud

Doctor of Philosophy in Computer Science

University of California, San Diego, 2010

Professor Amin Vahdat, Chair

One of the most challenging aspects of debugging distributed systems is understanding system behavior in the period leading up to a bug. Since traditional debuggers such as `gdb` are not well suited to distributed system debugging, developers often resort to annotating their code with log statements and then writing one-off scripts that perform ad-hoc searches through the logged data.

To improve this cumbersome process, we propose that the state of a distributed system execution should be programmatically and interactively available for postmortem analysis. We observe that the three defining properties of entries in a distributed system's log are "time," "node identifier," and "event type," and treat the log as a logical cube with these dimensions. By exploiting the structure of this *state matrix*, developers can use a high-level query language to efficiently extract information instead of manually inspecting log files or writing log processing scripts.

In this dissertation, we describe the debugging process based on a query-oriented approach. We begin with an introduction of the state matrix abstraction and show how it can capture useful properties of distributed systems' executions. We then present NYQL, an object-oriented query language operating over the contents of the state matrix and describe one possible implementation as a translation to SQL queries executed over a relational database.

Next, we present an implementation of a logging system that generates queryable logs in MACE, a source-to-source translator and library for building distributed systems. We present techniques for mitigating the logging overhead by giving NYQL queries to the MACE translator and show that in many cases queries can be resolved in a few seconds. We then demonstrate how using NYQL simplified debugging a handful of bugs in two different distributed systems.

Finally, we extend our logging techniques to systems without source-to-source translators by developing two general-purpose libraries — one in C++ and one in Java. We describe the differences between all three systems in terms of functionality and ease of use and then conclude with some future directions for distributed systems debugging.

# Chapter 1

# Introduction

Since its birth in the late 1960's, the Internet has seen exponential growth. It has evolved from a simple network of a few computers to one approaching a billion hosts [17]. The Internet, and more specifically, the World Wide Web, has become so ingrained in our daily lives that it is hard to imagine life without it. We now depend on sites like Google, Facebook, Amazon, and many others to inform us, entertain us, and let us share information with each other.

Large sites like these can do business only as often as they are available to their customers. Any downtime is potentially lost revenue, and often additionally results in public outcry from thousands of customers. For example, when Facebook went down in September of 2010, the phrase "Facebook down" was searched more than seven times more frequently than average on Google [7]. Many people who use Facebook as part of their business strategy were affected, including all of the companies who advertise on Facebook. However, Facebook is not the only company who experiences problems with site reliability. Gmail experienced six outages in just eight months, which was dubbed "the great Gmail outage of 2009" [11]. In addition, Amazon experienced three hours of downtime in June of 2010, which may have cost the company upwards of $9 million in revenue [1].

Although availability is incredibly important, it is not sufficient for success. User perception of any online service is also greatly affected by the service's reliability. In other words, the service must not only be available, but also behave as expected. If a user attempts to buy a product on Amazon and is greeted with an error message when they try to add it to their cart, they may eventually take their shopping dollars elsewhere. Similarly, if messages sent from Gmail bounced frequently, or searches for popular terms on Google returned irrelevant results, Google's reputation would suffer.

Unfortunately, building a website with even four nines of availability that can handle hundreds of thousands of clients a day — or more — *correctly* is no simple task. To handle the incredible demand, large sites rely on *distributed systems*. Instead of hosting a site's functionality on a single computer, it is instead broken up into many smaller services and split across potentially thousands of machines. These services provide a diverse set of features, including lock services [32], distributed filesystems [46], scalable computation platforms [40, 21], and distributed storage systems [34, 41, 28]. Since functionality at any site is broken up into many such smaller services, it is often imperative that *all* systems be running correctly for the site to perform as expected. A failure of any one of the component systems leads to a degraded or broken user experience.

There are two types of failures that affect these systems: hardware failures and software failures. Hardware failures cannot be avoided, as any mechanical system will eventually stop working. Failover, replication, and quorum-based consistency protocols are just a few of the techniques these systems may employ to handle these types of failures. Unfortunately, failure handling and recovery add complexity to already-complex systems, making them more difficult to reason about and test.

On the other hand, software failures occur as a direct result of human error, misunderstanding, or unexpected operating conditions, and are typically known as "bugs." These bugs fall into two categories: correctness bugs and performance bugs.

As the name implies, programs with correctness bugs fail to complete their desired tasks correctly. These kinds of bugs can manifest themselves in many ways, from causing a program to crash, to more subtle effects like computing an incorrect intermediate value, ultimately leading to unexpected behavior.

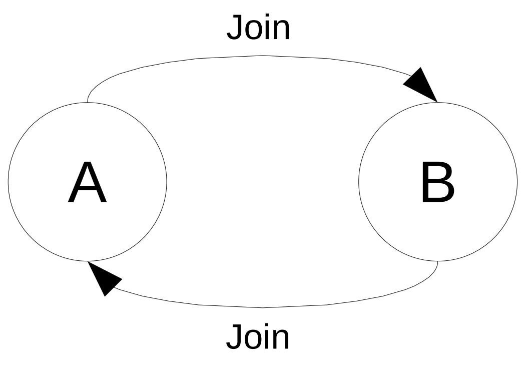
On the other hand, programs with performance bugs do not crash or compute incorrect results. Instead, they often take longer than expected to complete a desired task. For example, a node in a distributed system may spend more time on the critical path than intended, perhaps due to unexpected lock contention or filesystem access. As a result, the processing of each request may complete correctly, but take longer than expected, reducing the throughput of the system overall. Performance bugs may also arise as the result of inefficient protocol design. For instance, if a protocol uses all-to-all communication, it will often end up much slower than a protocol with a more conservative communication strategy, due to the overhead required to send and process all of the extra messages.

Since distributed systems used by large websites perform mission-critical functionality, it is important that they be free of correctness *and* performance bugs. However, it is impossible to write a completely bug-free system. These systems are incredibly complex, consisting of tens to hundreds of thousands of lines of code, and since they are written by humans, errors are inevitable. In fact, according to Code Complete, the industry average for bugs is 15 to 50 per 1000 lines of code [60]. As a result, developers often go through extensive testing procedures such as unit testing [23, 18] to try and detect as many bugs as possible.

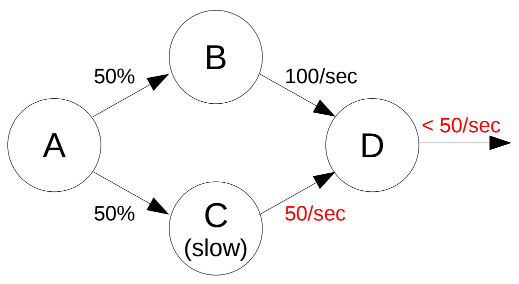
## 1.1 Debugging Challenges

Bugs would not be such a problem if they were easy to locate and diagnose. However, *debugging*, the process of correcting known bugs, is notoriously hard. Although there are many books on the subject [25, 76], debugging is still seen as a black art. Since so many different kinds of programming errors are possible, it is impossible to apply a small set of techniques to all situations. In addition, many properties of distributed systems themselves make them difficult to debug.

Since distributed systems are composed of many interacting nodes, the node a bug is discovered on is not necessarily the node where the bug occurred. Incorrect values may be propagated from node to node, until at some point a node behaves correctly — but with bad input data — which exposes a bug. In other cases, bugs can occur because developers fail to consider all the different possible message interleavings. The asynchronous nature of the Internet makes it difficult to reason about when and in what order messages will be delivered on different nodes. As a result, a node may receive message *B* at a time when the developer thought it should only receive message *A*, leading to incorrect behavior.



(a) Join Loop Problem



(b) MapReduce Performance Problem

**Figure 1.1**: Possible Bug Scenarios

As an example, consider the two nodes depicted in Figure 1.1(a). Suppose these nodes are part of a system where the goal is to form a tree, which can then be used to efficiently disseminate data, etc. To join, a node simply needs to know about any other node in the system. Even in the simplest case with two nodes, implementing the joining procedure correctly is non-trivial. When two nodes try to join each other with the goal of forming a tree, one of them must be the parent and one must be the child. If one node joins the other before the other attempts to join, then everything will be fine. On the other hand, if both nodes join each other at the same time, they must have some way of deterministically deciding who will be the parent and who will be the child. As more nodes are added to the system, the number of possible message orderings increases exponentially, making the correct behavior difficult to reason about.

In order to debug problems like these, the developer has to trace messages across potentially many nodes, inspecting the state on each node along the path, trying to discover the source of the bad data. In our tree example above, the developer would likely need to trace join messages and their responses, keep track of the parent and children of each node, and try to discover the point at which a node behaved incorrectly. In essence, since the developer does not know exactly what to look for, they are stuck with what amounts to looking for a needle in a haystack.

While correctness bugs can be quite difficult to debug, performance problems can be just as difficult, if not more so, to explain. Developers often need to compare timing characteristics of various parts of the system to search for bottlenecks. This process either requires adding code to measure execution timing at different granularities, or recording the times functions are called and return, and then computing various timings offline. In other cases, developers need to inspect communication patterns across nodes to determine whether the expected messages are being sent. They may also need to count the number of messages of each type that are sent from each node, or how many are sent in a specific time interval, etc.

As an example performance problem, consider the distributed system shown in Figure 1.1(b), which represents a simple instance of MapReduce [40]. Suppose node *A* is the master, and is able to perfectly split its work onto nodes *B* and *C*. If nodes *B* and *C* are similar, they should be able to perform the same number of *map* operations per second. However, in this case, node *C* is slow, and is only able to perform half as many operations per second as *B*. This situation could occur for a number of reasons, including *B* and *C* having different hardware, extraneous background processes unintentionally running on *C*, or some deficiency in the mapping process which makes the keys mapped to node *C* take longer to process. Since node *D* has to wait until it has received all of the values from *B* and *C* before it can start, throughput of the entire system is limited by *C*.

This example illustrates one of the challenges in dealing with performance problems. Even if the system is flawlessly implemented, other outside factors can still cause performance degradation. These situations need to be investigated all the same, until the source of the problem can be pinpointed and corrected.

### 1.1.1 Common Approaches

Although many standard debuggers such as gdb [8], DDD [4], the Eclipse debugger [6], and the Visual Studio debugger [5] exist, they are ineffective at debugging correctness bugs in distributed systems since they cannot single-step across machine boundaries. Even if these debuggers *could* step across machine boundaries, they would still be ineffective at debugging performance problems since the performance of the system as a whole must be observed. Executing the application one step at a time will not shed any insight on performance issues.

As a result, developers all too frequently resort to adding logging statements to their code, re-running the system, and then poring over the resulting log files in an attempt to understand what the system was doing in the period leading up to a bug. Although tedious when applied to a single process, this approach is especially difficult when applied to distributed systems.

Debugging typically begins with a simple search for the occurrence of a specific log message, indicating the presence of a bug. From there, developers search backwards, attempting to gain some insight into what went wrong. In order to do this, they need to trace message propagation across nodes, which involves jumping back and forth between multiple log files, and inspecting the node's state at each step.

Tracing messages between nodes is tedious to do by hand, especially when message chains are long or branch multiple times. Developers may need to keep notes on the series of messages they have traced just so they do not get lost. Another approach is to write a script to extract the messages, but even a filtered list of send and receive events may be difficult to trace. In addition, developers may need more information surrounding any of the send and receive events, such as what the node's state was at the time, or what other function calls were executed that led up to the event of interest. Any information not extracted by the initial script must then be fished out separately from the logs and matched up with the messages.

In addition to tracing messages, developers often come up with a series of questions they would like to answer in order to discover where their code went wrong. For example, was function $A$ called before function $B$ on node $X$, or after? How many times was function $A$ called between time $t_1$ and $t_2$? Was the second parameter to function $A$ ever less than 0? Was there an entry in node $A$'s map $M$ for node $B$ when node $A$ tries to send a message to node $B$? An ad-hoc script executed over the system logs is typically designed to answer each question. Each time the developer wishes to investigate something new, they will either write a new script, or expand an existing script to be more flexible.

### 1.1.2 Scripting Limitations

As bugs develop and are fixed, developers end up with a growing toolbox of debugging scripts, each designed to summarize and extract some subset of information present in their application's log files. This approach has a number of limitations.

#### Inefficiency

Since scripts operate over unstructured collections of log files, they almost always are required to process each log file in full from the beginning to produce their output. As a result, the log files may be read through many times in the debugging process, potentially causing even simple scripts to take longer than necessary if the logs are sufficiently large.

#### Complexity

Often times, developers want to ask questions that cannot be answered by single lines of the log files. Instead, the answer is split across multiple surrounding lines and must be pieced together. For example, determining whether a map has a certain key at the time a particular message is sent to a node first requires locating the places where the message is sent to the target node, and then checking previous lines containing the node's state. To write this script, the developer now has to build a reasonably sophisticated log processor, keeping track of the current node state when a function is entered, etc.

Unfortunately, many kinds of questions will require similar but ultimately different log processing functionality. Common functionality can be abstracted away into their own scripts, but the complexity of the log processing framework grows considerably as additional functionality is required. As a result, developers are left with complicated scripts that are difficult to understand and maintain. Passing the scripts off to another party for debugging will likely require detailed instructions on how the scripts can be used, indicating what each of them does and what parameters they require.

**Brittleness**

Log processing scripts are also inherently brittle, since they operate directly on a system's log files. Since log files are simply lines of text, these scripts often make assumptions about their format. For example, they may select out certain columns of text or use regular expressions to match specific patterns. Unfortunately, both of these techniques will tend towards error over time.

Logging statements may be modified to include more information, or shortened to be more concise. In either case, column numbers will change and cause any scripts using them to break. In addition, regular expressions may either fail to match the intended lines, or they may match new logging statements unintentionally. When the now-broken scripts produce unexpected output, developers must spend time debugging the debugging scripts themselves, adding another level of complexity to debugging the original application.

### 1.1.3 Summary

In sum, the iterative loop of forming a hypothesis about system behavior, asking a question, gathering the necessary subset of system state to answer the question, and analyzing the resulting data is too long and labor-intensive. All but the simplest of questions require either painstaking manual inspection, or complicated scripts to extract the necessary information. These scripts can be difficult to understand, difficult to share, and difficult to maintain as the system evolves.

## 1.2 Our Approach

Our key observation is that the entire state of a distributed system should be available for interactive analysis through a simple, high-level language. We propose that both the high-level query language and the program compiler cooperatively understand a shared *model* of system state such that the vast majority of necessary log information may be automatically generated by the compiler. Our model and query language abstract away the details of log file formatting, allowing developers to focus on interacting with the model of the data, rather than with the raw data itself. The query language provides developers with a powerful tool to gain a greater understanding of how their systems are behaving.

This dissertation attempts to prove that developers can find bugs in distributed systems more quickly than with traditional techniques by treating a system's output and state as a logical database and then using a domain-specific query language to extract information.

## 1.3 Contributions

This dissertation makes four main contributions. First, to support our goals of interactive log analysis, we present a new abstraction for interfacing with distributed system state. Rather than a collection of text strings, the contents of logs will be a collection of *objects*, which can then be manipulated by a high-level query language. This abstraction allows developers to log objects directly without having to worry about how the logs are physically formatted or stored on disk. The debugging process may then proceed around a query language based on objects, rather than through unformatted text processing.

Second, we introduce a mechanism for controlling the inherent tradeoff between logging and system performance. By using a set of target queries, we show how we can reduce logging overhead and improve system performance by limiting the set of logged objects.

Third, we present an implementation of a system that realizes the above support. The vast majority of necessary log information can be generated automatically by a source-to-source translator, and a simple structured logging primitive allows developers to add additional necessary annotations. We also compare the functionality of our translator-based system to two stand-alone logging libraries implemented in `C++` and `Java`. We then present a new query language centered around our object-based abstraction of log file data and show how it can be used to extract objects from our log model. We validate its utility by describing the techniques we used in finding four latent bugs in two different, complex distributed systems.

Finally, we present a comprehensive debugging environment that combines a query-based interface with a visual event graph of system events. This interface allows users to trace messages, view communication patterns, and inspect any event in the system in a simple, intuitive manner. We describe one user's experiences using this debugging tool to debug two bugs in her own research project.

## 1.4 Organization

In Chapter 2, we give some additional background about the debugging process, its relation to software testing, and its close ties with logging systems. We also compare our work to related work in the areas of program testing, tracing libraries, debugging, and query languages.

We introduce our object model for log file data, called the *state matrix*, in Chapter 3. We also present the design and implementation of our object-oriented query language, NYQL, as a translation to SQL queries that are then executed over a traditional relational database.

In Chapter 4, we describe the implementation of our logging subsystem in the context of MACE [49], a source-to-source translator and set of libraries for building distributed systems. We also present a number of query-based log configuration techniques, a binary rewriting technique, and a probabilistic path logging technique for controlling the tradeoff between execution speed and amount of program information captured. Finally, we evaluate the performance of our query-based approach, including log file preparation time, database population time, and query execution time. We also show how our log customization techniques are able to increase the performance of a CPU-bound application by eliminating subsets of objects from appearing in the log.

Chapter 5 is dedicated to our debugging experiences. We describe the process of using queries to help locate and fix a handful of bugs in two distributed systems. We also introduce our comprehensive debugging environment and describe its use in fixing bugs in another student's research.

In Chapter 6, we describe two new stand-alone logging libraries aimed at replicating as much of the functionality as possible of our MACE logging subsystem. We show that a `C++` library falls short in a number of areas without a source-to-source translator, but a library implemented in `Java` is able to retain almost all of the benefits of MACE logging.

Finally, we conclude and present directions for future research in Chapter 7.

Chapter 1, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

# Chapter 2

# Background

In this chapter, we cover some background on the debugging process, including its close relationship with logging libraries. We then provide an overview of related work in the field.

## 2.1 The Debugging Process

In the last chapter, we introduced *debugging* as the process of fixing bugs in computer programs. However, debugging is not the only process involved in removing bugs. Before bugs can be fixed, they must be somehow be discovered.

*Software testing*, or just *testing*, is the process of rigorously attempting to discover bugs in computer systems. Although our focus in this dissertation is on bug fixing and not bug finding, it is important to understand the distinction between the two and how debugging fits into the larger picture of making distributed systems more robust and correct, which is the ultimate goal.

As systems are built, developers test their software in various ways, in an attempt to gain faith that the systems work as expected. Regression tests, unit tests [23, 18], and even software design methodologies such as extreme programming (XP) [29] are just a few tools developers may employ to help catch bugs as early as possible. However, the goal of these testing frameworks is only to prove that bugs exist, by getting the program to a state where an invariant unexpectedly fails to hold.

Unfortunately, like debugging, testing distributed systems is much more difficult and complicated than with single-process applications. Developers often start with a simple setup, using as few nodes as possible. Once simple cases work as expected, they will start adding nodes and testing more complicated scenarios. When the system works at a reasonable scale, developers may spend time tuning the system to improve performance, if necessary. Developers may also start testing failure cases, which, with traditional techniques, require complicated testing applications to fail and restart nodes at different times. Alternatively, they may use model checking techniques, which we discuss in Section 2.3, to explore a wide range of usage scenarios without having to test them all by hand.

Even with extensive testing, bugs always get past testing phases because as systems gain complexity, it becomes impossible to test all of the different ways the systems may be used. As a result, programs may crash, assertions left in the code may fail, unexpected exceptions may be thrown, or users may notice unexpected behavior, all indicating the presence of a bug. No matter when a bug is discovered, whether it is during initial testing or after the system is deployed, discovering a bug is only the first step in eliminating it. This is the boundary between testing and debugging — testing finds bugs, the debugging process aims to fix them.

## 2.2 Logging for Debugging

As mentioned in the previous chapter, being able to successfully debug a distributed system is often critically dependent on the content of log files which detail the system's actions. If the logs do not contain enough information to help the developer diagnose what is going wrong, the developer will not be able to fix the problem. As a result, it is important that the logging library they use is as powerful and flexible as possible. In this section, we describe a set of features we believe to be important for debugging distributed systems. Since our approach is also log-based, the goals and criteria put forth in this section greatly influenced the design of our logging system, which we introduce in Chapter 4.

### 2.2.1 Important Log Types

Besides basic user-generated log statements, there are four additional types of logging messages we believe a good logging system should support to enable thorough distributed systems debugging.

```
func1(x=5, y="hello")
func2()
func3(param=[3, 4, 6])
func4()
```

(a) Flat Function Call Logging

```
func1(x=5, y="hello")
  func2()
    func3(param=[3, 4, 6])
  func4()
```

(b) Nested Function Call Logging

**Figure 2.1**: Flat vs. Nested Function Call Logging

#### Function Call Logging

The first, most basic logging type a library should support is function call and return logging. This type of logging is important so developers can see how control flow progresses through their code. Function call logging should also include parameter contents and return values if possible. It is important that the logging system be able to handle function return and not just function call, so that the sequence of function calls is properly nested in the log. Without nesting, a sequence of function calls is much less useful.

For example, Figure 2.1(a) shows a hypothetical series of flat function calls and their parameters, while Figure 2.1(b) shows the correctly nested version. The nested version enables developers to tell which functions call which other functions, as opposed to the flat version, which simply conveys which functions were called.

#### Event Boundaries

Second, we believe it is useful for a sequence of function calls to be broken up into logical units, each corresponding to some specific processing task. These tasks

```
event start
func1()
  func2()
  func3()
event end
event start
func4()
  func5()
    func3()
event end
```

**Figure 2.2**: Example Event Demarcation

might correspond to handling a network message, executing a timer callback, or dispatching a function from a program's main event loop. In the log, we can demarcate each task with "start" and "end" messages. Thus, all of the log messages that are generated between each "start" and "end" pair belong to the same logical task, or *event*. Figure 2.2 shows an example set of log messages enclosed by "start" and "end" markers.

With these markers, developers can focus on one event at a time, and skip over events that are of no interest to them. Each log statement should be between exactly one pair of event markers. No log statements should be "outside" event markers, or log processing scripts will have to both consider event and non-event lines, which complicates their implementation.

### Message Tracing

Distributed systems typically send lots of messages over the network. Thus, the third thing a good logging system should do is make it easy for developers to trace network messages. That is, for each message, a developer should be able to find when and where it was sent from, where it was sent to, and when it was delivered. It is also frequently useful to be able to inspect the contents of each message. In order to enable message tracing, the logging system should generate unique identifiers that can be propagated with each message sent. These identifiers, along with "message sent" and "message received" logging functions, enable developers to track each message in the log.

```
event start
start()
  send(m=HelloMsg(id=1, text="hello"),
       dst="10.0.0.2")
event end
event start
receive(m=HelloMsg(id=2, text="hello"),
        src="10.0.0.2")
  send(m=GoodbyeMsg(id=3),
       dst="10.0.0.2")
event end
event start
receive(m=GoodbyeMsg(id=4),
        src="10.0.0.2")
event end
```

(a) Node 1

```
event start
start()
receive(m=HelloMsg(id=1, text="hello"),
        src="10.0.0.1")
  send(m=HelloMsg(id=2, text="hello"),
       dst="10.0.0.1")
event end
event start
receive(m=GoodbyeMsg(id=3),
        src="10.0.0.1")
  send(m=GoodbyeMsg(id=4),
       dst="10.0.0.1")
event end
```

(b) Node 2

**Figure 2.3**: Example Log Files That Support Message Tracing

Figure 2.3 shows an example of log snippets on two different nodes that use message tracing. Here, we see node 1 sends a *HelloMsg* to node 2 with *id* 1, which arrives on node 2. Node 2 sends back its own *HelloMsg* with *id* 2. Node 1 responds with a *GoodbyeMsg* with *id* 3, which node 2 receives before finally sending its own *GoodbyeMsg* with *id* 4. We note that the actual log format shown for message sending and receiving is not important, as long as there is enough information to track each message.

**Causal Paths**

Building on message tracing, the fourth component of a powerful logging library for debugging distributed systems is *causal path tracing*. Other debugging systems such as Pip [66] and Project 5 [26] have found path-based analysis to be an important debugging tool. In these systems, a path can be thought of as all of the code that is executed across multiple machines as a result of a single event. In addition to generating a message identifier for each message, the logging system should generate path identifiers which are sent along with every message as well. For a more detailed description of causal paths, see Section 3.1.3.

Of course, there are many other things developers often wish to log, including parts of program state, timing information, or control flow information at a sub-function-call granularity, just as a few examples. A good logging library should also make it easy for developers to log whatever other information they would like.

### 2.2.2 Usability Issues

Aside from the set of events that a logging subsystem enables developers to capture, there are also a number of usability issues that affect how well-suited a logging subsystem is for debugging. These issues can be broken down into four main areas: customization, ease of use, log format, and performance. We cover each of these in turn.

#### Customization

Log customization refers to the control a developer has in selecting which logging statements execute. Controlling which logging statements appear is important for a few reasons. First, developers often desire to run their systems with various amounts of logging enabled. For instance, when the code is thought to be mostly bug-free, they will want very few logging statements enabled so the code can execute as fast as possible. On the other hand, when debugging, they will want control over which logging statements appear in the log, depending on the particular problem they are debugging.

The simplest form of log customization can be thought of as commenting out code by hand or inserting new logging statements. If a developer does not want certain log statements in a particular execution of their system, they can comment them out and recompile the code. Unfortunately, this approach requires manually commenting or uncommenting each logging statement the developer wishes to change, which can be quite tedious.

Instead, most logging systems rely on a set of *log levels*. Each logging statement is tagged with a level, indicating the severity of the message. When the code is run, a logging level is chosen. A log message is only actually generated if the log level given to it is less than or equal to the level picked by the developer at runtime.

In general, the more control a logging system gives to the user over which logging statements to include and which to exclude, the better. Runtime control is also generally preferred to compile-time control because it is more flexible and does not require the user to recompile their code when the logging is changed. On the other hand, this flexibility usually comes with a small performance penalty to check the logging level of each statement at runtime.

### Ease of Use

Another important aspect of logging subsystems is how easy they are to use. Ease of use is mostly determined by the logging system's API, and is a measure of how easy or difficult it is for a developer to log the kinds of information they need. For example, standard `printf`-style logging can both be thought of as easy and difficult to use, depending on the context. If the developer is only interested in logging string messages, it is hard to get much simpler than a language's default print statement. On the other hand, if a developer wants to log more complicated messages, such as the contents of an array, or of an associative map, `printf`-style logging becomes more difficult. In these cases, the developer has to iterate over the contents of the array or map and print each element separately, changing a single logging statement into multiple lines.

As a result, `printf` logging becomes rather difficult to use to log statements of any real complexity. Since our goal is to develop a logging system that is compatible with our object-oriented query language, we need to be able to handle data types like arrays and maps without the need for multi-line logging statements. In addition, we also need to make it as easy as possible to do event logging, message tracing, and the other features we described above.

### Log Format

The third usability issue that affects a logging subsystem is the formatting of the log file itself. The log files that we care about exist solely to aid developers in finding and correcting bugs. These log files are often too long and complicated to be inspected by hand, so they must be amenable to machine processing. A log file's formatting directly affects how easy the log processing step can be done.

The most important aspect of a log's formatting is consistency across all logged statements. Consistency makes it easier for a log processing script or program to parse each line, without having unnecessary special cases for each different type of log statement. One important benefit about our approach is that since we are changing what developers interact with, from log files themselves to an abstract model of a collection of objects, the log format that we choose becomes unimportant to developers since they will never see it.

### Performance

Finally, performance of a logging system is also of critical importance. Since each logging statement a developer adds only serves to slow down the execution of the program, it is critical that the logging overhead be as low as possible. Low overhead is especially critical when debugging performance problems, since the slowdown added by logging statements may change the timing characteristics of the original program, which can mask the symptoms of the problem.

## 2.3 Related Work

Our work is related to a wide range of research areas, including testing techniques, debugging infrastructures, tools for logging, and data visualization. In this section, we outline related work in these areas and compare them to our approach. We begin with techniques for program testing, since this is a complementary area to our work.

### Query Languages and Translation

In spirit, our data model is most similar to that of OQL, the Object Query Language [36]. Queries in OQL operate on an object-oriented database, similar in structure to ours. However, objects in OQL are closer to objects in a high-level programming language in that they support methods and inheritance, while ours are closer to `structs` in

C. In addition, there are no publicly available implementations of databases that support OQL, so we could not rely on it as a query language without providing an implementation of it ourself.

Shanmugasundaram et al. [67] present a system for representing XML data as a series of SQL tables and show how it can be queried with XML-QL [42] through a translation process to SQL. Our initial implementation of NYQL uses a traditional relational database as a backend as well. Although their table format is similar to ours, theirs is more general since it deals with the inherent graph structure of Document Type Descriptors (DTDs). While our current data model is based on a set of trees and not general graphs, we may need to use a similar approach to deal with recursive data types.

The LINQ to SQL project [53] bridges the gap between object-oriented code and a relational query language. It allows users writing .NET applications to manipulate objects in their source code using a SQL-like query language. This approach is extended in DryadLINQ [75] to allow users to write queries over objects spread across a cluster of machines. However, these projects are more general purpose than ours. They target large-scale data processing rather than debugging and would not be as efficient for the latter.

In [65], Pike et al. describe a system for analyzing data distributed across hundreds or thousands of computers. A query is expressed in a new interpreted language called Sawzall which combines C and Pascal syntax and describes how data files should be processed. Query execution is broken up into two phases – a filtering phase and an aggregation phase. In the filtering phase, each data file is broken up into records and processed according to the user's query. In the aggregation phase, the values returned from the filtering phase are combined according to a set number of predefined aggregators, although the authors mention that adding additional aggregators is possible. Since this system is essentially a massively parallel computation, it is structured on top of MapReduce [40] and uses the Google File System [46] to store its output. Although Sawzall allows queries to be run efficiently across huge data sets distributed across thousands of machines, the queries themselves can not be as expressive as those in NYQL. For instance, Sawzall queries can only reference a single record in a single data file and have no memory about records that were previously processed. Although the Sawzall chaining mechanism can be used to do multiple passes over the data, the authors admit this is a clumsy approach. Also, the authors state that Sawzall is not good at handling database join operations, while SQL is the opposite. Finally, the data files processed in a single Sawzall query must be composed of exactly one type of record, while NYQL queries can be run over an arbitrary number of arbitrary record types.

Pig Latin [63], developed at Yahoo! is similar to Sawzall in that it is designed for ad-hoc analysis of massive data sets. It runs on top of Hadoop [21], an open-source implementation of MapReduce and provides an imperative SQL-like language with a fully-nested data model like ours. Pig Latin provides first-class support for user-defined functions, which can appear in the filtering, grouping, and per-tuple processing stages. While NYQL also supports user-defined functions, it currently only supports them in "output" statements. Like Sawzall, Pig Latin's focus is more for processing a large set of similar input records and producing a set of output records, while our language is strictly intended for debugging.

**Replay Debugging**

While our system is designed to allow developers to explore the state of each node and messages sent in distributed systems once a bug is captured, replay debugging techniques focus on reproducing executions containing a bug. As a result, logging statements can be eliminated since the state of the system can be checked at replay as the data is re-generated. WiDS checker [57] uses model checking techniques to verify both simulation and deployment runs built with the WiDS toolkit. Their approach uses "time travel" debugging and online predicate checking to check for violations as the program is replayed. However, their predicate language, while powerful, can become quite complicated. For example, they support injecting python code to manipulate state variables, which complicates their syntax. On the other hand, our query language is very simple and easy to understand. They also provide a message-based event graph similar to Nebula, although it does not appear to be as feature-rich. In addition, checking complex predicates during replay can lengthen the execution time from a factor 4 to 20 in the authors' experience.

Liblog [45] uses interposition to log side effects of non-deterministic C library calls, allowing applications to be replayed. Although our work is not focused on the ability to replay a run, by logging the system state over time we gain similar functionality, by being able to access global system state through queries. Friday [44] acts as a “distributed GDB,” allowing the developer to replay multiple communicating processes concurrently. The developer can step through the distributed system’s execution and set breakpoints or watchpoints. Replay of multiple processes has the advantage that the developer can inspect the system state at a finer granularity than what we support. On the other hand, deterministic replay requires fairly substantial logging overhead and may not be appropriate for performance-sensitive applications. In addition, single-stepping through a replay may not be sufficient for debugging all bugs, especially in the case of performance problems. Communication patterns of an overall system are also very difficult to visualize in this manner, which is something that Nebula makes easy.

Software-only replay debuggers have also been extended to deal with multi-core architectures. Altekar et al. [27] note that in many cases, an exact run does not need to be replayed – it suffices to find an execution that produces the same *output conditions* as the original. PRES [64] is another replay debugger for multi-core systems, although it is based on probabilistically reproducing the bug. They show that with a relatively low number of attempts, they can successfully reproduce a bug.

### Query-based Debugging and Visualization

Using queries for debugging is not a new concept, although our approach differs from past work in a number of ways. In [68], the authors describe extensions to P2 [58] to provide an integrated distributed continuous query processor for execution tracing of systems written in P2. This query processor executes queries written in OverLog, the same language that implements P2. P2’s query processor shares a number of similarities with our work. Both aim to remove ad-hoc script-based debugging, and both have a mechanism for specifying what should be logged using a query-based mechanism. On the other hand, using OverLog as a query language means that their debugging system is intimately tied to P2 — they present no story for how their query-based debugging techniques could apply to other systems. Our work takes care to define a *new* query language, which is not tied to any particular implementation and does not require a specific distributed system to work. Finally, OverLog is a much more complicated language than NYQL. As a result, P2 supports can support more interesting queries than what are possible with NYQL, but the language itself can make them very hard to understand. NYQL is designed to be a very simple query language, giving developers a fairly powerful interface to logged data while still staying as simple as possible.

Hy+ [38] is a system for network management and distributed debugging based on GraphLog, a visual query language. Nebula, our graphical debugging environment incorporating a query interface, which we describe in Section 5.3, shares some of the same tenets as Hy+, including visualization, abstraction, and filtering. However, GraphLog is more of a "visual" query language, while ours is based on the structure of the underlying data. In addition, our user interfaces show results differently. Theirs is a more general graph structure of simple nodes with a string label, while ours is more tree oriented containing well-structured data.

Query-based approaches have also been used in the area of network monitoring [69, 37, 73]. However, none of these systems are as general-purpose as ours and can only query a subset of the necessary events when debugging distributed systems.

In [31], Braun describes a visualization system for debugging. It shares a few features with our graphical user interface, including a tree-structured view of "tasks," similar to our function call stacks, and arrows representing communication patterns. xDIVA [35] provides *visualization metaphors* (VMs), which allow developers to define mappings from different data structure types to visual representations that can aid in debugging. While we have not pursued this direction of research, integrating data structure visualization of some type into Nebula could be valuable future work.

**Model Checking**

Model checking, as it applies to testing programs, is a means of programmatically exploring the state space of an application, in an attempt to find a state where a bug exists. Spin [48] is one such model checker, able to find bugs based correctness properties expressed as linear temporal logic statements in models of distributed systems. These properties can specify both *safety* constraints (the specified condition is always true), as well as *liveness* constraints (the specified condition must eventually be true).

Verisoft [47] was one of the first tools developed for model checking *actual program implementations*, as opposed to *models* of programs, which is where model checking originally got its name. Although not as flexible as Spin, it is able to find deadlocks and assertions failures in real `C` and `C++` systems composed of multiple concurrent processes, as opposed to the models that Spin checks.

The `Java` PathFinder (JPF) [71] was developed at NASA as an integrated tool for model checking, program analysis, and testing. This work focused on testing `Java` programs and featured a new `Java` Virtual Machine, able to analyze Java bytecode, rather than the `Java` source itself. Like Spin, JPF is able to check for linear temporal logic property violations, as well as deadlocks and user-defined assertions.

MaceMC [50] combines state-space exploration with random walks to find liveness violations in real distributed systems written in MACE [49]. The authors argue that liveness properties are more natural to test, since they describe how the system should behave when it is *correct*, as opposed to safety properties, which only specify ways the system can be *incorrect*.

Chess [62] is a model checking tool for enumerating all of the possible thread interleavings in a concurrent program. This enables developers to deterministically reproduce potentially find rare bugs that are normally difficult to reproduce. Musuvathi et al. extend this work in [61], which adds a notion of *fairness* for scheduling threads in programs that do not necessarily terminate.

MacePC [51], an extension of MaceMC, embeds a notion of time into the state-space exploration. This allows MacePC to search for particular executions where the simulated runtime differs from the runtime of a common execution by some margin. Executions found in this way are likely to suffer from performance bugs.

While model checking is a very powerful testing tool, often helping developers discover bugs that are hard to otherwise reproduce, it unfortunately does not fix these bugs on its own. Instead, developers must still rely on separate techniques to diagnose the problems they find. As a result, these techniques are entirely complementary to ours, as they solve orthogonal problems.

**Live Debugging**

Although we currently focus on post-mortem bug analysis, there are a number of systems based on instrumenting and tracing live runs. D$^3$S [56] uses dynamic instrumentation to inject property checking code into running systems, which allows them to test for faults without modifying the original code. Our approach to modifying the logging generated by a running system, described in Section 4.4.4, also uses binary rewriting. While the uses for dynamic binary instrumentation are different in these two cases, it is a promising technique to help deal with systems that cannot be restarted.

MaceODB [39] is a Mace [49] extension that translates developer-specified properties into code that tests these properties at runtime. These properties can be either for application *safety* — a condition that should always be true, or *liveness* — a condition that should eventually be true, and can be specified over the state of the entire system. The MaceODB runtime manages shipping the appropriate application state needed for distributed computation of property evaluation among all nodes necessary. However, using this approach for *deployed* systems, as they suggest, would mean that the developer had a known set of predicates they could leave running to be checked. If a bug arose that was not covered by any of the specified properties, the developer would have to fall back on other approaches to debug the system. In addition, there is no way to modify the predicates checked at runtime. A combination of this approach and the binary rewriting approach of D$^3$S could be promising.

CrystalBall [74] is an interesting approach to live system debugging that integrates model checking techniques with real system execution. Each node continuously runs a state-space exploration on a recent consistent snapshot of its neighborhood and uses the results to try and predict and avoid possible future violations of specified safety properties. Although the authors demonstrated the usefulness of this approach in finding real bugs, the results are similar to that of program testing — a property violation may be discovered, but no support is provided for determining *why* it failed.

Flight Data Recorder [70] uses a clever log compression scheme and a query-based interface to check for configuration errors in users' Windows registries. While not intended to debug distributed systems, log compression schemes could be useful in reducing the overhead of logging in systems like ours.

**Tracing Techniques**

DTrace [33] uses dynamic instrumentation for tracing low-level operating system events in both user and kernel space. It includes a high-level programming language (D) for specifying which events should be traced and allows users to set predicates on event conditions. DTrace can instrument applications without restarting them, which is a benefit over our current implementation. However, predicates defined through NYQL queries can refer to program state or message contents that DTrace does not have access to. DTrace could be used in conjunction with NYQL to populate a set of low-level logs which could then be queried with our language.

Project 5 [26] use statistical techniques to correlate message timings sent between nodes to reconstruct causal paths of communication in distributed systems. We employ paths as one of the fundamental objects that can be queried. However, because Project 5 treats the distributed system as a black box, it cannot leverage additional information for querying system state nor does it contain a domain-specific language for manipulating path information.

Pip [66] and X-Trace [43] are two complementary techniques to ours. Both systems focus on detecting paths of communication in distributed systems, but do not take black box approaches. With Pip, a developer can annotate their code to map high-level named paths to particular code segments. Developers can then write expectations about the quantity or behavior of different paths. X-Trace is similar in spirit but relies on protocol extension to perform metadata propagation for path reconstruction to work properly.

Chapter 2, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

# Chapter 3

# A New Debugging Model

So far, we have examined the most common distributed system debugging process. We have shown that most log file processing is tedious, error-prone, and fragile. In this chapter, we introduce a new approach to debugging that addresses these three main concerns. We present a new abstraction for log file data, the *state matrix*. We then describe a new query language, NYQL, centered around it. Finally, we describe our implementation of NYQL.

## 3.1 The State Matrix

Many of the problems with traditional log file analysis stem from the fact that text log files lack structure. Each log entry is independent of all others, so it becomes difficult to traverse them in a meaningful fashion. The key to our approach lies in the way we conceptualize log files. Rather than lines of text, we prefer to think about log files more generally as *objects*. Furthermore, we organize these objects into a logical three dimensional matrix that developers can interact with.

We call this abstraction the *state matrix*. A depiction of the state matrix can be seen in Figure 3.1. All log messages are entries in this matrix, organized by time, node, and event type. The “time” dimension contains an entry for every timestamp at which a logging event occurred on any node. The “node” dimension has an entry for every node in the system. Here, a node is used to represent each logically distinct entity in the system, whether they be identified by host names, host names plus ports, or any other

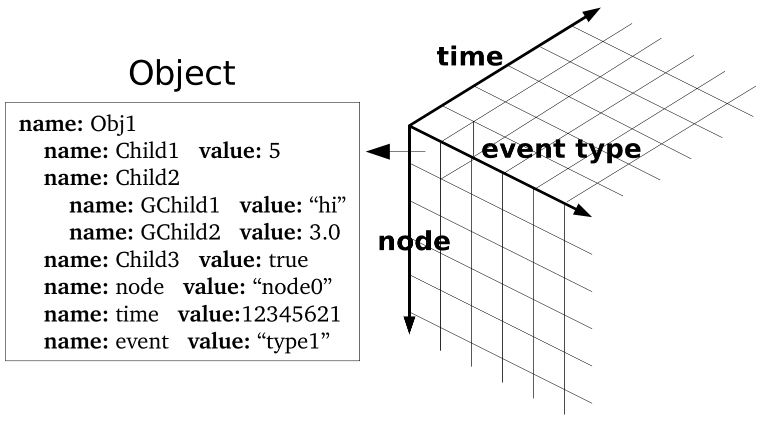


**Figure 3.1**: The State Matrix Abstraction

identifiers. Finally, "event type" organizes all logging messages by type. Each different log statement in a program's source code can be thought of as having a different event type. We note that this dimension is unordered, as different event types have no intrinsic ordering.

### 3.1.1 The Object Model

As we mentioned above, we view each element of the state matrix as an object. Objects in our model can be viewed as tree-structured entities that are defined recursively as follows. Each object contains two or three things: (i) a name, (ii) an arbitrary number of *children*, which are also objects, and (iii) a value, if the object has no children. Additionally, each *root object* in the state matrix (e.g., Obj1) has three children corresponding to the three state matrix dimensions – "node," "time," and "event." These three children allow us to reference each object's position within the state matrix from the object itself. We access sub-objects in the state matrix using the traditional "dot" operator. For instance, in Figure 3.1, "Obj1.Child2" refers to the entire Child2 object, while "Obj1.Child2.GChild2" refers to the value 3.0.

```
name: VecObj
   name: values
      name: value   value: 2
   name: values
      name: value   value: -3
   name: values
      name: value   value: 0
   name: values
      name: value   value: -8
```

vector<int> VecObj = {2, -3, 0, 8}

(a) Vector

```
name: VecObj
   name: keys
      name: value   value: 1
   name: values
      name: value   value: “test”
   name: keys
      name: value   value: 2
   name: values
      name: value   value: “map”
```

map<int, string> MapObj =
{1=>“test”, 2=>“map”}

(b) Map

**Figure 3.2**: State Matrix Data Representations

### 3.1.2 Container Types

In addition to basic tree-structured objects, elements of the state matrix can also contain a few special container types such as vectors and maps. Figure 3.2(a) shows the representation for a vector of integers containing the values 2, $-3$, 0, and 8. Vector objects contain a child named “values” for each element in the vector. If the vector contains simple types, then each “values” object has a single child named “value” containing the element’s value. If the vector is composed of objects, then “values” becomes an object of that type.

Figure 3.2(b) shows an example of a map represented in the state matrix. Maps are similar to vectors, but instead of containing only “values,” they also have “keys” corresponding to each value. We note that the vector type is sufficient for representing arrays, sets, lists, and any other linear container types, and the map type is capable of storing hash tables or traditional sorted maps. Although these two representations may seem a bit unusual, we give a sense of their usefulness in Section 3.2.3.

### 3.1.3 Composite Objects

Although the state matrix as an abstraction can store any types of objects, we believe it should, at the very least, store objects corresponding to the set of features described in Section 2.2.1 to be useful from a debugging standpoint. That is, the state matrix should contain function call parameters and possibly return values, function call nesting, object aggregation into logical events, causal paths, and program state. We note that some of these components are not objects that are logged in the traditional sense. For instance, we do not expect programmers to call a "logEvent" function and pass an entire Event object. Similarly, an object representing a causal path cannot even be logged in a single place, as it contains events happening on multiple nodes. Instead, we define *composite objects* as objects that exist for the purpose of being queried, but are composed of other objects in the state matrix. The linking of objects into composite objects is achieved through extra data stored as *state matrix metadata*.

The first type of composite object is the *Event*. Event objects are used to logically aggregate logging statements (objects) corresponding to some higher-level functionality, such as receiving a network message or invoking a member of a class's public API. In our current model, events contain a nested sequence of objects corresponding to function calls. Each object is named in relation to the name of the function, and the children contain the values of the parameters that were passed when the function was called. We call these types of objects *function call objects* or *parameter objects* interchangeably. Event objects may also contain any other objects the user chooses to log. Events are demarcated by *start* and *end* markers stored in the state matrix metadata. The stack depth of each logged function call is also stored here and is used to compute the correct nesting.

While Event objects are composed of individual function call objects, *Path* objects are composed of Event objects corresponding to a causal path of communication, similar to the model previously used by Pip [66]. In our model, a Path object represents a partially-ordered set of Events, using the relation *causes* for ordering. We define $causes(A, B)$ to be true if and only if a message is sent during Event $A$ which causes Event $B$, or $causes(A, C)$ and $causes(C, B)$ are both true for some Event $C$. More informally, a Path object captures events that are executed on multiple nodes of a distributed system as a consequence of a single event.

## 3.2 NYQL: A New Query Language

In this section, we introduce NYQL, a new query language designed to alleviate many of the common problems with debugging distributed systems. Although many query languages currently exist, we felt none of them fit our needs. In spirit, our data model is most similar to that of OQL [36], the Object Query Language. Queries in OQL operate on an object-oriented database, similar in structure to ours. However, we chose not to use OQL as a query language for a few reasons. First, since OQL is designed as a general-purpose object-oriented query language, it has many features and complexities that we do not need. Second, and partly because of this complexity, no commercial databases exist that support OQL.

### 3.2.1 Why Not SQL?

Although OQL is not readily available, there are many commercial databases available that support SQL, the Structured Query Language. Since the state matrix represents a large, virtual database, SQL seems like a natural choice as a front-end query language. However, the state matrix consists of *objects*, whereas SQL queries work over *flat tables*. One way to address this discrepancy is to split objects across multiple tables, allowing the user to query these tables directly. While this could work in theory, it would require programmers to reconstruct complete objects themselves. Since users will have to perform joins for each descendant object, the query complexity grows proportionally to the queried data structure complexity, which is an undesirable property. In addition, the table returned by such a SQL query would not maintain any of the desired structure of the original objects. Finally, changes to table schemas for optimization purposes or just general restructuring could invalidate many user queries. These are the same problems that caused us to abandon text logging in the first place. Thus, we set out to design a simple domain-specific query language specifically targeting exploration of our proposed state matrix abstraction.

### 3.2.2 NYQL Fundamentals

Recall our goals of *expressibility* and *clarity* for a query language. Meeting these goals means making it easy to extract information from the state matrix while expressing relationships across event types. Additionally, we desired to support queries over composite objects such as Events and Paths. With these ideas in mind, we created a new high-level imperative query language, NYQL, that explicitly centers around the state matrix abstraction. At a high level, NYQL has five main components:

**Loops.** Loops allow traversing slices of the state matrix and comprise the body of almost all NYQL queries.

**Conditional Expressions.** Filtering is a necessary component of all useful query languages. These expressions allow users to focus on elements of the state matrix matching specified criteria.

**Variable Assignment.** Often, a debugger wishes to find related events that occurred at similar times. For instance, when iterating over all occurrences of a particular function call object, a debugger may wish to see the Event object each function call was part of, or they may wish to see a node's state at that time. Variable assignment allows users to find and save references to particular object occurrences.

**Aggregation Functions.** Aggregating data in various ways is a very common task when debugging, for example measuring elapsed time between events or counting the number of times a particular event took place. NYQL supports aggregation through user-defined functions. We cover how these work in more detail in Section 3.3.6.

**Output Statements.** Output statements return query results to the user. NYQL has two different kinds of output statements – one returns objects the user can inspect and the other produces a graph. Graphing is commonly done as a part of data analysis and so is supported directly in NYQL.

### 3.2.3 NYQL Grammar and Examples

We now introduce NYQL syntax through its grammar and a series of examples. We first define a few basic rules used in the rest of the grammar. Identifiers are a single letter followed by zero or more letters or digits. We define any production of the form *X*List to mean a comma-separated list of *X* elements. The `Var` production simply expands to a list of Identifiers separated by dots. Although we do not define it in the grammar, the `Expr` production used in various places represents arithmetic or logical expressions involving variables and constants. Finally, we define the `eps` production which matches the empty string.

```
Identifier ::= [:letter:][:letterdigit:]*
XList ::= X "," XList | X
Var ::= Identifier "." Var | Identifier
eps ::=
```

**Basic Structure**

A NYQL query is composed of one or more statements. Each statement is either a “foreach” statement, a “let” statement, an “output” statement, or a “graph” statement.

```
Query ::= Statement+
Statement ::= ForEach | Let | Output | Graph
```

While the grammar allows arbitrary interleaving of these statements, there are semantic restrictions. Each query must have exactly one “output” or “graph” statement, but it cannot have both. This statement must occur in the inner-most loop of the query.

**ForEach Statements**

“foreach” statements are used to iterate over slices of the state matrix. They have five components: (i) an identifier used to reference each object the loop will iterate over, (ii) a list of object names to iterate over, (iii) an optional *matrix expression* used to slice the state matrix across the “node” and “time” dimensions, iv) an optional *where clause*, used to further narrow the scope of the loop, and v) a list of statements to execute for each object meeting the specified criteria.

```
ForEach ::= "foreach" Identifier "in" ObjList MatrixExp Where "{"
  Statement+
"}"

Obj ::= Identifier | Identifier "*" | FuncObj
FuncObj ::= Identifier "(" NonParen ")"
```

```
MatrixExp ::= MatrixNotation VarSuffix
MatrixNotation ::= "[" Exprs "]" | eps
Exprs ::= Expr | Expr "," Expr
VarSuffix ::= "." Identifier VarSuffix | eps
Where ::= "where" Expr | eps
```

There are three ways valid state matrix objects can be named. They can either be a single identifier, a complete function prototype (for parameter objects), or an identifier followed by a star. The latter case indicates that the loop should iterate over all objects starting with the given prefix. We note that multiple object names can be given in a comma-separated list. In this case, the loop is to execute over all objects listed. However, any members used in where clauses or output statements must be present in *all* objects listed. The `VarSuffix` production is used if a loop is to iterate over a vector or a map child of objects from an outer loop. Thus, we have a uniform way of iterating over objects in the state matrix as well as container types.

The optional matrix expression, enclosed in brackets, is either one or two expressions separated by a comma. The first expression is used to put restrictions on the “node” dimension; the second is for the “time” dimension. If only one expression is given, it can be for either “node” or “time.” These expressions commonly take the form “node == $X$” or “time < $Y$,” etc.

The `Where` production is used as it is in SQL to filter the objects given in `ObjList` based on the values of any of their children. For example, a loop “foreach b in Obj1” (where Obj1 is defined as in Figure 3.1), we could add the clause “where (b.Child2.GChild2 > 2.5)” Since b is of type Obj1, and Obj1 objects have a child named Child2 which in turn has a child named GChild2, this where clause is valid.

## Let Statements

While “foreach” statements are used to iterate over slices of the state matrix, “let” statements are used to obtain a reference to a single object or a collection of objects.

```
Let ::= "let" Identifier "=" ObjType MatrixExp ";"
ObjType ::= Identifier | FuncObj
```

With “let” statements, we require that `MatrixExp` either contains two expressions, one for node and one for time, or does not appear at all. Traditionally, using the

"==" operator for both dimensions would give us a reference to a single object, but in many cases, the equals operator is not sufficient for the time dimension. Since all logging events across nodes are likely to occur at unique timestamps, using a timestamp from one event to match another is almost never going to work.

Instead, we introduce a new operator in NYQL— "@." This operator can only be used on the time dimension of `MatrixExp`. Let `ObjType` be $T$. Then, the expression "time @ $X$" is equivalent to time = max$\{t.\text{time} : t \in T \text{ and } t.\text{time} \leq X\}$. In other words, the time variable is assigned the greatest value of time that exists for the target object type that is less than or equal to the given value.

## Output Statements

The "output" statement is one of two ways of returning results from a query. This statement returns the objects named in `AsExprList`. Expressions returned here can simply be objects, children of objects, arithmetic expressions, or *aggregate functions*, if the parameter given to them is a set of objects or a container type. Each element may also be renamed using the "as" clause, although arithmetic expressions or aggregate functions must be given names in this manner.

```
Output ::= "output" AsExprList OrderBy LabelBy Limit ";"
AsExpr ::= Expr "as" Identifier | Expr
OrderBy ::= "order by" VarList | eps
LabelBy ::= "label by" VarList | eps
Limit ::= "limit" Integer | eps
```

"output" statements also contain three optional directives. The `OrderBy` production allows users to control which fields to use when sorting the output, exactly like in SQL. The `LabelBy` production allows users to change how the output objects are labelled. Traditionally, an object named "output" is returned for each element of the state matrix matching the user's query. Each "output" object has a child for each element in the user's output statement. If `LabelBy` is present, each output object will be named as the value of the given object instead. Finally, the `Limit` production works as it does in SQL, allowing the user to restrict the number of results returned from a query.

## Graph Statements

The second method of obtaining results from a query is with a “graph” statement. While “output” is used to return objects, “graph” is used to generate graphs.

```
Graph ::= "graph" GraphExprList FileRedirect ";"
GraphExpr ::= Expr "vs" Expr
FileRedirect ::= ">" Identifier | eps
```

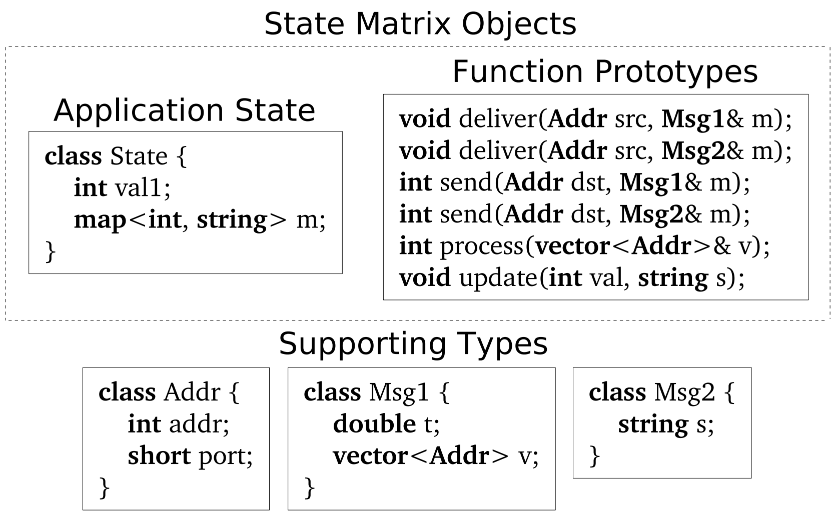
A “graph” statement will plot multiple lines on a single graph, each corresponding to a `GraphExpr`. Each `GraphExpr` is of the form X_axis vs Y_axis, where both axis objects must be integer or floating point values. The optional `FileRedirect` production allows users to name the generated output files. Raw data is written to *Identifier*.dat, the gnuplot [12] code for generating the graph is written to *Identifier*.gnuplot, and the graph is generated as *Identifier*.ps. If no file redirection is specified, the default identifier is “default.”

## Examples

We now present a series of example NYQL queries designed to show different language features. Since NYQL operates over objects, we need a reference set of objects to work with. Figure 3.4 shows a set of C++-like class definitions as well as a set of function prototypes. For the following examples, assume our state matrix contains objects corresponding to the class and function definitions in the dotted outline. The State class represents the state of the running application and would be logged periodically. The parameters for each of the functions listed would be logged each time the functions are called. The Addr and Msg classes are simply supporting types used to make the examples more interesting. For details on how these objects would be logged, see Chapter 4.

```
foreach s in State [node == x, time <= 1234567890.25] {
  output s order by s.time limit 5;
}
```

**Figure 3.3**: Example NYQL Query 1



```
class State {
   int val1;
   map<int, string> m;
}
```

```
void deliver(Addr src, Msg1& m);
void deliver(Addr src, Msg2& m);
int send(Addr dst, Msg1& m);
int send(Addr dst, Msg2& m);
int process(vector<Addr>& v);
void update(int val, string s);
```

```
class Addr {
   int addr;
   short port;
}
```

```
class Msg1 {
   double t;
   vector<Addr> v;
}
```

```
class Msg2 {
   string s;
}
```

**Figure 3.4**: Example State Matrix Objects and Supporting Types

Our first example, shown in Figure 3.3, consists of a single loop over all of the State objects in the state matrix. Here we wish to only see State objects that were logged on node “x” no later than time 1234567890.25. By outputting *s*, we output the entire contents of each State object which includes integer child *val*1 and map *m*, as well as the time the object was logged and the node identifier, which are automatically added to all state matrix objects. The “limit” and “order by” keywords work as they do in SQL — the output will be limited to the first two State objects in ascending time order.

```
foreach d in deliver(Addr src, Msg1& m) {
  let e = Event[node == d.node, time @ d.time];
  output e label by e.time;
}
```

**Figure 3.5**: Example NYQL Query 2

In the second query, shown in Figure 3.5, we iterate over all of the function call objects corresponding to delivery of Msg1 messages. We use the entire function prototype here to disambiguate between functions with the same name but different prototypes. We use the “@” operator with a “let” statement to get a reference to the Event object corresponding to each call of the deliver function. Then, we output the entire Event object. We use "label by" to label each returned object by the time the Event object is logged.

```
let e = Event;
output max(e.duration) as maxDur, min(e.duration) as minDur;
```

**Figure 3.6**: Example NYQL Query 3

Example three, shown in Figure 3.6, is a query without loops. Here we are assigning a variable to the collection of "Event" objects in the state matrix. The object *Event.duration* refers to the length of time, in milliseconds, that it took the event to execute. Aggregate functions in NYQL operate on a named child of a collection of objects, and since *e* represents all Event objects, we can pass *e.duration* to the max and min functions. Thus, we are computing the maximum and minimum event durations across all nodes.

```
foreach d in deliver*, send* [time < $START + 5] {
  let e = Event[node == d.node, time @ d.time];
  graph d.time vs (e.duration * 1000) > durplot;
}
```

**Figure 3.7**: Example NYQL Query 4

The fourth query, shown in Figure 3.7, uses three more language features. First, we specify "deliver*" *and* "send*" as the objects we are looping over, which means any objects starting with the name "deliver" or "send" will match. Thus, this query iterates over occurrences of the four deliver/send objects. Second, we use the special variable $START, which contains the time of the earliest logged object in the run, so we are looking at all messages sent or received in the first five seconds. Finally, we are using the "graph" command to generate a graph with the time each message is delivered or received on the x-axis and the execution time (in microseconds) associated with each of these events on the y-axis.

Figure 3.8 shows a query with two nested loops. The first loop iterates over all occurrences of the process() function. The second loop is iterating over all of the elements of the *v* parameter. Since *b* is the process object which has a parameter *v*, *b.v*

```
foreach b in process(vector<Addr>& v) {
  foreach c in b.v.values where (c.value.port == 80) {
    output b.node, b.time, c order by c.value.addr;
  }
}
```

**Figure 3.8**: Example NYQL Query 5

refers to this parameter. As discussed in Section 3.1.2, vector objects have a “values” child for each element in the vector, so *b.v.values* refers to the vector’s values. Also recall that each “values” element has a single child named “value.” In this case, the element type is Addr, which has two children — “addr” and “port.” Thus, we compare against the port child of each element in the vector by specifying *c.value.port*. Finally, we output the node, time, and “addr” member of any Addr object passed to a process() function whose port was 80.

```
foreach e in Event where (e.duration > 100) {
  let p = Path[node == e.node, time @ e.time];
  output e.duration, p order by e.duration;
}
```

**Figure 3.9**: Example NYQL Query 6

Figure 3.9 shows our final example query. In it, we are iterating over all Event objects where the execution duration was greater than 100 milliseconds. Then, we are finding the corresponding Path object this Event was part of. As a result, we end up with all Path objects in which any individual event took longer than 100 milliseconds.

## 3.3 Implementing NYQL

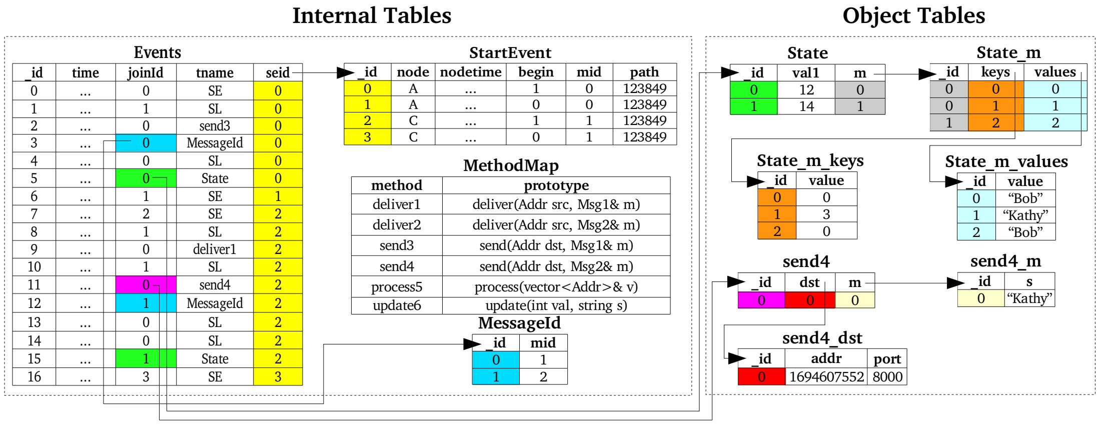
One of the main advantages of having a structured, high-level language like NYQL for debugging distributed systems is that it frees developers from having to worry about log file formats or manipulating low-level data directly. The side effect of this advantage is that developers have a stable interface to work with, regardless of the underlying language implementation.

This section describes our prototype of NYQL, which uses a PostgreSQL database (version 8.1.11) to store log file data. Each NYQL query is translated into a set of SQL queries that are run against the database. We then post-process the query results to reconstruct the objects present in the user's NYQL query. We cover our table schemas in Section 3.3.2, the database population process in Section 3.3.3, the translation process from NYQL to SQL in Sections 3.3.4 and 3.3.6, and finally the object reconstruction process in Section 3.3.5. However, we first give some rationale behind our implementation choice next in Section 3.3.1.

### 3.3.1 Rationale

Although we are using a SQL database in our implementation, this is not the only possible choice. In fact, using SQL might seem counterintuitive since we spent Section 3.2.1 describing why SQL was *not* a good query language for debugging. However, there is an important distinction between a query language the user sees and what goes on behind the scenes. The SQL queries we generate are quite complex and could not easily be generated by humans. However, they are all automatically generated and executed by our NYQL to SQL translator so this complexity is hidden from users. On the other hand, the relational database community is well established, with many available, optimized database implementations to choose from. We were confident we would not have to spend any time debugging the back end — SQL queries would all work as expected. In addition, we expected reasonable performance, even though relational databases are not inherently designed to store object-oriented data.

On the other hand, since the objects we consider are organized as trees, a natural alternative to our approach would be to serialize objects as XML documents. These documents could be queried by compiling NYQL expressions into XML-QL [42], or the more recent standard XML query language XQuery [30], which is strictly more expressive than both SQL and NYQL. The approach would even support a relational back-end, given the XML support of commercial relational database management systems [3, 10, 24]. We chose instead to exploit the limited expressiveness of NYQL and its narrow focus on the state matrix in order to implement a more targeted translation to SQL. We cover the performance impact of this choice in Section 4.5.



Events

| _id | time | joinId | tname | seid |
|---|---|---|---|---|
| 0 | ... | 0 | SE | 0 |
| 1 | ... | 1 | SL | 0 |
| 2 | ... | 0 | send3 | 0 |
| 3 | ... | 0 | MessageId | 0 |
| 4 | ... | 0 | SL | 0 |
| 5 | ... | 0 | State | 0 |
| 6 | ... | 1 | SE | 1 |
| 7 | ... | 2 | SE | 2 |
| 8 | ... | 1 | SL | 2 |
| 9 | ... | 0 | deliver1 | 2 |
| 10 | ... | 1 | SL | 2 |
| 11 | ... | 0 | send4 | 2 |
| 12 | ... | 1 | MessageId | 2 |
| 13 | ... | 0 | SL | 2 |
| 14 | ... | 0 | SL | 2 |
| 15 | ... | 1 | State | 2 |
| 16 | ... | 3 | SE | 3 |

StartEvent

| _id | node | nodetime | begin | mid | path |
|---|---|---|---|---|---|
| 0 | A | ... | 1 | 0 | 123849 |
| 1 | A | ... | 0 | 0 | 123849 |
| 2 | C | ... | 1 | 1 | 123849 |
| 3 | C | ... | 0 | 1 | 123849 |

MethodMap

| method | prototype |
|---|---|
| deliver1 | deliver(Addr src, Msg1& m) |
| deliver2 | deliver(Addr src, Msg2& m) |
| send3 | send(Addr dst, Msg1& m) |
| send4 | send(Addr dst, Msg2& m) |
| process5 | process(vector<Addr>& v) |
| update6 | update(int val, string s) |

MessageId

| _id | mid |
|---|---|
| 0 | 1 |
| 1 | 2 |

State

| _id | val1 | m |
|---|---|---|
| 0 | 12 | 0 |
| 1 | 14 | 1 |

State_m

| _id | keys | values |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 2 | 2 |

State_m_keys

| _id | value |
|---|---|
| 0 | 0 |
| 1 | 3 |
| 2 | 0 |

State_m_values

| _id | value |
|---|---|
| 0 | “Bob” |
| 1 | “Kathy” |
| 2 | “Bob” |

send4

| _id | dst | m |
|---|---|---|
| 0 | 0 | 0 |

send4_m

| _id | s |
|---|---|
| 0 | “Kathy” |

send4_dst

| _id | addr | port |
|---|---|---|
| 0 | 1694607552 | 8000 |

**Figure 3.10**: Table Relationship for a Subset of the Objects in Figure 3.4

## 3.3.2 Table Schemas

One challenge of our approach is deciding how tree-structured objects should be stored as tables. Our general approach is that each object is stored in a table with one column per child. If the child is a basic type, like an integer, floating point number, string, etc., the column contains the actual value of the child. On the other hand, if the child is an object itself, the column contains integers representing foreign keys to the table representing the child.

Figure 3.10 shows a subset of the tables that are generated for the hypothetical state matrix objects presented in Figure 3.4. Arrows between tables and colored cells represent foreign key relationships. There are two types of tables in our implementation – *object tables* and *internal tables*. Object tables directly correspond to state matrix objects and can be queried in NYQL. On the other hand, the contents of internal tables are not available directly through queries, but serve to help with the translation process. They can be thought of as implementing the *state matrix metadata* mentioned in Section 3.1.3, including support for constructing Event and Path objects.

### Object Tables

Object tables directly correspond to non-composite objects in the state matrix. Although Figure 3.4 shows seven objects — six function call objects plus the State object — we only show how two are stored due to space limitations. Every table has an *id* column, used as the table's primary key. In general, we have one row (and one unique *id* value) for each time an object is logged. Here we see the State object has three columns, one for *id* and one for each of its children *val*1 and *m*. Since *val*1 is an integer, the actual values are stored in the State table. On the other hand, *m* is an object, so the *m* column is used as a foreign key into the State_m table, which is responsible for storing the map. We note that for any table $T$ containing a column $child$ that is a foreign key into another table, the table for the child is named $T\_child$.

The State_m table has three columns, *id*, *keys*, and *values*, corresponding to its state matrix representation as described in Section 3.1.2. Both the *keys* and *values* columns are foreign keys into the corresponding keys and values tables. In this case, since both the key and value types are basic types, the keys and values tables have a single column named *value* that stores the actual keys and values in the map. Note that in this case, the *id* value of 0 is repeated twice in the State_m table. This indicates that when the map was logged with *id* value 0, it had two keys and two values.

The send4 table corresponds to the send() function call that takes a Msg2 object as a parameter. This correspondence can be seen in the MethodMap table, which we describe in the "Internal Tables" section below. The send4 table has two non-*id* columns — *dst* and *m* — one for each of the function's parameters. Since both parameters are objects, both columns are foreign keys to their respective tables. send4_m simply has a single non-*id* column, representing the *s* field of the Msg2 object. send4_dst has two non-*id* columns, just like the Addr object it represents.

Although not shown, object tables also exist for deliver1, deliver2, send3, process5, and update6. These tables are constructed in the same fashion as State and send4, so we omit their descriptions.

## Internal Tables

A collection of object tables alone is not enough to resolve queries. For instance, they do not contain timing information or node identifiers. In addition, the tables for function call objects, such as send4, do not include any notion of nesting, and we have no indication where logical events begin or end, so we cannot construct Event objects.

Finally, none of the object tables include any sort of path information, so we cannot construct Path objects either.

The internal tables exist to remedy these issues, and act as the "glue" keeping the system together. The first table we describe is MethodMap. Although we use full function prototypes as object names in queries, prototypes contain a number of characters that are invalid for table names, including spaces. Instead, we assign table names to function call objects by taking the function name and appending an integer counter. We maintain the mapping between table name and object name (prototype) in the MethodMap table.

The most important internal table is the Events table, which acts as an index to all other tables in the system and stores some necessary information for constructing Event objects. Figure 3.10 shows the entries that would be logged for two complete hypothetical events. The Events table has an *id* column used as its primary key. The *time* column records a timestamp for each logged entry. The *joinId* column is used as a foreign key into all other tables in the system, as specified by the *tname* column. Finally, the *seid* column is the event number each entry belongs to and also functions as a foreign key into the StartEvent table. Although not shown for simplicity, the Events table also contains a *tid* column which stores the thread identifier of each log statement.

We now walk through the entries in the Events table to give an idea of how everything fits together, and to describe the other internal tables in context. The first entry is an "SE" entry, which corresponds to the StartEvent table. Since the *joinId* is 0, this means the data for this object can be found at *id* value 0 in the StartEvent table. We see that this entry marks the start of an event on node *A* on path 123849. The *mid* column is similar to a Lamport clock [54] and is used to establish the partial ordering of events with the same *path* value, as well as to match individual message sends and receives. Finally, *nodetime* allows a different recording of time other than the wallclock time, which is what is recorded in the Events table. For instance, a simulator could log its own value of time in this field.

Returning to the Events table, we see the second entry is "SL." These entries are used to establish the nesting relationship of each object in an Event. For "SL" entries, we use the *joinId* column to indicate whether the "stack depth" should be incremented or decremented, since there is no separate SL table that requires this field. The value of 1 means the stack depth should be incremented. Next, we see a “send3” entry. From consulting the MethodMap, we see that this entry corresponds to the execution of the send() function that takes a Msg1 object as a parameter. The next entry is “MessageId,” which gets logged whenever a message gets sent. The MessageId table simply records the *mid* value — which must be sent in the message — and is logged on the receiving end in the StartEvent table. Our last three entries in this event are another “SL,” this time indicating the stack depth should be decremented (marking the end of the send() function), followed by a “State” entry corresponding to an entry in the State table, and finally an “SE,” indicating an event has ended.

The second event shown indicates that the deliver1 method was called, which calls send4, as indicated by the “SL” with *joinId* 1 at Event *id* 10. Since another message was sent, we have another “MessageId” entry, followed by two “SL” entries with *joinId* 0, indicating the end of the send() function followed by the end of the deliver() function. Finally, we have another occurrence of a State object being logged followed by the closing “SE” entry.

### 3.3.3 Database Population

Before we can end up with tables as described in the previous section, we must first populate the database. There are two primary ways of doing this — have each logging statement directly insert the logged values into the appropriate tables on the fly, or log to files and copy all of the values into a database after the program terminates. We chose the latter method to avoid slowing down the program being debugged due to the cost of accessing a database, or having to buffer a potentially large amount of data to be written asynchronously.

However, the approach we chose is not without its own drawbacks. Instead of slowing down the program, we must now wait for logs to be copied into a database once a run is complete. In fact, we cannot even directly copy the logs after a run. Since we will end up with *n* log files, one for each node in the system, we first have to merge the log files together. This is necessary since each node assigns its own local values for all of the *id* columns it generates, but we must end up with a global set of *id* values covering *all* nodes in the system for a correct database.

The basic logic merging process works as follows. We first read an Event's worth of data from each of the *n* log files and choose the one with the earliest start time. We then assign new values for all *id* values present by keeping track of a global *id* value for each object name. However, we make sure to maintain the local *id* relationships within an Event, so that objects like vectors with multiple items get renumbered correctly. Once we have renumbered the Event, we write the output to a series of files, one per table. We do this to take advantage of PostgreSQL's bulk copying mechanism, which should make populating the database as efficient as possible. We then read a new Event from the node's log file we just finished, find the next smallest timestamp, and repeat the process until all log files have been read, renumbered, and written out. Finally, we use PostgreSQL's *copy* command to copy each table's data into the database.

### 3.3.4 Basic Query Translation

In this section, we describe the basic translation process from NYQL to SQL. There are a number of features that complicate this process, but we defer their discussion until Section 3.3.6. Query translation has four main steps, although a startup process must also be completed before any queries can be executed. We present each step in the context of translating the query shown in Figure 3.11 to SQL. We note that in many of our SQL queries, we return multiple columns with the same name, which does not work in practice. To handle this issue, our implementation assigns every column returned from a SQL query a unique name using "SELECT $X$ AS $Y$," but we omit this renaming process for clarity.

#### Pre-query Startup

Before any queries are translated, there are two startup tasks that must be done. We note that these tasks are not done for every query, but only once each time a user invokes our debugging environment.

First, we connect to the database and retrieve the contents of the MethodMap table. This allows us to translate between function call object names as specified by the user (*MethodMap.prototype*) and actual table names (*MethodMap.method*). See

Figure 3.10 for an example MethodMap table. Second, we query the database for the schema of all tables. Using the knowledge of our table naming scheme for child objects, along with the data returned from our schema queries, we build a mapping of all of the available objects in the system. We refer to this mapping as the *TypeMap* and use it to validate all user queries. We take special note of tables starting with a double underscore ("__"), as these are tables we have created. We describe the function and creation of these tables in the "Table Creation" section below.

```
foreach s in State [time <= 1234567890.25] {
  output s.node, s.time, 2 * s.val1 as v, s.m order by s.time limit 2;
}
```

**Figure 3.11**: NYQL Translation Example

### Parsing and Validating

The first step in the translation process is parsing and semantic checking. When we parse a query, we validate all object references against the TypeMap. In our example query, besides parsing for syntax, we check that "State" is a valid object type and that it has children named $time$, $val1$, and $m$, which it does. We also keep track of the each individual object the user wishes to output, which include $State.node$, $State.time$, $State.val1$, $State.m.keys.value$, $State.m.values.value$. We refer to this list as the *output object list*.

### Table Creation

After parsing a query, we check whether any table beginning with "__" contains all of the objects in the output object list. If such a table does not exist, we create a table named "__table1," for instance, that will store all of the values specified by the output object list. We do this as an optimization technique, since objects are frequently split up into many tables. As we can see in Figure 3.3.2, the data for the State object is spread across four tables. A naïve approach would require at least three SQL joins per query to extract this information, in addition to joins with the Events and StartEvent tables to associate a node identifier and a timestamp with each object. As a result, we join all of the data in an output object list into a single table, along with the associated node identifiers and timestamps, so that repeated queries over the same data can avoid these extra joins.

The table creation query can be broken down into three nested SELECT statements. The inner-most part of the query, which we designate the *NodeTimeQuery*, performs a join between the Events table and the SE table and extracts the object name as text ("State"), the *joinId* and *tid* of each State object along with the *node* and *nodetime* fields. In essence, this query matches each occurrence of a State object with the node identifier and timestamp from the SE table.

```
CREATE VIEW NodeTimeQuery AS
SELECT 'State'::text AS event, joinId, tid, node, nodetime AS time,
FROM Events, SE
WHERE (tname = 'State' AND seid = SE.id)
```

The next step in the table creation process is to join the results of the NodeTimeQuery with the State table itself to extract the necessary objects. In this case, the query includes *val*1 and *m*, so we include both of these columns in this query. We refer to this query as the *ObjectQuery*, shown below.

```
CREATE VIEW ObjectQuery AS
SELECT event, State.id, State.val1, State.m, tid, node, time
FROM NodeTimeQuery, State
WHERE (State.id = joinId)
```

If the object we are creating a table for only has children that are basic types, then at this point we are done. However, in this case, *State.m* is a map and contains data in three other tables, which we still need to extract. The final piece of the process performs left joins with each of the other tables containing data for the State object. We return all of the *id* values as well, since we use these in the object reconstruction process described in Section 3.3.5.

```
CREATE TABLE __table1 AS
SELECT event, ObjectQuery.id, State_m.id, State_m_keys.id,
       State_m_values.id, val1, State_m_keys.value,
       State_m_values.value, tid, node, time
FROM ObjectQuery LEFT JOIN State_m ON m = State_m.id
LEFT JOIN State_m_keys ON State_m.keys = State_m_keys.id
LEFT JOIN State_m_values ON State_m.values = State_m_values.id
```

We note that we repeat this process for each *root* object type in the output object list — we do not create a single table that joins multiple state matrix objects together. In addition, since $State.val1$ and $State.m$ both belong to the $State$ object, we only create one table.

### Index Query Generation

When the previous step completes, we have the names of a set of tables that store the objects, timestamps, and node identifiers of all of the objects in the output object list. The next step is to produce an *index query*, which extracts the appropriate *id* values for the object types that appear in each loop. The index query also computes the values of any expressions in the output object list, and handles any “order by,” “limit,” and “where” clauses as well. The index query we generate for the NYQL query shown in Figure 3.11 is shown below. Once again, we omit parameter renaming for clarity.

```
CREATE VIEW IndexQuery AS
SELECT id, 2 * val1 AS e0 FROM __table1
WHERE (time <= 1234567890.25) ORDER BY time LIMIT 2
```

Since we only have one loop, and one output object type, the index query is straightforward. We select *id*, which refers to $State.id$, and the desired expression as $e0$ from __table1, the table we created in the previous step. We add our “order by,” “limit,” and “where” clauses, so this query returns the correct two *id* values and expressions corresponding to the objects that should be returned from the user’s query.

To handle nested loops, we simply add a table to the “FROM” list for each loop in the query and extract an *id* column corresponding to each object being iterated over. SQL’s natural interpretation of multiple “FROM” tables gives us the exact same behavior as multiple nested loops. The only complication arises when a user uses a “let” statement with the @ operator. We defer the description of the translation of these types of queries to Section 3.3.6.

### Data Extraction

The final step in the basic translation process is to extract the elements in the user’s output object list corresponding to the *id* values in the index query. We do this with one query per root object type in the object output list. In this case, we only have one loop over one type so we end up with one query.

```
SELECT node, time, e0 AS v, id, m_id, m_keys_id, m_values_id,
       m_keys_value, m_values_value
FROM __table1, IndexQuery
WHERE (__table1.id = IndexQuery.id)
ORDER BY id, m_id, m_keys_id, m_values_id
```

As in the table creation process, we select all of the *id* values for each object as they are used in the object reconstruction process. Here we name each column representatively, although the actual column names would have been chosen uniquely in the table creation and index query generation step. Also, we order the results by all *id* columns, starting from *State.id* down to *State.m.values.id*. This makes reconstructing objects easier, since consecutive rows in the output with the same value of any of the *id* columns belong to the same object.

### 3.3.5 Object Reconstruction

After executing all of the data extraction queries as described above, we have enough information to satisfy the user's NYQL query. However, the final step is to turn the output from multiple SQL queries back into the objects NYQL returns. Table 3.1 shows the results of our data extraction query above against the database depicted in Figure 3.10.

**Table 3.1**: Sample Data Extraction Query Results

| id | node | time | v | m_id | m_keys _id | m_keys _value | m_values _id | m_values _value |
|---|---|---|---|---|---|---|---|---|
| 0 | A | 1234567890.1 | 24 | 0 | 0 | 0 | 0 | "Bob" |
| 0 | A | 1234567890.1 | 24 | 0 | 1 | 3 | 1 | "Kathy" |
| 1 | C | 1234567890.2 | 28 | 1 | 2 | 0 | 2 | "Bob" |

We parse these results one row at a time, using the *id* columns as indicators where objects begin and end. We start with the highest-level *id* column first and work our way down. Since we do not have a current State object, we make a new one with no children when we find *id* of 0. Next, we look at the columns corresponding to the children of State objects — *node* and *time* — and add children for “time = 1234567890.1” and “node = A” to our State object. Next we see *m_id* is 0, and we create an empty child for *m*. Next up is *m_keys_id*, so we create an empty child of our *m* object named *keys*. We find *m_keys_value* with value 0, so we add *value* as a child to *keys* with value 0. Finally, we similarly add a child for *values* and a grandchild *value* containing “Bob.”

On the second line, we see *id* is still 0, so we know we have not finished constructing our State object, and we skip the *time* and *node* fields. Next we see that *m_id* is also still 0, so we have not finished constructing it yet either. However, this time the value of *m_keys_id* has changed to 1, so we make a new *keys* child under *m*, and add a child for “value = 3.” Similarly, we see that *m_values_id* has changed to 1 as well, so we add another *values* child to *m* and add its child for “value = Kathy.” We have now finished parsing the second line, and when we see that *id* has changed to 1 on the third line, we put our completed State object in a map for *id* 0 and begin constructing a new State object for *id* 1.

This process repeats until we have built a map of objects for every *id* value in the data extraction query. Since there will be one data extraction query per type in the output, we will end up with one object map per type as well. The final step traverses the results of the index query, looking up the appropriate objects in the object map as specified by the output statement. Note that in this step, we may need to return sub-objects of the objects in the object map. For example, we have State objects, but the object statement contains $State.node$, $State.time$, etc. We simply traverse each object to extract the appropriate children. Our NYQL query is now complete, and we display the results to the user in a graphical user interface. We discuss this interface and other functionality it provides in Chapter 5.

### 3.3.6 Advanced Translation

There are a number of circumstances that require more involved query translation than the basic process we just described. These include “let” statements with the “@” operator, loops over multiple objects (e.g., “foreach b in $t_1$, $t_2$” or “foreach b in $prefix*$”), queries returning Event or Path objects, and aggregate functions. We cover each of these cases in turn.

### "Let" Statements and the "@" Operator

Under normal circumstances, we translate each loop and "let" statement in a NyQL query independently. However, when a user uses the "@" operator, it creates a dependence between the "let" statement and the loop whose variable appears on the right hand side of the "@" operator. For example, consider the query shown in Figure 3.12.

```
foreach s in send(Addr src, Msg1& m) {
  let s2 = State[node == s.node, time @ s.time];
  output s2;
}
```

**Figure 3.12**: Example "@" Query

Instead of two nested loops that would produce all pairs of State and send objects, we now only want a single State object for each element in the outer loop. Thus, our output should contain one State object for every send object. In order to capture this relationship, we add an extra step to the translation process and generate a *relation query* that maps *id* values of objects in the outer loop to the correct *id* for the inner loop. The relation query is a nested SELECT statement, shown below. Here the table Outer refers to the table we would create for the send objects, and Inner refers to the table we would create for the State objects as described in Section 3.3.4.

```
CREATE TABLE RQ1 AS
SELECT t1.id AS oid, Inner.id AS iid
FROM (SELECT Outer.id, (SELECT max(Inner.time) FROM Inner
WHERE(Inner.time <= Outer.time AND Inner.node = Outer.node)) AS T
FROM Outer) t1 LEFT JOIN Inner ON t1.T = Inner.time
```

We materialize this query as a table so that any other queries that need this relationship can skip this step. Finally, when generating the index query, we make sure that the *id*s we select match both the inner and outer *id*s of the relation query, as shown below.

```
CREATE VIEW IndexQuery AS
SELECT Outer.id, Inner.id FROM Outer, Inner, RQ1
WHERE (RQ1.iid = Inner.id AND RQ1.oid = Outer.id)
```

### Multiple Loop Targets

When there are multiple object names specified as the target of a loop, as in Example NYQL Query 4, shown in Figure 3.7, we can no longer execute a single set of SQL queries. However, this case is equivalent to executing multiple NYQL queries, each with a single loop object specified. However, if we have multiple loops, each with multiple loop objects, we need a new query for each unique combination of objects. In the general case, we will end up with a number of NYQL queries equal to the product of the sizes of the lists in each loop. In Example NYQL Query 4, there are two "send" object types and two "deliver" object types, so we generate and execute four NYQL queries.

If there is no "order by" clause, we can simply concatenate the results of each NYQL query and present them to the user. On the other hand, if an "order by" clause is specified, we have to combine the results of each of the sub-queries we execute. Luckily, the results of each query are already correctly sorted, so we must simply merge the result of sorted lists, which is efficient. This sorting takes place outside of SQL and happens in a special phase after the object reconstruction phase for each sub-query is completed.

### Constructing Event Objects

As we mentioned in Section 3.1.3, Event objects are composite objects and do not exist in their own tables like other object types. As a result, we need special handling to construct them. At a high level, Event objects occur as defined by entries in the StartEvent table where the *begin* column equals 1 (see Figure 3.10). However, the contents of each Event object are based on the contents of the Events table between each StartEvent begin/end pair.

When a user writes a query that outputs an Event object, as in Example NYQL Query 2, shown in Figure 3.5, we translate the query as if they had chosen to output $StartEvent.id$. Since each Event object corresponds to an entry in the StartEvent table where *begin* = 1, we add an extra clause to the relation query (described above) which ensures we only select the correct entries in the StartEvent table.

The second step occurs during the object reconstruction phase (Section 3.3.5). Each time we process an output for $StartEvent.id$, we know we need to build an Event object corresponding to that *id* value. Since we know that StartEvent entries are paired, we execute a SQL query that selects everything in the Events table between the starting and ending StartEvent entries. We then look at the *tname* column of each row we select. We use "SL" entries to keep track of the current stack depth. For non-"SL" entries, we translate and execute a NYQL query of the form "foreach b in *tname* where (b.id = *joinId*) output b; " We make sure to translate *tname* using the MethodMap table if *tname* appears in the *method* column. The result of executing this query will be the object that was logged at that point in the Event. When we are done processing all of the lines from the Events table, we construct our final Event object by composing each object we reconstructed using the nesting given by the "SL" entries.

### Constructing Path Objects

Like Event objects, Path objects are also composite, as they are composed of Event objects. We begin constructing a Path by translating the user's query as if they were outputting *StartEvent.path* whenever they wish to output Path, as in NYQL Example Query 6, shown in Figure 3.9. Each time we come across a value for *StartEvent.path* when we are reconstructing objects, we execute a SQL query to select all StartEvents that fall on the same path, as shown below.

```
SELECT id FROM StartEvent
WHERE (begin = 1 AND path = $path$) ORDER BY mid
```

The results of this query include the *id* values of all Events on the path we are constructing. We simply execute the Event construction algorithm described above for each of these *id* values and add each Event object constructed as a child of our Path object.

### Computing Event.duration

While Event objects contain all of the objects that were logged between beginning and end event markers, *Event.duration* is a special child of the Event object that is not computed unless explicitly requested, as in NYQL Example Queries 3, 4, and 6 (Figures 3.6, 3.7, and 3.9).

The query below computes each Event duration as the difference in time between each odd-numbered StartEvent with *begin* = 0 and the corresponding even-numbered StartEvent with *begin* = 1. We store the result as a second table created table over StartEvent so that queries may use the duration field in “order by” clauses as in Figure 3.9. This can be very useful in quickly determining which events take the longest to execute.

```
CREATE TABLE EventDurations AS
SELECT A.id, A.event, A.tid, A.node, A.time, A.begin, A.mid,
       A.path, B.time - A.time AS duration
FROM CreatedStartEvent A, CreatedStartEvent B
WHERE (A.id / 2 = B.id / 2 AND A.id % 2 = 0 AND B.id % 2 = 1)
```

## Computing Aggregate Functions

As mentioned in Section 3.2.2, aggregation functions in NYQL exist solely as user-defined functions. In our implementation, the user implements a Java class deriving from *UserDefinedFunc* and provides four methods: update(), finalize(), getName(), and getValue(), described below.

update() takes one parameter of the type the aggregate function is to process. This method is used for updating any internal state the function needs to maintain for each value to be aggregated. finalize() is called when all values are done being processed and is used to do any final computation, if necessary. getName() simply returns the name of the method, and getValue() returns the final aggregated value.

When our tool runs, it checks a specific directory for any class files, all assumed to derive from *UserDefinedFunc*. It then dynamically loads each class file that it finds, and calls getName() on each to determine the set of valid user-defined functions. When a query is executed containing one of these user-defined functions in the “output” statement, we generate a query as if the function were not present. Once we generate our results, we do a post-processing step where we instantiate the appropriate objects for the user-defined function and pass each value to be aggregated to the update() method. This method updates the current state of the aggregate value. After update() is called on each value, we call finalize() to compute the final value. Finally, we call getValue() to obtain the value that is displayed in the user interface.

Chapter 3, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

# Chapter 4

# A Logging Case Study: MACE

In the previous chapter, we described a new abstraction for log file data — the state matrix — and a query langugage for searching and extracting data from this matrix. However, as we mentioned in Chapter 1, log file post-processing is only one component of the debugging process. In this chapter, we discuss the process of log file generation and configuration. In Sections 4.1–4.3, we present a logging framework in the context of MACE [49], a source-to-source translator and set of libraries for building distributed systems. We discuss a simple query-based mechanism for customizing what appears in the log in Section 4.4. Finally, we analyze the performance impact of our logging scheme, both in the baseline case and also under customization in Section 4.5. We begin with an overview of MACE.

## 4.1 MACE Overview

MACE is a meta-language for writing distributed systems. It eliminates much of the tedium commonly required for distributed systems programming, such as socket and connection management, timer scheduling, and message serialization and deserialization. Although most large-scale distributed systems are not written with the aid of a source-to-source translator, let alone with MACE, we chose to use MACE as the basis for our logging framework for a number of reasons. First, we have significant experience with MACE and find it considerably easier to use than coding from scratch. Second, we have found that using a source-to-source translator provides many advantages over library-based approaches. In particular, the ability to control the code generation process has been very powerful, as we will see below. We will return to the more general case of logging without a translator in Chapter 6.

Distributed systems written with MACE are modeled as finite state machines, where each *transition* is implemented as a block of C++ code. Each application is divided into a layered set of *services*, each with a specified API. Services communicate with each other by making *upcalls* and *downcalls* to the layers above and below them. Before getting into the details of our logging framework, we present a simple MACE service we will use as an example throughout this section.

Figure 4.1 shows the skeleton of a tree-building service named RANDTREE. RANDTREE has one service layered below it — a TCPTRANSPORT used for sending messages. The *messages* block defines the set of messages this service can send. In this case, there is a join message and a join response message. The *state_variables* section defines the set of service variables. Our example has two state variables: *parent*, a MaceKey, which is a MACE type that represents an IP address or host name, and *children*, a NodeSet, which is a set of MaceKeys. The *transitions* block defines the set of C++ transitions, analogous to event handlers. Here we implement members of the service's API, including message handlers and timer handlers.

## 4.2 Compiler-generated Logging

In Chapter 2, we described a set of features we believe a good logging/debugging system should support. These features include logging function call parameters and return values, event markers for aggregating log statements, message tracing, and causal path identification. In this section, we describe how we modified MACE's code generator to add each of these features. Note that we were able to add *all* of these features without a *single* line of user-generated code, simply because we had a source-to-source translator at our disposal.

```
service RandTree;

services {
  Transport trans = TcpTransport();
}

messages {
  Join {
    // fields needed for joining
  }

  JoinReply {
    // fields needed for replying
  }
}

state_variables {
  MaceKey parent;
  NodeSet children;
}

transitions {
  downcall joinOverlay(MaceKey peer) {
    // called when application wishes to join the tree
    // C++ code here
    downcall_route(trans, Join(...));
  }

  upcall deliver(MaceKey src, Join msg) {
    // process "Join" message
    // C++ code here
  }

  upcall deliver(MaceKey src, JoinReply msg) {
    // process "JoinReply" message
    // C++ code here
  }
}
```

**Figure 4.1**: Simple MACE Service

### 4.2.1 Logging Function Calls

Since each MACE service declares a set of transitions (see Figure 4.1), a natural point to instrument function calls is at these transition boundaries. Although we cannot

```
upcall deliver(MaceKey src, Join msg) {
  // process "Join" message
  . . .
  return 0;
}
```

(a) MACE Code

```
int upcall_deliver(MaceKey src, Join msg) {
  log_upcall_deliver(src, msg);
  // process "Join" message
  // <original function body>
  return Log::logRv(0, "upcall_deliver");
}
```

(b) Generated C++ Code

**Figure 4.2**: Logging Function Call Parameters and Return Values

automatically instrument *every* function call made in this manner, logging at transition boundaries provides an acceptable balance between logging speed/size of the generated log and debugging usefulness. Figure 4.2 shows the transformation we perform.

There are two elements the MACE translator generates for each transition — a call to log the function call parameters, and a call to log the function's return value. For a transition named $X$, MACE generates a function log_$X()$ with the same prototype as the original function and a void return value. The body of this function contains code to log the values of each of the parameters under an object named $X$. The logRv() function is not generated by the MACE translator but is part of MACE's library. It is a template function with two arguments — the first argument is a template argument containing the value that should be logged, and the second is the name of the object being logged. This function logs the first argument under an object named "rv_*name*" and then returns it. In this example, logRv() would log the value 0 under an object named "rv_deliver." We defer the description of object encoding until Section 4.3.2.

### 4.2.2 Logging Events

In order to implement event logging, we once again took advantage of MACE's natural event-centric structure. We define an event to consist of all of the code executed during each MACE transition. Logging event start and end is slightly trickier than inserting a logging statement at the beginning and end of each transition though, since transitions may call other transitions as subroutines. To handle this issue, we developed

```
class Event {
  static int depth = 0;
  int myDepth;

  Event() {
    myDepth = depth++;
    if (myDepth == 0) {
      // log event start
    }
  }

  ~Event() {
    if (myDepth == 0) {
      // log event end
    }
    depth−−;
  }
}
```

**Figure 4.3**: Event Logging

an *Event* class and have the MACE translator instantiate an Event object at the beginning of each transition, *before* the code to log the transition's parameters.

Figure 4.3 shows C++ pseudocode for the Event class. We use a static integer to keep track of the current stack depth, which is incremented every time an Event object is constructed. A single counter is sufficient, since MACE only allows a single event to take place at a time. If the current depth is 0 when an Event object is constructed, we log that an event is starting. Similarly, we decrement the stack depth each time an Event object is destructed, and log that an event is ending if the depth is 0. We generalize this event-logging mechanism to handle multiple concurrent threads in Section 6.1.4.

### 4.2.3 Logging Program State

In addition to function call parameters, it is often important to log program state at various points. In MACE, this is easy since each service has a *state_variables* block. The MACE translator generates a logging function for each service which logs each of the state variables as a child of an object of the service's name. For example, our

RandTree example from Figure 4.1 would have code generated to log RandTree objects with children named “parent” and “children,” corresponding to RandTree’s two state variables. By default, the Mace translator inserts calls to this logging function after every event.

### 4.2.4 Message Tracing

Message tracing is one of the most important tools for debugging distributed systems, yet also one of the easiest to implement. Recall that for each message sent, we would like to know when and where the message was sent, when and where it was received, and what the contents of the message were. These first two events are already recorded in Mace due to the function call tracing described above. In order to match each message sent with its receipt, we have the Mace compiler add a monotonically increasing integer value to each message. This value is logged both when the message is sent and when it is received, so given a tuple of $(src, dest, messageId)$, we can match each message sent with where it is received. For uses of message tracing, see Chapter 5.

### 4.2.5 Causal Path Reconstruction

At a minimum, we could reconstruct causal paths with only event logging and message tracing. However, this approach would be inefficient, since we would have to trace each individual message. Instead, we propagate a path identifier along with the aforementioned message identifier with each message sent. The path identifier is logged as part of the event, so it is critical that it is set *before* for the event begins for message receive events. If no path identifier is set when an event begins, it is assumed that the current event is the first in the path, and a new path identifier is chosen.

## 4.3 General-purpose Object Logging

Although we are able to take advantage of the Mace translator to insert many logging calls automatically, there will always be cases where users wish to add their own, more specific log statements. By default, Mace provides a number of logging

```
structured_logs {
  logMsg1(const vector<int>& choices, int chosen);
  logString(const string& msg);
}

// code
vector<int> v = ...;
int ch = ...;
// log a vector and an integer as children of an object named logMsg1
logMsg1(v, ch);
// log a string as a child of an object named logString
logString("This log function can do normal string logging");
```

**Figure 4.4**: MACE Structured Logging Example

methods similar to C's printf() and C++ streams. Unfortunately, none of MACE's logging methods were suitable for our needs for NYQL. Since NYQL's state matrix is composed of objects, any logging API that only supports string logging will not suffice.

### 4.3.1 Structured Logging

We present a new mechanism called *structured logging* for MACE, which allows users to log collections of objects instead of just strings. To use structured logging, users define a code block in their MACE service. Inside this block, the user can define a set of function prototypes, each corresponding to a type of object they wish to log. Then, the user calls one of their newly-defined functions to log the desired values.

Figure 4.4 shows an example of structured logging in MACE. In this example, we define two function prototypes, logMsg1() and logString(). The first function, logMsg1(), takes a vector of integers and an integer as parameters. When a user calls logMsg1(), the vector and the integer passed as parameters will be logged as children of an object named logMsg1. We also show logString() as an example of traditional string logging.

Implementing structured logging in MACE once again makes use of the source-to-source translator. We generate a function corresponding to each of the prototypes located in a structured logging block with the same function signatures. As a result, if a user makes a mistake calling one of their structured logging functions by passing the wrong number of arguments, or arguments that are of the wrong type, the generated C++ code will not compile. The type safety provided by this mechanism is very useful in ensuring that unexpected objects do not show up in the log.

### 4.3.2 Logging Arbitrary Objects

So far, we have made the assumption that we can log objects of arbitrary type, both when logging function call parameters (Section 4.2.1) and in structured logging (Section 4.3.1). In this section, we describe how object logging is handled.

We handle the physical serialization of objects to a log file with a set of specialized template functions. Objects like basic types and strings are easy to handle, as there are a fixed set of them and they are well known. In addition, we provide specializations for many of the common C++ STL container classes, such as list, vector, map, and hash_map, among others. These specializations alone are enough for many logging purposes, but if a user wishes to log objects of their own types, we provide an interface that their classes must implement. In this case, the user has to implement the serialization themselves for the class, but frequently this code will simply consist of calls to log the first class member, then the second, etc.

## 4.4 Log Customization

While generating a large state matrix is powerful for developers, the required logging does come at a cost. Unfortunately, the overhead caused by logging program state after every event, the contents of every message sent/received, etc., can be high and may mask performance problems or make it impossible to leave logging "always on." As a result, it is imperative that we provide developers with a simple way to control the amount of logging overhead.

Although many large software projects have methods to selectively control what gets logged, we developed a simple, query-based approach. Specifically, we allow developers to write queries in a special query block in their MACE services. If such a block exists in a service, the MACE compiler will aim to minimize the amount of logging generated while still being able to satisfiy the given queries. Otherwise, if no queries are

```
queries {
  foreach j in joinOverlay(MaceKey peer) {
    output j;
  }
  foreach d in deliver(MaceKey src, Join msg) {
    output d;
  }
}
```

(a) Queries Block

joinOverlay(MaceKey peer)
deliver(MaceKey src, Join msg)
~~deliver(MaceKey src, JoinReply msg)~~
~~RandTree~~
Event

(b) Logged Objects

**Figure 4.5**: Eliminating Entire Objects with Queries

given, all logging is enabled by default. There are three optimizations our query-based customization supports. We describe each in turn below.

### 4.4.1 Eliminating Uninteresting Objects

Our first optimization prevents entire unnecessary objects from appearing in the log. Suppose our RANDTREE service contained the queries block shown in Figure 4.5(a). In this case, the user is interested in seeing all occurrences of the joinOverlay() method as well as all occasions when a Join message was received, but does not care about any other function calls or the service's state itself. As a result, the MACE compiler needs to only generate logging statements for the joinOverlay() calls as well as the particular call to deliver() with a Join message. We note that Event objects are still logged, as they are fundamental to resolving all queries. Figure 4.5(b) shows the set of all objects normally logged by our RANDTREE service, with those eliminated by the above queries block crossed out.

### 4.4.2 Eliminating Uninteresting Sub-objects

We can further optimize what gets logged by looking at the specific members of each object that are present in the *output* statement of each query. For instance, consider the queries block in Figure 4.6(a). In this case, the user has expressed interest

```
queries {
  foreach r in RandTree {
    output r.children;
  }
  foreach d in deliver* {
    output d.src;
  }
}
```

(a) Queries Block

```
~~joinOverlay(MaceKey peer)~~
deliver(MaceKey src, Join msg)
  src
  ~~msg~~
deliver(MaceKey src, JoinReply msg)
  src
  ~~msg~~
RandTree
  ~~parent~~
  children
Event
```

(b) Logged Objects

**Figure 4.6**: Eliminating Sub-objects with Queries

only in the *children* member of the RandTree object, as well as the *src* child of *any* delivered message. Figure 4.6(b) shows the set of objects that will be logged as a result of this queries block. Calls to joinOverlay() will not be logged at all. For deliver() methods, only the *src* parameter will be logged — the contents of the messages will not appear. Similarly, when RANDTREE's state is logged after every event, only the *children* member will appear in the log. This particular optimization can greatly reduce the logging overhead if a user is only interested in small portions of otherwise large objects, such as program state and network messages.

### 4.4.3 Conditional Logging

Our first two customizations allow us to control which objects and sub-objects get logged, but once this decision is made, the objects that appear in the queries block will *always* be logged. We improve on our past two optimizations by allowing the user to express that an object should only be logged *sometimes*, naturally using an expression in the form of a *where* clause.

Figure 4.7(a) shows an example queries block with *where* clauses. In this example, the user is only interested in RANDTREE's state if the *children* member has more than two elements. The user is also interested in seeing all network messages sent from

```
queries {
  foreach r in RandTree
    where (r.children.size() > 2) {
    output r;
  }
  foreach d in deliver*
    where (d.src == "10.0.0.1") {
    output d;
  }
}
```

(a) Queries Block

```
~~joinOverlay(MaceKey peer)~~
if (src == "10.0.0.1")
  deliver(MaceKey src, Join msg)
if (src == "10.0.0.1")
  deliver(MaceKey src, JoinReply msg)
if (children.size() > 2)
  RandTree
Event
```

(b) Logged Objects and Conditions

**Figure 4.7**: Conditional Logging with Queries

a particular address. Instead of logging RANDTREE state after every event and all network messages as in previous examples, we insert the conditions specified in the *where* clauses into the generated code before the appropriate objects are logged. We show how we use these expressions to test whether objects should be logged in Figure 4.7(b). As before, calls to joinOverlay() are not logged since the joinOverlay object does not appear in any queries in the queries block.

We note that in this example, we used the clause "r.children.size() > 2" for our condition on RandTree objects, even though it is not valid NYQL since we currently do not support function calls in *where* clauses. However, we relax our usual syntax here to support any valid C++ expression since we are simply inserting it directly into the user's code. As a result, we gain additional flexibility in specifying when an object should be logged.

### 4.4.4 Dynamic Binary Rewriting

Unfortunately, each of the three previous optimizations require users to recompile their applications to obtain the benefits of reduced logging. To eliminate this constraint, we developed a solution using Pin [59], a dynamic binary rewriting toolkit.

Recall that we generate a logging function for each of a service's transitions (Section 4.2.1). For every function of the form log_$objName(p_1, \ldots, p_m)$ we generate, we now also generate a function shouldLog_$objName(p_1,\dots,p_m)$. These “shouldLog” functions all return true by default, indicating that the corresponding object should always be logged.

```
upcall deliver(MaceKey src, Join msg) {
  // process "Join" message
  ...
  return 0;
}
```

(a) MACE Code

```
int upcall_deliver(MaceKey src, Join msg) {
  if (shouldLog_upcall_deliver(src, msg)) {
    log_upcall_deliver(src, msg);
  }
  // process "Join" message
  // <original function body>
  return logRv(0, "upcall_deliver");
}
```

(b) Generated C++ Code

**Figure 4.8**: Generated Code Supporting Dynamic Binary Rewriting

Figure 4.8 shows how we use our shouldLog functions to conditionally execute our generated logging functions. This figure is similar to Figure 4.2, except now we only call log_upcall_deliver() if shouldLog_upcall_deliver() returns *true*. It is these shouldLog functions that we use Pin to dynamically rewrite, allowing debuggers to change logging behavior on the fly.

Figure 4.9 shows the general architecture of our approach. In the first step, we give a set of queries of interest to a parser, much like we gave queries to the MACE compiler in our previous sections. The parser extracts a set of object names $\{o_1,\dots,o_n\}$ and expressions $\{e_1,\dots,e_n\}$ from the queries, assigning an expression of *true* to any object without a *where* clause specified. In the second step, we need to translate from object names to the mangled `C++` names of the shouldLog functions we wish to modify. To do this, we run `nm` [16] on the application’s executable that we wish to modify and search for symbols matching shouldLog_$o_1$, shouldLog_$o_2$, etc. It is important to note that the application *must* be compiled with debug symbols, or this step will not work.

Once we have the mangled `C++` symbol names, we give these along with the expressions from the first step to a code generator, which generates a `C++` Pin tool used to specify *n* new shouldLog functions to replace the existing ones — each new

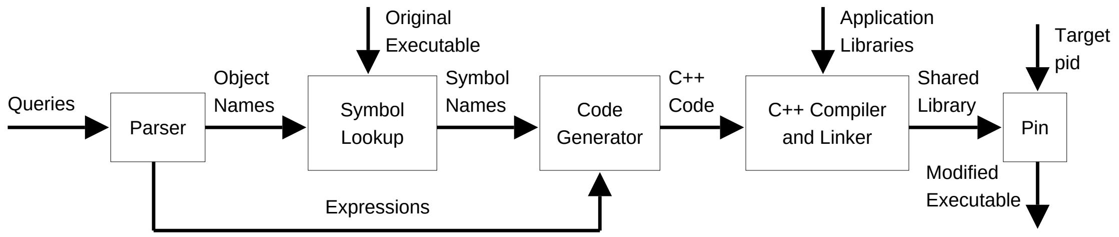


**Figure 4.9**: Query-based Dynamic Binary Instrumentation

shouldLog_$o_i$ simply returns $e_i$. We use Pin's RTN_FindByName() and RTN_ReplaceSignatureProbed() functions to handle the replacement. Care must be taken in this step to generate code that includes any source files defining types that are passed as parameters to the shouldLog functions, or the generated code will not compile. Next, we compile the generated `C++` code to produce a shared library that can be used with Pin. Similar to the previous step, libraries containing the implementations of any included header files must be provided for successful linking. Finally, the generated shared library is given to Pin, along with the process id of the running target application. At this point, the running application will begin executing the new code specified by the shared library, thus modifying the behavior of the logging subsystem.

Although this process does allow users to change logging behavior on an already-running application, it does have its drawbacks. We have already mentioned that the application must be compiled with debug symbols, but generating the shared library to reconfigure the logging can take anywhere from a handful of seconds to a minute or more, depending on the number of functions to rewrite and on the complexity of any included code needed for type definitions. As a result, binary rewriting cannot be used as an instantaneous solution, but it can be quite useful nonetheless.

### 4.4.5 Probabilistic Path Logging

In addition to the query-based logging customizations above, we also present a probabilistic logging scheme based around causal paths. The idea here is that we might not need every logging statement to locate a particular bug, but we would still like a representative sample of paths. In this scheme, the user chooses a probability $p$ that each path should be logged. When each path begins, either all logging will be generated as normal (with probability $p$), or no logging will take place (with probability $1 - p$).

We use a single boolean variable *logPath* to keep track of whether the current event should be logged, which is sufficient for MACE's single-threaded event model. We modify the generated code to only perform a logging statement if *logPath* is *true*. When the first event in a path begins, we set *logPath* to *true* with probability $p$ and *false* otherwise. Any time a message is sent, the current value of *logPath* is included and a node receiving a message sets its *logPath* variable to the value that was sent. As a result, all logging that occurs on each path will be based on the same value of *logPath*, chosen at the start of each path. We extend this approach to a multi-threaded environment in Section 6.1.6.

## 4.5 System Performance

In this section, we present a performance evaluation of all aspects of our debugging system, including logging overhead, post-processing, and NYQL query evaluation. All log files used in these experiments were generated by MACE applications employing the techniques we described in this chapter. We present results from two systems, RANDTREE and PAXOS [55]. See Sections 5.2.1 and 5.2.2 for descriptions of these systems.

### 4.5.1 Baseline Performance

As discussed in Section 3.3.1, using a SQL database as a back-end is not the only possible implementation of the state matrix abstraction. In this section, we quantify the effects of this choice on the performance of query resolution, both in terms of startup costs and query evaluation itself. All results were generated on a Pentium 4, 2.8Ghz machine with 1GB of RAM. Table 4.1 presents a summary of our results with details below.

Our RANDTREE evaluation is based on a dataset collected from a run of 30 nodes. In this experiment, all 30 nodes try to form a single tree. The run completes when all nodes manage to join. The joining process resulted in 1.1 MB of log files.

**Table 4.1**: RANDTREE and PAXOS Query Overhead

| | RandTree | Paxos |
|---|---|---|
| Log size | 1.1 MB | 921 MB |
| Database size | 9.2 MB | 2.49 GB |
| Log processing | 10 s | 947 s |
| Database create | 9 s | 900 s |
| Query 1 cold | 2 s | 21 s |
| Query 1 repeated | 1 s | 7 s |
| Objects returned | 30 | 20,000 |
| Query 2 cold | 24 s | 27 s |
| Query 2 repeated | 18 s | 2.3 s/2.7 s/5.7 s |
| Objects returned | 314 | 10/100/1000 |

Preparing the log files for insertion into the database as described in Section 3.3.3 took 10 seconds. We next measured the time to copy the data from all log files into the database, which took 9 seconds. The resulting database was 9 MB on disk, as reported by the *pg_database_size()* method of PostgreSQL. Thus, our one-time setup cost was around 20 seconds for this run of RANDTREE.

We then ran two different queries, shown in Figure 4.10, and measured their execution times. Recall that the first time a query returns an object type that has not been queried before, we create a table for the object that joins all of the child objects together into a single table (Section 3.3.4). This table is then used in all further queries for that object type. Thus, we report both the running time of each query's first execution, in addition to subsequent executions. The first RANDTREE query extracts the causal paths for each call to *joinOverlay*. This query creates three tables and completes in only two seconds the first time it runs. Subsequent executions of this query complete in one second. The query returns a total of 30 objects, one for each node.

The second query we measured for RANDTREE involved outputting the entire event, including function call parameters, each time any message was delivered. The first run of the query took 24 seconds, requiring 16 table creations. Repeated executions of this query require 18 seconds per query on average. This query returns 314 objects to the user.

The second data set we collect is from PAXOS, a more complicated protocol than RANDTREE. Our test involved three nodes, where one continuously tried to propose

```
// RandTree Query 1
foreach j in joinOverlay(MaceKey peer) {
  let p = Path[node == j.node,
               time @ j.time];
  output p;
}

// RandTree Query 2
foreach d in deliver* {
  let e = Event[node == d.node,
                time @ d.time];
  output e.params;
}
```

```
// Paxos Query 1
foreach d in deliver(MaceKey src,
                     Accepted msg) {
  output d;
}

// Paxos Query 2
foreach d in deliver(MaceKey src,
                     Chosen msg) {
  let e = Event[node == d.node,
                time @ d.time];
  output e.noparams order by d.time limit 10;
}
```

**Figure 4.10**: RANDTREE and PAXOS Queries Used in Table 4.1

new values as quickly as possible. We ran the system until 10,000 proposals had been completed, reulting in 921 MB of log files. The log merging process took approximately 15.75 minutes, followed by another 15 minutes to load the database. While these times may approach or exceed what developers consider a reasonable time to wait, we argue that this dataset is on the large end of the scale. With more restrictive logging of program state coupled with more efficient log pre-processing or more recent hardware, we could reduce start-up costs considerably. The resulting database was 2.49 GB according to *pg_database_size()*.

The first PAXOS query extracts all function objects corresponding to delivering an *Accepted* message. The query time on cold start was 21 seconds, and necessitated the creation of two tables. Subsequent runs of the same query completed in seven seconds. The query returned 20,000 objects — far more than a user typically should retrieve with a single query.

Our second query extracts the first 10 events (excluding parameters) corresponding to the delivery of *Chosen* messages. Each event consisted of a nested sequence of nine function calls. The first run of this query took 27 seconds and required creating three tables. Subsequent runs took only 2 seconds, and increasing the number of events returned to 100 and then 1000 took only 3 seconds and 6 seconds, respectively.

**Table 4.2**: PAXOS Throughput with Various Logging Enabled

| Logging | Query | Tput |
|---|---|---|
| None | N/A | 2700/s |
| All | N/A | 122/s |
| All *Accepted* messages | **foreach** b in deliver(MaceKey src, Accepted msg) {<br>  **output** b;<br>} | 789/s |
| Every 3rd *Accepted* message | **foreach** b in deliver(MaceKey src, Accepted msg)<br>    **where** (msg.applied %3 == 0) {<br>  **output** b;<br>} | 855/s |
| All PAXOS state | **foreach** b in Paxos {<br>  **output** b;<br>} | 156/s |
| Some PAXOS state | **foreach** b in Paxos {<br>  **output** b.acceptedProposals;<br>} | 736/s |

### 4.5.2 Logging Optimizations

In addition to the baseline system, we also evaluate the impact of reducing the enabled logging via the customizations described in Sections 4.4.1–4.4.3 and 4.4.5. We focus on PAXOS because the baseline logging overhead for RANDTREE is sufficiently small that additional optimizations are not necessary to maintain good performance. For these experiments, we used three dual-core 2.8 GHz Xeons running one instance of PAXOS each.

To evaluate the customizations in Sections 4.4.1–4.4.3, we ran PAXOS six times under various conditions: once with all logging disabled, once with full logging enabled, and then four times with logging selectively enabled by queries given to the MACE compiler. Table 4.2 presents our performance results. Full logging signifincantly impacts performance — dropping throughput from 2700 proposals per second to just over 120. If we just log PAXOS state after every event, we can make a marginal improvement in throughput (row 5). However, reducing the logged state to a single map (accepted-Proposals) gives much better performance (row 6). Logging only one type of message gives a large performance boost (row 3), and an even larger one when further filtering messages based on content (row 4).

For probabilistic path logging, we ran the system with all logging enabled for

**Table 4.3**: PAXOS Throughput and Log Size with Probabilistic Path Logging

| $p$ | Throughput | Log Size |
|---|---|---|
| 1.0 | 122/s | 830 MB |
| 0.5 | 238/s | 409 MB |
| 0.33 | 355/s | 261 MB |
| 0.1 | 950/s | 83 MB |
| 0.05 | 1430/s | 43 MB |
| 0 | 2700/s | 0 MB |

various logging probabilities $p$. Once again, we used three nodes and ran the system until 10,000 proposals had been made. Table 4.3 shows our results. Probabilistic logging does not affect performance in the base cases of $p = 1$ and $p = 0$ (relative to Table 4.2). Further, we obtain close to linear speedup down to $p = 0.1$ and nearly linear log savings throughout.

These four optimizations show that application developers can flexibily control what to log. The difference in performance seen can be great — enough to unmask bugs not seen at slower speeds, as well as allowing stable, long-running applications to potentially maintain critical, always-on logging.

## 4.6 Summary

In this chapter, we covered many aspects of logging support needed for NYQL processing. We described how we implemented many NYQL features, including event aggregation, message tracing, causal path reconstruction, and function call tracing in MACE, a meta-language and set of libraries for building distributed systems. We have shown how source-to-source translators can insert the vast majority of logging calls needed for NYQL queries automatically, although they are not required for a system to be queried with NYQL.

In addition, we presented a number of query-based techniques for controlling what gets logged. We showed, through evaluation, that these techniques are effective in mitigating the logging overhead by reducing the amount of state that gets logged. We also showed that our implementation of NYQL is fast enough to resolve a number of useful queries. In the next chapter, we describe our experiences using NYQL to debug some real systems.

Chapter 4, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

# Chapter 5

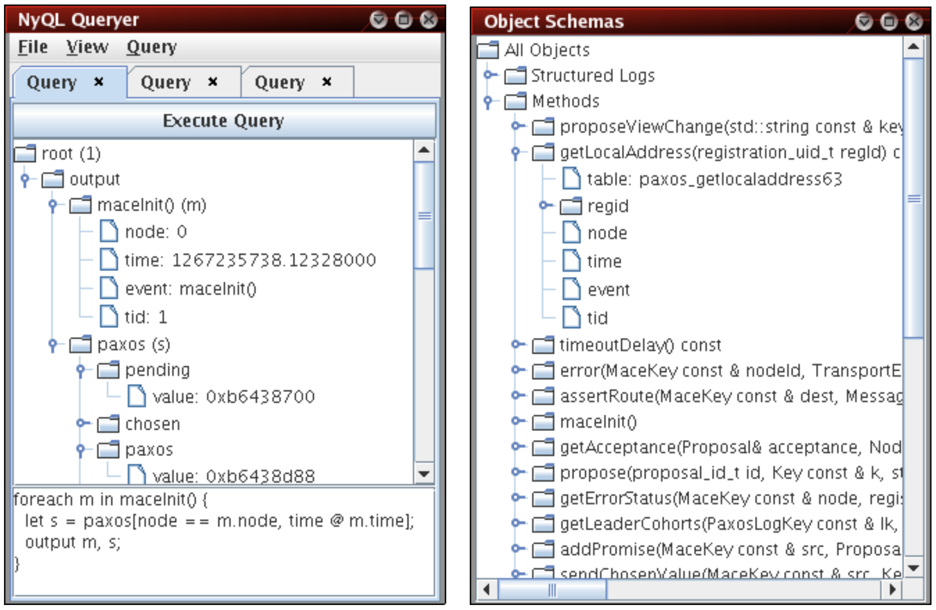
# Debugging Experiences

Thus far, we have presented NYQL, a new query language for distributed systems debugging and have shown that it can resolve a number of queries efficiently. However, languages like NYQL are only useful if they can be used to debug real systems. There are a number of additional factors besides the language itself that must be considered in order to construct a useful debugging tool.

In this chapter, we present the design of our NYQL debugging environment and describe a number of features that we believe make it useful and convenient. We then describe four bugs in two different distributed systems that we were able to diagnose and fix using our tool. Finally, we introduce a comprehensive debugging tool — Nebula— which incorporates query processing along with a *visual event graph* and describe some debugging experiences with it.

## 5.1 An Interface for Queries

The user interface is a critical aspect of a debugging tool. Even if the underlying technology is powerful and robust, all is lost if a user cannot interact with it in an intuitive manner. Thus, we set out to design an interface that was simple, yet powerful. The first decision we faced was whether to make the tool text-based or graphical. We decided on the latter for two main reasons: (i) it is much easier to navigate and view hierarchical objects in a graphical environment, and (ii) users will need to manage the results from potentially many queries simultaneously, which is difficult to do in a



```
root (1)
  output
    maceInit() (m)
      node: 0
      time: 1267235738.12328000
      event: maceInit()
      tid: 1
    paxos (s)
      pending
        value: 0xb6438700
      chosen
      paxos
        value: 0xb6438d88
```

```
foreach m in maceInit() {
 let s = paxos[node == m.node, time @ m.time];
 output m, s;
}
```

```
All Objects
  Structured Logs
  Methods
    proposeViewChange(std::string const & key
    getLocalAddress(registration_uid_t regId) c
      table: paxos_getlocaladdress63
      regid
      node
      time
      event
      tid
    timeoutDelay() const
    error(MaceKey const & nodeId, TransportE
    assertRoute(MaceKey const & dest, Messag
    maceInit()
    getAcceptance(Proposal& acceptance, Nod
    propose(proposal_id_t id, Key const & k, st
    getErrorStatus(MaceKey const & node, regi
    getLeaderCohorts(PaxosLogKey const & lk,
    addPromise(MaceKey const & src, Proposa
    sendChosenValue(MaceKey const & src, Ke
```

(a) Main Interface (b) Object Schemas Window

**Figure 5.1**: NYQL User Interface

console application.

Our interface is implemented in Java and has two components — an *input box*, where the user inputs a query, and an *output pane* which displays a tree view of the user's query results. The main interface can be seen in Figure 5.1(a). To allow users to keep track of multiple queries and query results simultaneously, we wrap the input box and output pane inside of a tabbed pane. Each tab maintains a single query and its results, so even after a query is executed, the user has a record of what generated the results. Finally, each tab contains a button to execute the query currently in the input pane. When clicked, the results will replace whatever is currently in the output pane.

The output pane contains a single node named "root," with the number of objects returned in parentheses. Each child of the root node contains all of the objects specified in a user's "output" statement and is labelled "output" unless a "label by" clause is present (see Figure 3.5 for an example). In our example screenshot, the query returned a single result consisting of two objects — one for maceInit (m) and one for paxos (s). Here we see the node, time, event name, and thread identifier of the maceInit() function call, as well as part of a paxos object.

The second part of our interface, the *Object Schemas Window*, can be seen in Figure 5.1(b). The Object Schemas Window can be accessed from the View menu of the main interface, and serves as a record of all of the available objects in the system, along with their schemas. Since we use entire function prototypes as object names, it is critical that users can retrieve these function prototypes easily, without having to remember them or go look back in their source code. The Object Schemas Window also keeps track of the schemas for any structured log objects the user has defined (see Section 4.3.1). This window eases the debugging process since users can see exactly what data is available to be queried.

Finally, although we believe query-based debugging is a powerful tool, we acknowledge there will always be cases where a concept cannot be expressed in NYQL, or additional text-based debugging tools would be useful. For these reasons, we have incorporated support for exporting any query's results as a text file for additional processing.

## 5.2 Debugging with NYQL

In this section, we describe our experiences using the NYQL Queryer to debug two distributed systems written in MACE— a protocol for building random trees (RANDTREE), and an implementation of PAXOS [55], a distributed consensus protocol. We give overviews of both systems below.

Once we identified a run with a bug, we loaded the logs into our database and used the NYQL Queryer to explore the system. We describe the queries we made and how they allowed us to quickly determine the cause of each bug. Finally, we describe the fixes for each bug we found.

### 5.2.1 RandTree

RANDTREE is a protocol for constructing random trees. The basic protocol is very simple. A node tries to join by contacting another node in the tree, who forwards the message to the root. If the root has not reached its maximum allowed number of children, it accepts the new node as a child. Otherwise, it randomly forwards the join message to a child. The process continues recursively until a suitable parent is found for the new node. There are two notable implementation details. First, RANDTREE is implemented such that if node *A* tries to join another node with a lower IP address *B*, *B* will respond with a JOIN_ROOT_FORCE message, and then immediately try to join *A*. This mechanism breaks ties between two nodes potentially trying to join each other. Secondly, a recovery timer is used as a fail-safe mechanism for recovering from tree partitions, generally configured to fire every 10 seconds. In each of our error cases, we saw that it took 10 seconds for the nodes to form a tree, which suggested that they were not correctly joining until this timer fired.

```
foreach b in deliver*, route* [node == X] {
  let z = RandTree[node == b.node, time @ b.time];
  output b, z order by b.time;
}
```

**Figure 5.2**: Message/State Extraction Query for RANDTREE

### Bugs

We used our tool to help fix three bugs in RANDTREE. Although we suspected that the recovery timer was “fixing” the broken join process, we first validated this hypothesis by writing a query to output the call stacks across all nodes every time the recovery timer fired. This result showed that the recovery timer was indeed firing. Next, to find who was failing to join the tree promptly, we executed a query to extract all objects corresponding to functions where a message was either sent or received across all nodes where the time was greater than $start$ + 5 seconds. The results of this query showed us who was sending late “join” messages. From here, we ran a similar query to extract all of the messages sent/received on the slow node along with the node’s state at that time. We will refer to this query as the *message/state extraction query*, shown in Figure 5.2.

In the first case, we see that node *A* sends a join message to node *B*, but never receives a response. Next, we ran the message/state extraction query on node *B*, and saw that it receives the join message from *A*, but does nothing in response since it is in the "init" state and configured to drop all messages until it itself tries to join the tree.

To fix this bug, we have nodes buffer all join messages while they are in the "init" state. Then, when the node tries to join the tree, we process all of the buffered join messages and respond to them appropriately.

Our second case involved a particular interleaving of messages between three nodes, yet we were able to debug this case entirely with the message/state extraction query. Node *A* sends a join message to *B*, and then receives a join message from *C*, who has a lower IP address than *A*. *A* then tries to join node *C*. Next, *A* receives the join reply from *B*, and since *B*'s IP is higher than *A*'s, responds with a JOIN_ROOT_FORCE. At this point, *A* knows that *B* will try and join it, and changes its state to "joined." Finally, the join response from *A* comes in, but *B* does nothing with it since it thinks it is already joined and the join response did not come from its parent (which does not exist). As a result, *A* thinks it is joined, but it has failed to join the rest of the tree and is instead the root of its own subtree, a condition not corrected until the recovery timer fires.

The fix for this bug was simple — a node does not ignore a join response when it is already joined if it does not currently have a parent.

```
foreach b in RandTree[time @ LOOP_MSG_RCV_TIME] {
  output b;
}
```

**Figure 5.3**: All-node State Extraction Query for RANDTREE

For the third scenario, we once again started with the message/state extraction query. This time, node *A* failed to join correctly because it received a JOIN_LOOP_ERROR — a message stating that it tried to join a node that thought node *A* was the root, which would lead to a loop in the tree. First, we wrote a query to extract the state of all nodes at the time the JOIN_LOOP_ERROR was sent, shown in Figure 5.3. Inspecting the state at each node allowed us to reconstruct the structure of the tree shown in Figure 5.4, as indicated by the solid arrows. Note that the arrows correspond to the state seen at the *parent* node, and not necessarily the child. At this point, we notice that two nodes seem to think node *D* is a child, so we ran the message/state

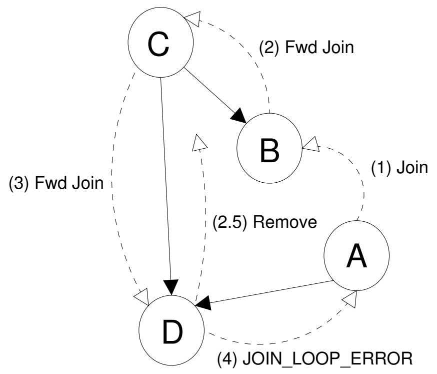


**Figure 5.4**: Setup for RANDTREE Bug Three

extraction query for node *D* and discovered that *D* was in the process of sending a "remove" message to *C* when it received *A*'s forwarded join. Since *D* thinks *A* is currently the root, it sends the JOIN_LOOP_ERROR to *A*.

We fixed this bug by checking if a forwarded join arrived through a different root node than what is currently believed to be the root. If it is, we re-route the message back to the root the message arrived through (node *C*). Then, node *C* will process the remove message from *D*, receive the forwarded join message, and send it to node *B*. Finally, node *B* will accept *A* as a child.

### 5.2.2 Paxos

PAXOS is the classic meta-algorithm for maintaining a globally consistent view of distributed state. Our test application consists of three nodes running a leader-based variant where only a single node proposes values and multiple concurrent proposals are possible. Each proposal is identified by an *index*. Typically, the leader sends a *Propose* message indicating the index at which it wishes to propose a value to the other nodes, who respond with an *Accepted* message. Once the leader receives at least one *Accepted* message, it sends a *Chosen* message to the other two nodes containing the chosen value and the highest index that has been been agreed upon so far. One important implementation detail is that the version of PAXOS we test periodically exchanges information about the highest-seen and highest-applied index with the other nodes in the system for garbage collection and failure recovery purposes.

### Bugs

Here, we describe a single, complex bug. When we ran PAXOS, we noticed that it would generally maintain a steady rate of proposals, except around 10 to 20 seconds into the run it would experience a momentary drop in throughput. Similar to RANDTREE, we suspected this had something to do with a timer. First, we wrote a query to extract all occasions where a timer fired. We noticed that a timer did indeed fire shortly before the performance drop. Next, we retrieved the Event object corresponding to the timer handler method to see what the node did next. We saw that it sent an *Indices* message to the other two nodes. Although we knew that nodes typically respond to *Indices* messages with one of their own, we checked if this was the case. We ran a query to extract all times *Indices* messages were delivered along with the corresponding Event objects on the other two nodes. In both cases, the nodes responded with another *Indices* message.

At this point, we more closely examined the processing of *Indices* messages, since once a node sends a message of that type and receives one in return, that chain of events ends. From the *Indices* message call stacks, we saw only one other function was called – *scavenge()*. Next, we looked at the source code for the *scavenge()* method and noticed it iterating through a loop, erasing things from a list that matched certain characteristics. We hypothesized that perhaps the list was getting too large, so we kept a count of how many elements were being removed from the list and added a structured log object to log this value.

After we re-ran the system with the new logging, we wrote a query to extract all occurrences of our new structured log object, sorted by the value that was logged. We saw that most values were at or near zero, but there were a few outliers over 1000. This seemed suspicious, but since *scavenge()* is called from multiple places in the code, we needed more detail to be sure these outliers were our culprit. Next, we wrote a query to extract the Event containing each structured log entry we found where the value was greater than 1000. Although writing this query was trivial with our system, it would have taken a considerable amount of time to write a script to extract this information, let alone time to debug the script itself. As suspected, we saw that these calls originated from the receipt of an *Indices* message. Looking at the timing information for that function, we saw that it took an unusually large amount of time relative to other "normal" calls to *scavenge()*.

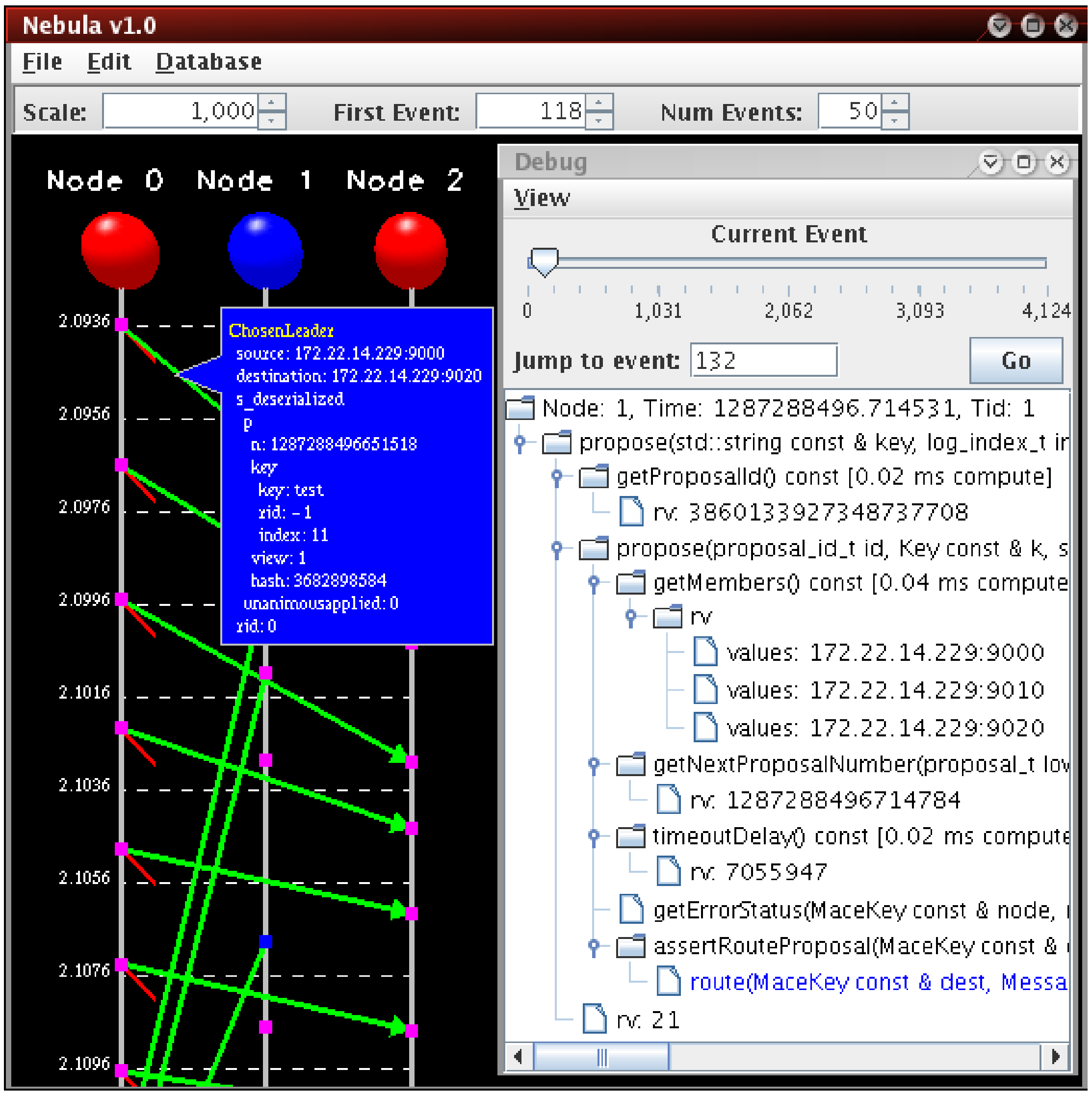
Although we had found the cause for our slowdown, we still had not nailed down the cause of the list's large size. We then wrote a query to extract the contents of the list at times prior to the slowdown, and noticed that it got progressively larger. However, this was not the intended behavior. We searched the source code to find places where the list was modified before realizing that the function that processed *Accepted* messages was returning in one particular case before it could process and remove elements from the list. This caused the list to grow inordinately large. After adding code to process the list before returning, we found that we no longer experienced slowdown.

### 5.2.3 Discussion

The above case studies give a sense of the rapid, iterative exploratory process enabled by our tools. We have found it especially useful and convenient to search for message send/receive events. Being able to reconstruct Event objects around any other logged object has also been useful. While we have a tremendous amount of information available without writing *any* manual logging statements, structured logging is another powerful tool for logging temporary variables or breaking events down on a sub-function level.

## 5.3 Nebula and the Visual Event Graph

We have found that debugging with NYQL can significantly reduce a lot of common burdens associated with debugging distributed systems. However, in our experiences with it, we also found that there were certain patterns that grew increasingly common. For example, we relied on the message/state extraction query (Figure 5.2) in many of our debugging sessions. Although NYQL queries were sufficient for us to debug the examples we described in the last section, it would have been useful to be able to visualize communication patterns between nodes. Instead, we were forced to write a query to see the messages a node sent to others, and then write additional queries to inspect the receiving nodes' state.



```
ChosenLeader
 source: 172.22.14.229:9000
 destination: 172.22.14.229:9020
 s_deserialized
  p
   n: 1287288496651518
   key
    key: test
    rid: -1
    index: 11
   view: 1
   hash: 3682898584
  unanimousapplied: 0
 rid: 0
```

```
Node: 1, Time: 1287288496.714531, Tid: 1
  propose(std::string const & key, log_index_t ir
    getProposalId() const [0.02 ms compute]
      rv: 3860133927348737708
    propose(proposal_id_t id, Key const & k, s
      getMembers() const [0.04 ms compute
        rv
          values: 172.22.14.229:9000
          values: 172.22.14.229:9010
          values: 172.22.14.229:9020
      getNextProposalNumber(proposal_t lov
        rv: 1287288496714784
      timeoutDelay() const [0.02 ms compute
        rv: 7055947
      getErrorStatus(MaceKey const & node,
      assertRouteProposal(MaceKey const &
        route(MaceKey const & dest, Messa
    rv: 21
```

**Figure 5.5**: Nebula User Interface

To simplify the process of traversing messages and viewing node state, we developed Nebula, a graphical user interface which combines the NyQL Queryer interface with an event graph. Our event graph is visually similar to the one presented in [52] and is based on the space-time display introduced by Lamport in [54]. However, our implementation provides a number of additional useful features, such as visualization of all of the logging calls made at each event and the contents of message parameters, the ability to view a causal path for any event in the system, and the ability to see any node's state at any point.

Figure 5.5 shows two of the four main components of the Nebula debugging interface — the event graph representation and the *debug window*, used to visualize Event objects. The other two components are the NYQL Queryer, described in Section 5.1, and the *state-view window*, shown in Figure 5.6. We describe each of these components below.

### 5.3.1 Event Graph

The Nebula event graph has six main elements: (i) node markers, (ii & iii) complete and incomplete message arrows, (iv) message content popups, (v) event markers, and (vi) time markers. Nodes are represented as red spheres and are laid out horizontally across the screen. Each has a gray *time line* emanating from the sphere's center that extends towards the bottom of the screen and represents the flow of time. Time increases with the distance from each node marker, and is marked in periodic intervals along the left-most time line. These values represent the number of seconds since the start of the application.

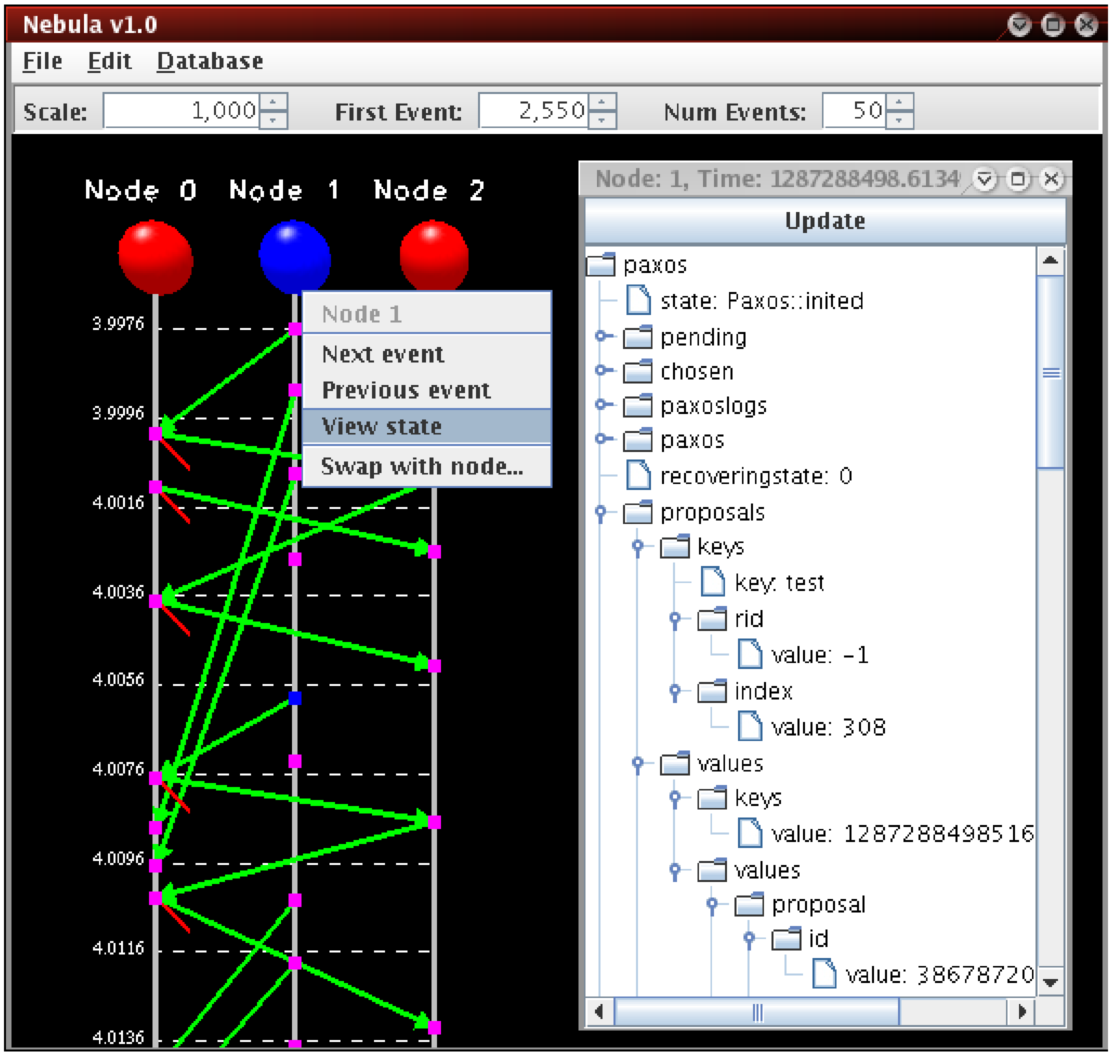
Each event, corresponding to an Event object in the state matrix, is represented as a magenta square. Complete messages are represented as green arrows pointing to the destination and always point in a downwards direction since time flows down in the graph. The short red diagonal lines represent incomplete messages, and simply indicate that a message was sent during an event but the corresponding receive event is not currently displayed on the graph.

The toolbar under the menu contains visualization parameters for controlling what portion of the event graph is displayed. In Figure 5.5, we see that the first event displayed is event number 118 and there are 50 total events displayed. The scale parameter controls the vertical spacing of the time axis of the event graph. This allows users to tune the spacing of the graph based on the timescale at which their application operates.

Finally, a message content popup is shown as the blue box, and appears whenever the user hovers the mouse over any complete or incomplete message arrow. The name of the message along with its contents is displayed and disappears when the user moves their mouse away from a message arrow.

### 5.3.2 Debug Window

The second part of the Nebula interface shown in Figure 5.5 is the debug window, used for traversing the event graph as well as displaying the contents of the currently selected event. The debug window has four components: (i) the event slider, (ii) the jump box, (iii) the event display pane, and (iv) a “Go” button.



```
paxos
  state: Paxos::inited
  pending
  chosen
  paxoslogs
  paxos
  recoveringstate: 0
  proposals
    keys
      key: test
      rid
        value: -1
      index
        value: 308
    values
      keys
        value: 128728849851 6
      values
        proposal
          id
            value: 38678720
```

**Figure 5.6**: Nebula State View

Users can jump to any event they wish by either adjusting the slider to the desired location or typing an event number into the jump box and clicking Go. Event objects displayed here are slightly different than those returned in NYQL queries. The top-level object is labelled by the node identifier, timestamp, and thread identifier the event took place on. Beneath that, we see all objects that were logged as part of the event. However, unlike Event objects in NYQL, these do not display function call parameters by default. In addition, each function call is also annotated with the time it took to complete. We also display return values of each function as children named “rv.”

### 5.3.3 State-view Window

Nebula’s third interface component is the *state-view window* and is shown in Figure 5.6. Right-clicking on one of the node markers in the event graph brings up a context menu with the option to view a node’s state. Here, Nebula extracts the contents of a user-specified object at the time of the currently selected event on the node that was clicked on.

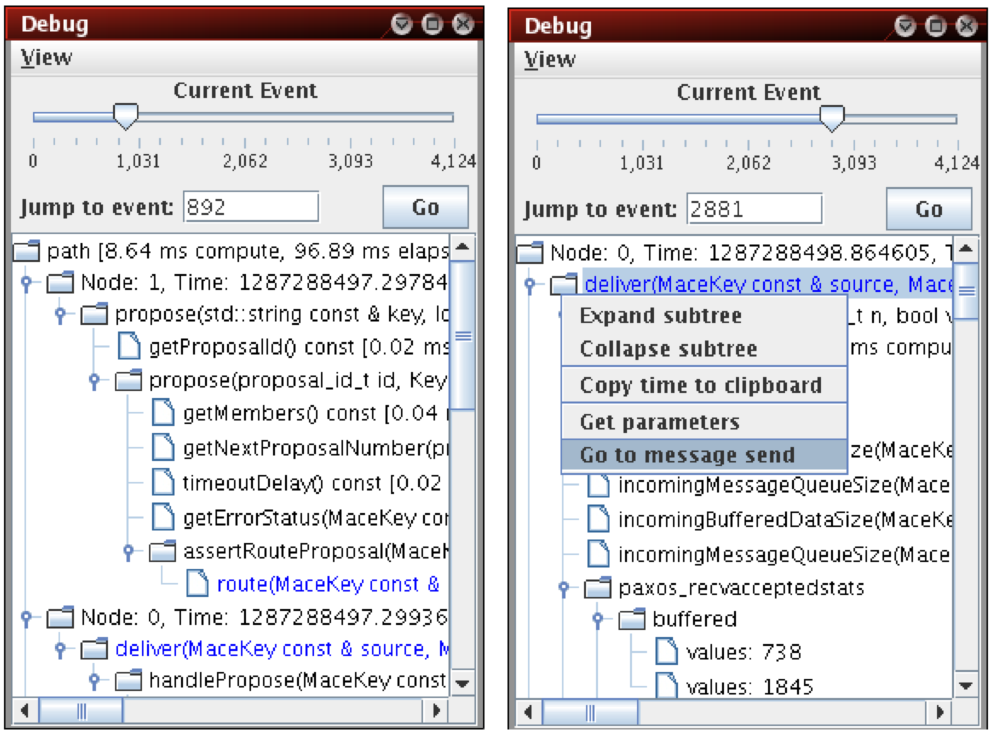
In Figure 5.6, we have configured Nebula to treat the “paxos” object as the state object. When we right-click on the Node 1 identifier and select “View state,” the paxos object logged by Node 1 most recently to the time of the currently selected event is extracted and presented. Once a state-view window is open, we can update the displayed state to the currently highlighted event by clicking the “Update” button. Alternatively, the user can open multiple state-view windows by right-clicking on a node identifier and choosing the “View state” option again.

### 5.3.4 Navigating the Event Graph

The combination of the event graph, the debug window, and the state-view window provide a powerful interface for navigating the event graph and viewing node state at various times. Here we describe a few additional methods of navigating the event graph.

Besides entering event numbers in the jump box or using the event slider, users can also use the “f” and “b” keys to step forward and back one event at a time. Press-ing “F” or “B” results in similar behavior, except that the events will be restricted to the currently selected node. Any time a user changes the currently selected event, the corresponding event is highlighted in the event graph. The node marker as well as the event marker are highlighted in blue for the currently selected event.

As shown in Figures 5.5 and 5.7, certain event components are displayed in blue text. These correspond to sending or receiving messages — in our examples, *deliver* or *route* function calls. If the user right-clicks on one of these entries, as shown in Figure 5.7(b), they are given the option to jump to the other endpoint of the message transmission. For example, if the user right-clicks on a route function call, they are given the option to jump to the event where the message is received. On the other hand, if the user right-clicks on a deliver function call, they have the option to jump to the event where the message was sent. Users can also fully expand or collapse any element in the tree or retrieve the parameters for any function call in an event.



```
path [8.64 ms compute, 96.89 ms elaps
  Node: 1, Time: 1287288497.29784
    propose(std::string const & key, Ic
      getProposalId() const [0.02 ms
      propose(proposal_id_t id, Key
        getMembers() const [0.04 r
        getNextProposalNumber(p
        timeoutDelay() const [0.02
        getErrorStatus(MaceKey co
        assertRouteProposal(MaceK
          route(MaceKey const &
  Node: 0, Time: 1287288497.29936
    deliver(MaceKey const & source, M
      handlePropose(MaceKey const
```

```
Node: 0, Time: 1287288498.864605, T
  deliver(MaceKey const & source, Mace
                                  _t n, bool v
                                  ms compu

                                  ze(MaceKe
    incomingMessageQueueSize(Mace
    incomingBufferedDataSize(MaceKe
    incomingMessageQueueSize(Mace
    paxos_recvacceptedstats
      buffered
        values: 738
        values: 1845
```

(a) Path View (b) Context Menu

**Figure 5.7**: Nebula Path View and Context Menu

The final piece of functionality provided by Nebula is the ability to extract the Path object around any event. This option is available by right-clicking on the top-level tree node of any event shown in the debug window. The results are shown in Figure 5.7(a). The top-level node is named "path" and contains two measures of time. The first, "compute," is the sum of all of the compute times of all events on the path. The second, "elapsed," is simply the time difference between the greatest ending timestamp of any event in the path and the timestamp of the beginning of the first event in the path. As a result, it is possible for the compute time to be larger than the elapsed time if the path involves significant parallel computation.

## 5.4 Implementing Nebula

Given our database backend and the implementation of the NYQL Queryer, implementing the event graph in Nebula was fairly straightforward. However, due to the different access patterns associated with message tracing and extracting a single Event object at a time, we did not use NYQL internally. Instead, we execute SQL queries directly to build the event graph, the Event objects displayed in the debug window, and the objects displayed in the state-view window. This way, we avoid the table creation process described in Section 3.3.4.

### 5.4.1 Building the Event Graph

To build the event graph, we must compute a few things. These include the number of nodes in the system and the number of Event objects that were logged. After Nebula has connected to a database, we issue the following two queries to count the number of nodes and Events, respectively:

```
(1) SELECT COUNT(DISTINCT node) FROM StartEvent
(2) SELECT COUNT(*) FROM StartEvent WHERE begin = 1
```

Next, we must build the visible portion of the event graph. Recall from Section 5.3.1 that this is controlled by two parameters, *StartEvent* and *NumEvents*. By default, Nebula starts with $StartEvent = 0$ and $NumEvents = 25$ so we will use these values below. First, we select everything from the StartEvent table corresponding to the first 25 events (Query 3). We use this information to create an event marker for each visible event. Next, we calculate the starting and ending *id* values in the Events table corresponding to the beginning of Event 0 and the end of Event 24 (Queries 4 and 5). We then select everything from the Events table between these two *id* values corresponding to message sends (Query 6).

```
(3) SELECT * FROM StartEvent WHERE id >= 0 AND id < 50
    AND begin = 1
(4) SELECT id AS startId FROM Events WHERE tname = 'SE'
    AND joinId = 0
(5) SELECT id AS endId FROM Events WHERE tname = 'SE'
    AND joinId = 49
(6) SELECT * FROM Events WHERE id >= startId AND id <= endId
    AND tname = 'MessageId'
```

For each message send we find, we next execute a query to extract the message identifier from the MessageId table (Query 7) and then find the Event *id* corresponding to where the message was received (Query 8). Keep in mind that the message identifiers of received messages are logged in the StartEvent table as the *mid* column.

```
(7) SELECT mid AS desired FROM MessageId WHERE id = FoundId
(8) SELECT id FROM StartEvent WHERE mid = desired
```

Once each of these queries returns, we can draw the visible portion of the event graph. When a user changes the *StartEvent* or *NumEvents* parameters, we simply re-construct the graph from scratch. We could optimize this process if any portions of the graph were to be redrawn, but rebuilding the whole thing has been fast enough that optimizing this step was not necessary.

### 5.4.2 Displaying Message Contents

To construct the contents of a message when the user hovers their mouse over a message arrow (as seen in Figure 5.5), we take advantage of knowledge of our logging system. That is, there is no specific log entry that only stores message contents. However, we do know that each deliver function takes a message as an argument, so we can extract the message contents from there.

Each message arrow we store keeps track of the receiving event id as well as the message id. The first thing we do is to extract everything from the Events table corresponding to the receiving event. This involves a number of queries similar to Queries 4–6 above, but with slight modifications. For Query 4, we set *joinId* equal to the *id* we store for the message arrow. In Query 5, we set the *joinId* to be the *id* used in Query 4 plus 1. Finally for Query 6, we omit the condition on *tname* so that we extract everything for the event and not just MessageId entries.

Next, we look for an entry in the returned data whose *tname* starts with "deliver." This should be one of the first few entries since the event should start with a deliver function call. Finally, since all of our deliver functions follow the same prototype pattern, we know that the message is the second parameter. We use the value of the *tname* and *joinId* columns to extract a single object from the correct table. This process potentially involves multiple queries, one for each of the tables the object is split over. However, each query is simple and is of the form:

```
(9) SELECT * FROM tableName WHERE id = joinId
```

We then combine the results from each of these queries to construct the string representation of the contents of the message that gets displayed on the event graph. This process contrasts with the NYQL method of reconstructing objects, where all the data is retrieved from a single table that has already joined each of the constituent tables together. Since we are only interested in a single object at a time, this method is sufficiently fast and is much simpler.

### 5.4.3 The Debug and State-view Windows

Constructing Events to display in the debug window involves a number of similar queries to the message construction process described above. Instead of extracting the contents of the Events table for a given StartEvent *id* and looking for a deliver function, we process every entry. We use entries where the *tname* is "SL" to keep track of stack depth, as we do in the Event object construction portion of Section 3.3.6. For each non-"SL" entry we encounter, we construct a single object using the above method (Query 9).

The contents of state-view window is also built using similar techniques. Since we know the *id* of the Event we are interested in, we extract everything from the Events table as if we were constructing the Event object for the debug window. However, this time we process the rows looking for a *tname* match against an object name the user has specified, representing the state of their application. We then use the single object reconstruction mechanism described above to build the instance of this object for the state-view window.

## 5.5 Debugging with Nebula

In this section, we report another graduate student's experiences using Nebula to debug their research on UNRP, the Unified Naming and Routing Protocol [72]. The goal of UNRP is to assign topologically meaningful host labels to hosts in a data center network which can be used in a variety of routing or forwarding contexts. Since UNRP is designed for data centers, it overlays a logical hierarchy on an input topology, which it assumes to be layered. Switches at the "bottom" layer are referred to as *edges* and are connected to hosts, while switches at the "top" are *core* switches. The rest of the switches are *aggregations*.

To assign labels, UNRP groups aggregation switches into *pods*, wherein the switches in each pod have the same set of neighboring edge switches. Each switch in a pod is assigned the same coordinate, which must be different from coordinates assigned to switches in all other pods. An example topology for this scenario can be seen in Figure 5.8. Here, we see cores $c1$ and $c2$, aggregations $a1$ – $a4$, edges $e1$ – $e4$, and hosts $h1$ – $h4$.

In each of the bugs below, the student used the MACE modelchecker (MACEMC) [50] to produce a buggy run. Buggy runs are typically terminated at the violation of a *property*, a logical condition over the state of all of the nodes in the system. The logs generated from each run are then analyzed with Nebula. Although we only describe two bugs, the student found and fixed around 10 bugs using Nebula during the development of UNRP.

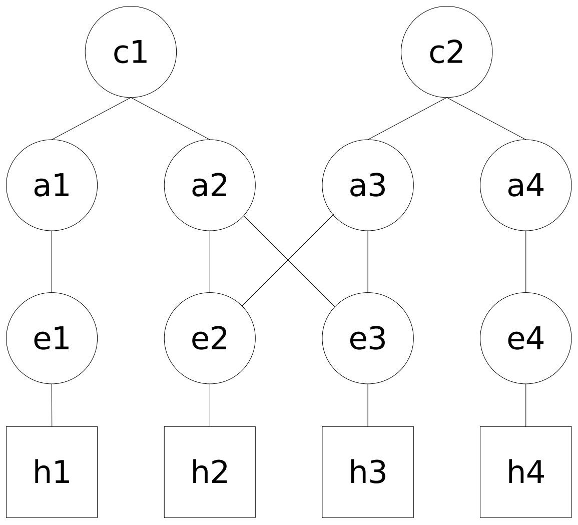


**Figure 5.8**: Sample UNRP Topology

### 5.5.1 UNRP Bug 1

The student noticed the first bug when a modelchecker run did not terminate with a property violation, but instead continued to run forever. After loading the logs into the database and connecting to it with Nebula, she jumped to the last event to see what caused it to loop. Referring to Figure 5.8, she noticed that the last event was $e2$ processing the delivery of a message containing *coordinate hints* from $a2$. These messages contain a list of pod coordinates that the edge switch is not allowed to choose for its neighboring pod. In this case, she inspected the values in this message and saw that they included the value -1. This value is not a valid pod number and should not have been included in the message, so the student jumped to the event where the message was sent.

From having implemented the protocol, the student knew that the contents of the coordinate hints message sent by aggregation switches are a union of all values sent to it in coordinate hints messages from their neighboring core switches. As a result, $c1$ must have sent $a2$ the -1. At this point, she simply looked back at the event graph for messages sent from $c1$ to $a2$ until she found the coordinate hints message. She could have written a query to find the event where the -1 was sent, but visual inspection was quicker in this case.

After inspecting the code for the sending of coordinate hints messages from core switches, the student discovered core switches simply send the contents of their local coordinate hints maps, which contain -1 entries for pods that are not yet assigned coordinates. When there are no pod coordinates available for choosing, an edge switch is supposed to choose -1 as this will indicate an error condition, but in this case -1 could not be chosen either because it was in the list of coordinates that were already taken. The simple solution was to exclude any -1 values from coordinate hints messages, which fixed the problem.

### 5.5.2 UNRP Bug 2

The second UNRP bug was discovered by a property that checks that all of the aggregations in the same pod share the same pod coordinate. Pods are assigned by UNRP such that nodes that are connected to the same set of edge switches belong to the same pod. Thus, in Figure 5.8, $a2$ and $a3$ should belong to the same pod.

The student first jumped to the last event in the run, since this is where the property failed. She then used Nebula's state view feature to view the state on $a2$ and $a3$ and noticed that they did have different pod coordinates, $X$ and $Y$, respectively. Next, she wanted to see where $a2$ received its pod coordinate, so she wrote the query shown in Figure 5.9. This query finds the first Event where node $a2$ has its *coordinate* state variable set to $X$.

```
foreach u in UNRP where(u.node == a2 && u.coordinate == X) {
  let e = Event[node == u.node, time @ u.time];
  output e order by u.time limit 1;
}
```

**Figure 5.9**: UNRP Pod Coordinate Query

Upon jumping to this event in the graph, the student noticed the event corresponded to $a2$ receiving an update message from $e3$, indicating it should set its coordinate to $X$. Curious why $e3$ would send $X$ to $a2$, she began looking backwards at the events on $e3$. Eventually, she discovered that $e3$ received a message from $a2$, indicating it had gotten disconnected from $e2$ and $e3$ should now be the one responsible for setting the coordinate. This would explain why $e3$ sent $X$ to $a2$. However, since $a3$ had a different coordinate than $X$, she figured something later on must be causing the discrepancy.

Next, the student returned to the event returned from the query in Figure 5.9 and began tracing forwards on $a2$, noticing that $a2$ receives a message from $e2$ (including a coordinate assignment) and therefore realizes that $e2$ is no longer disconnected. This message indicates that $Y$ (the correct value) is the coordinate. However, a few events later, the student sees that $a2$ announces $X$ as its pod coordinate to its neighbors, indicating that the value of $Y$ from $e2$ was ignored. At this point, she had found the bug. Once she inspected the code, she found that a cache entry was not cleared when $e2$ was disconnected, causing $a2$ to ignore $e2$'s request to set its coordinate to $Y$. She fixed the bug by changing the caching logic when a neighbor gets disconnected.

## 5.6 Summary

Debugging real systems is often a complicated process, involving tracing messages, finding events of interest, and looking at patterns. In this chapter, we have applied NYQL to debugging two systems, RANDTREE and PAXOS, showing that it can be a powerful tool to quickly focus on relevant areas of a distributed system's execution. The bugs we fixed are non-trivial — they involved viewing relevant parts of multiple nodes' state at different times, tracing messages, computing durations of different function calls, and finding instances of log messages matching certain criteria. Although nothing we have done would be impossible with traditional ad-hoc debugging techniques, a structured logging system combined with a well-defined query interface reduces the time to implement such queries by orders of magnitude while reducing the possibility of programmer error.

We also introduced Nebula, a graphical user interface integrating a navigable event graph, message and state inspection, and NYQL queries. We described its implementation on top of our database backend, and then described how it was used to debug a third distributed system, UNRP. Nebula not only makes it easy to visualize communication patterns, but also to traverse the events of the system by following messages or stepping through the execution of a single node. Both of these things are difficult and clumsy to do with queries alone, but were invaluable when debugging UNRP. This functionality would have been useful when debugging RANDTREE and PAXOS as well, which is what led us to pursue a tool that provides more than just a query interface.

Chapter 5, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

# Chapter 6

# Moving Away From A Compiler

In Chapter 4, we discussed a logging infrastructure for debugging with NYQL in the context of MACE, a source-to-source translator and set of libraries for building distributed systems. Although we have found that MACE greatly simplifies and speeds up the process of building distributed systems, the reality is that most distributed systems are not built around a source-to-source translator, let alone MACE in particular.

As a result, we wanted to evaluate the differences between compiler-generated and traditional hand-generated logging infrastructures, which are all that are available in most distributed systems. To make this comparison, we implemented two logging libraries, one in `C++` and one in `Java`, with the goal of emulating as much of the functionality of our MACE implementation as possible. We choose `C++` since it is the most prevalent language for distributed systems implementation, and `Java` because it is an interpreted language. Interpreted languages like `Java`, `Ruby`, and `Python` present a midpoint between statically-compiled languages without compiler support, and systems like MACE that provide full control of code generation.

In this chapter, we describe the similarities and differences between the major MACE logging features as implemented in our stand-alone `C++` and `Java` libraries. We show that there are significant advantages in using a translator or code generator in the toolchain, although the `Java` Virtual Machine has also been able to provide similar advantages. However, we begin by describing the `C++` logging library.

## 6.1 Logging in C++

The most notable difference between logging in MACE and logging with a C++ library is that with the library, users have to insert all logging statements themselves. Besides the obvious programming burden this introduces, there are a number of other difficulties involved when using a stand-alone library, which we describe below.

### 6.1.1 Structured Logging

As described in Section 4.3.1, MACE has mechanism for defining the composition of arbitrary log messages known as structured logging. In MACE, the user defines a structured log block in their code containing a set of function prototypes, each corresponding to a generated function that logs objects of the specified types.

Without a code generator, the structured logging functionality cannot be exactly replicated. Instead, we rely on a set of templatized logging functions, each designed to handle a different number of arguments. Figure 6.1 shows an example of structured logging in MACE, and its equivalent in our C++ library.

```
structured_logs {
  logMsg1(const vector<int>& c, int chosen);
  logString(const string& msg);
}

// code
vector<int> v = ...;
int ch = ...;
logMsg1(v, ch);
logString("String");
```

(a) MACE

```
DEFINE_LOG(logMsg1, vector<int> c,
                int chosen)
DEFINE_LOG(logString, string msg)

// code
vector<int> v = ...;
int ch = ...;
Log::logObjects("logMsg1", v, ch);
Log::logObjects("logString", "String");
```

(b) C++

**Figure 6.1**: Structured Logging in MACE and C++

The first difference we see is that the structured log block itself from MACE is replaced with a series of calls to the DEFINE_LOG macro. This macro takes the log name as the first parameter, followed by a variable number of arguments representing types and parameter names as in the MACE version and is responsible for setting up the table schema for this new object type, as well as logging the CREATE TABLE statement. As a result, the DEFINE_LOG macro must be called from the body of a user's code since it does not expand into its own function definition that must then be called separately.

Second, we can drop keywords like "const" and "&" since DEFINE_LOG does not expand into a function definition. In our first attempt with the library, DEFINE_LOG *did* expand to a function definition, although we eventually decided against it. We describe the reasons for this choice below in the Alternative Implementation section.

The third difference is the way objects are logged. In MACE, a function is defined for each prototype in the structured logs block, but in C++ this is not the case. Instead, we call a generic "logObjects" method, passing the object name we wish to log, followed by all of the parameters.

## Implications

The way structured logging is implemented in our C++ library results in a few unfortunate side effects that a translator-supported implementation does not suffer. The main cause of these issues is the fact that new methods are not generated for each structured log type defined.

Although C++ supports variadic functions, our implementation of Log::logObjects() does not use them since there is no way to tell how many arguments to process without the user specifying the number as a parameter, which we wanted to avoid. For more information on variable argument list processing, see [20]. Instead, our library defines seven versions of Log::logObjects(), each handling a number of parameters between zero and six. As a result, we can only support structured log definitions with up to six parameters. More could be supported, but the library would have to be modified.

The most significant drawback of our implementation is that there is no type checking of the arguments passed to Log::logObjects(). For instance, if a user wanted to log an object of type "logMsg1" as seen in Figure 6.1(b), they could pass the wrong number of arguments or arguments whose types did not match those declared in the corresponding DEFINE_LOG statement and the code would still compile. Since all of Log::logObjects() parameters are template types, there is no way to check whether the passed parameters are "correct." DEFINE_LOG, then, simply becomes a way for a programmer to express what types *should* be passed, although, it is up to them to make sure they adhere to these rules. If they do not, the error will not be caught until the programmer tries to populate the database, at which time the log will not match the schema of the created table.

## Alternative Implementation

As mentioned above, the deficiencies of our C++ structured logging implementation stem from the fact that the DEFINE_LOG macro does not expand into a function definition. Let us now consider an alternate scenario where this is not the case. First, consider a portion of the generated code for the "logMsg1" function used in Figure 6.1(a), shown in Figure 6.2.

```
void logMsg1(const vector<int>& c, int chosen) {
  ...
  // compute table indices for children objects
  uint32_t id = ...
  uint32_t c_id = ...
  uint32_t chosen_id = ...

  fprintf(sqlEventsLog, "%u\t%u\t%u\n", id, c_id, chosen_id);

  // log each child object
  logObj(&c, ...);
  logObj(&chosen, ...);
}
```

**Figure 6.2**: Generated Code for Structured Log "logMsg1" Shown in Figure 6.1(a)

We have omitted some unimportant details, but the relevant pieces are shown. For each parameter, we need to initialize a local variable with the table index that parameter will be logged at. Second, we need to print a line for the logMsg1 table, which contains the object's *id* and the *id*s of each child object. Finally, we need to call a logging function on each of the passed parameters.

It is possible to count the number of macro arguments [19], and even iterate over them using the Boost preprocessor library [2], but the main problem arises when we need to invoke the logging function on each parameter. Suppose we call DEFINE_TYPE(foo, map<string, vector<int> > param). Our macro has three arguments, delineated by commas — thus, the parameter we care about is actually "vector<int> > param." There is no way to extract just the "param" token in the preprocessor so that we can generate a call to log it. Furthermore, there is no way to tell that our second macro argument "map<string" does not contain a parameter name at all. Of course, we could pass the whole set of arguments to a C++ method to parse out the parameters, but obtaining the parameter names in C++ code does not allow us to refer to the variables with those names. As a result, there is no way to invoke called to logObj() on the correct arguments.

It may be possible to get the desired functionality with either multiple macros or a more convoluted syntax for declaring logging types, but this is not an avenue we have explored. Instead, our implementation has opted for speed and simplicity at the cost of type safety. However, it should be readily apparent that this tradeoff does not exist in MACE since we do not have to rely on the C++ preprocessor to generate code.

### 6.1.2 Function Call Parameters

The MACE compiler adds code to all functions defined in a user's source code, including class member functions. In our C++ implementation, users call a macro we provide to log a function's parameters. Its usage can be seen in Figure 6.3.

```
void someFunction(int x, double y, vector<int> z) {
  LOG_FUNC_CALL(someFunction, x, y, z);

  // function body below
  ...
}
```

**Figure 6.3**: Function Call Logging Example

The first LOG_FUNC_CALL parameter is the name of the object being logged. The rest of the parameters are simply the parameters that should be logged for this function call. In most cases, users should use the name of the function. However, unlike function call objects logged by MACE, this interface does not support naming objects as complete function prototypes. The object name given is used as both the *method* and *prototype* columns in the MethodMap table for each function. Therefore, if users have overloaded functions, they will have to manually give them different names. An alternative interface could allow users to define both the *method* and *prototype* columns separately, but this would complicate the common case which we wanted to avoid.

The LOG_FUNC_CALL macro uses the DEFINE_LOG_TYPE macro internally to set up metadata for the function call object the first time it is called. It also uses Log::logObjects() to log the parameters passed. As a result, it is subject to the same limitations as these methods. Specifically, this limits parameter logging to functions with six or fewer parameters in our current implementation. However, the user has the flexibility to choose whatever parameters they wish to log, so they can simply skip any uninteresting parameters. MACE, in contrast, automatically logs all parameters for each function. Users can control which parameters get logged only through a queries block as described in Section 4.4.

### 6.1.3 Return Values

In our C++ library, return value logging must also be done by hand. It is handled in exactly the same way as the generated MACE code shown in Figure 4.2(b). Since we use the same method as the generated MACE code, return value logging in C++ has no additional drawbacks besides having to be inserted by hand.

### 6.1.4 Events

Events are an important aspect of the NYQL debugging language. In MACE, Event objects are inserted into the generated code of all transitions and timer handlers. Their implementation is described in Section 4.2.2 and pseudocode is shown in Figure 4.3. Event logging is similar in our C++ library, although there are a few differences.

In general, we believe events should be defined the same way as they are in MACE. That is, functions that handle processing network messages, timer handlers, and any other "top-level" functions that initiate some sort of interesting functionality.

However, unlike MACE, we cannot make the assumption that our library is operating in a single-threaded environment. Our library implementation simply keeps track of the stack depth using thread-local storage, as opposed to a single static member of the Event class. This allows events to be recorded properly on multiple threads at the same time.

We provide a macro LOG_EVENT for logging function calls that should also have event logging. This macro behaves the same as LOG_FUNC_CALL, except it first instantiates an Event object.

### 6.1.5 Program State

One benefit of MACE is that the compiler knows what constitutes program state, since it is defined in the state_variables block in each MACE service. As a result, the MACE compiler can generate code to log the contents of all state variables every time an Event ends. In a general C++ library, this is not possible. However, it would be possible to register an object of interest with the Event class to be logged every time an event ended, but we have not implemented this functionality.

### 6.1.6 Controlling What Gets Logged

There are two classes of log customization MACE supports as described in Section 4.4. The first kind of customization is done at compile time based on the contents of a queries block in the user's source code. This customization can change the enabled logging in three ways by: (i) removing objects entirely from the log (Section 4.4.1), (ii) removing specific sub-objects from the log (Section 4.4.2), and (iii) conditionally logging objects based on clauses inserted into the generated code (Section 4.4.3). The second class of customizations, described in Sections 4.4.4 and 4.4.5, is done at run time. These customizations include dynamic binary rewriting and probabilistic path logging.

#### Compile-time Customization

Of the three compile-time customizations provided in MACE, none of them can be done at compile time in a C++ library. However, entire object elimination can be done at run time with a set of flags, one per object type, that the user can set in a configuration file. Before each object is logged, the code generated by the LOG_FUNC_CALL macro could check the appropriate flag, and only log the object if it is requested by the user.

However, this approach has two disadvantages to the query-based approach provided in MACE. First, a check has to be performed before each object is logged, potentially slowing down the logging subsystem. Second, if the library wanted to support setting flags with wildcards, as in "log all objects corresponding to deliver*," the testing code to determine whether an object should be logged would have to be more complicated, both in terms of runtime cost as well as code complexity. MACE's system avoids these complexities because it parses the user's source code and thus knows the entire set of objects that will exist when the system is run. As a result, it only has to take the performance hit at compile time to decide whether an object should be logged. The generated code is as efficient as possible as no checks must be done at run time.

Preventing specific sub-objects from appearing in the log, which MACE also supports at compile time, cannot fully be done in our C++ library. However, eliminating certain function call parameters can be done, since the user is responsible for listing each of the parameters that will be logged (see Figure 6.3). Unfortunately, this forces the user to modify the call site of each log statement they wish to change. In addition, the library cannot support eliminating sub-objects of other kinds of log statements, or sub-objects of function call parameters themselves. In MACE, the queries block is the single, central location for all logging modifications. It can also handle arbitrary sub-object removal, which is something not possible in a stand-alone C++ library.

The third compile-time customization that MACE supports — conditional logging via clause injection — is also not possible with a library-based approach. To get such functionality, the user would have to insert the desired conditions directly around the site of each log statement they wish to modify. As with the previous customization, this approach is clumsy at best since the library cannot offer any support for kind of logging control. Again, logging customization code ends up spread throughout a user's source code, instead of in one central place like in MACE.

**Runtime Customization**

The first runtime customization MACE provides is dynamic binary instrumentation. This technique allows users to change logging behavior at run time without requiring an application restart by rewriting functions that decide when each object type should be logged. This technique can be applied equally well to systems written with our logging library, so long as the code is structured in a similar fashion.

Recall from Section 4.4.4 that MACE generates a "shouldLog" function for every object type. These "shouldlLog" functions return true if the corresponding object type should be logged, and false otherwise. The only trick, then, for using dynamic binary rewriting with our library implementation is to generate similar "shouldLog" functions. Unfortunately, our DEFINE_LOG_TYPE macro must be executed *within* a code block, especially since they are called from the LOG_FUNC_CALL macro, so they cannot be used to define a "shouldLog" function. The only suitable solution we see is for the user to define and call the "shouldLog" functions themselves, which is tedious and error-prone.

The second runtime customization present in MACE is probabilistic path logging. Fortunately, from a usability standpoint this functionality is encapsulated almost entirely in our library as it is in MACE. The only difference is that before sending a message on the network, the user needs to query our library for the current value of the *ShouldLogPath* variable, which needs to be propagated in every message. However, this can be added to whatever network layer the application is using, so it should add almost no burden to the programmer. Like the Event class, we also had to store all state related to paths and path logging in thread-local storage to correctly handle multi-threaded applications.

## 6.2 Logging in `Java`

Our second logging library, implemented in `Java`, is able to overcome many of the deficiencies present in the `C++` library. In fact, due to the interpreted nature of the language, the compile-time code generation done in MACE can instead be done at run time. In this section, we give an overview of the features of our `Java` library and compare them to MACE logging as well as our C++ library.

### 6.2.1 Structured Logging

We start by describing our structured logging facility. Unlike C++, Java has no macros, so we could not implement structured logging this way. Fortunately, later versions of Java (1.6 and up) fully support *annotations* and customizable annotation processors. For a brief overview of Java annotations, see [14]. Figure 6.4 shows an example of our Java structured logging mechanism.

```
public class Foo {
  @StructuredLogs({
    "logType1(int x, Vector<String> y)",
    "logType2(String arg)",
    "logType3(HashMap<Integer, Integer> map)"
  })
  private static class FooLog extends FooLogGen { }

  public void someFunc() {
    int x = 12;
    Vector<String> v = ...

    // log an int and a Vector<String> as "logType1"
    FooLog.logType1(x, v);
  }
}
```

**Figure 6.4**: Structured Logging in Java

To use structured logging in a class, in this case “Foo,” the user must define a static inner class that is responsible for structured logging. The class can be public or private and given any name the user desires, but it must be declared as extending from *name*Gen. This inner class must then be annotated with the tag “@StructuredLogs.” This annotation takes one argument, a list of strings representing function prototypes the user wishes to call. This annotation is functionally equivalent to the structured_logs block in MACE. Then, anywhere in the class, the user can call functions with these prototypes as if they were defined as static methods of the inner class they defined.

In order for structured logging to work, the code must be compiled with the "-processor" flag set to the class file of our annotation processor, which must be compiled first. The processor gets a chance to inspect any classes annotated with "@Structured-Logs" and can generate new code that is compiled along with the rest of the user's code. In our case, the annotation processor defines the class "FooLogGen," adding a method for each of the elements in the @StructuredLogs list. Thus, when the user makes the call to "FooLog.logType1(x, v)," it has been defined in the base class FooLogGen and the code compiles correctly. Unfortunately, the annotation processor is not allowed to modify existing classes, which is why we cannot just add methods to FooLog directly. However, generating a non-existing base class is a common workaround to this problem.

Although the syntax is slightly messier than in MACE, `Java` structured logging shares all of the same advantages. Since we are able to generate functions matching the user's declarations, structured logging is type-safe, unlike our `C++` implementation. The `Java` version also does not have a limit on the number of parameters that can be passed. As long as the prototypes given by the user are valid `Java` prototypes, any number of parameters can be used.

### 6.2.2 Logging Objects

Logging collections of objects, or different object types in general, is more difficult in `Java` than `C++` due to the differences between `Java` generics and `C++` templates. For example, consider the `C++` and `Java` code fragments shown in Figure 6.5.

In `C++`, templates behave as we expect. When calling an overloaded function from inside a function with a templatized parameter, the correct function is dispatched. However, in `Java`, this is not the case. `Java` does not generate a copy of a templatized function for each different template parameter passed to it. As a result, in Figure 6.5(b), when we call logObj from logObject, the template type $T$ is "erased." The parameter $obj$ is thus seen as an instance of type "Object," and the logObj method that takes an Object as a parameter is always called. Unfortunately, this is not the desired behavior, so we could not implement our logging mechanism the same way as in `C++`. For more information on generics in `Java`, see [9].

Instead, we created an interface for all loggable objects called "LogObject." If

```
void logObj(const int* obj) {
  // log an integer
}

template<class T>
void logObj(const vector<T>* obj) {
  for (int i=0; i<obj−>size(); i++) {
    // calls the correct logObj
    logObj(&((*obj)[i]));
  }
}

void logObj(const void* obj) {
  // log unsupported type
}

template<class T>
void logObject(const T& obj) {
  . . .
  // dispatch to a specialized function
  logObj(&obj);
}

int main(int argc, char** argv) {
  vector<int> v = . . .;
  // calls logObj(int)
  logObject(4);
  // calls logObj(vector<T>)
  logObject(v);
  return 0;
}
```

(a) C++

```
public class Main {
  public static void logObj(int x) {
    // log an integer
  }

  public static <T> void logObj(Vector<T> obj) {
    for (T elt : obj) {
      // always calls logObj(Object)!
      logObj(elt);
    }
  }

  public static void logObj(Object obj) {
    // log an unsupported type
  }

  public static <T> void logObject(T obj) {
    . . .
    // dispatch to a specialized function
    logObj(obj);
  }

  public static void main(String[ ] args) {
    // calls logObj(Object)!
    logObject(5);
    // calls logObj(Object)!
    logObject(new Vector<String>());
  }
}
```

(b) Java

**Figure 6.5**: C++ Templates vs. Java Generics Example

a user wishes to log any object that is not a basic type, it must implement the LogObject interface. This means that all of the common Java collection types, like Vector, HashMap, HashSet, etc., are not loggable. However, we have provided simple extension classes which derive from their Java counterparts but provide the LogObject interface. Users must simply import our logging package instead of java.util.* to use these classes. Unfortunately, these classes must make use of the *instanceof* operator to check whether their template types are LogObjects to correctly log each of their elements. This adds a bit of overhead to these logging methods, but we believe it cannot be avoided.

### 6.2.3 Events

Like our C++ library, we keep track of stack depth on a per-thread basis using thread-local storage. However, since Java uses garbage collection for memory reclamation, classes do not have destructors. As a result, we cannot rely on the Event object destructor to decrement the stack count as we do in Figure 4.3. Instead, we define a method called "destruct" that must be called at the end of each method where an Event object is instantiated. Fortunately, the user does not have to insert this code themselves — it is automatically inserted at run time, a process we describe next.

### 6.2.4 Automatic Logging

With the Java Instrumentation API [13], we are able to modify classes are they are loaded by the Java Virtual Machine (JVM). Furthermore, we use Javassist [15] to perform source-level additions to the code. The combination of these two packages provides a powerful method of inserting logging code into classes of interest. In order to use our class transformer, users run the java command with the "-javaagent" option pointing to a jar file containing the compiled transformer class. As an argument, they pass a set of class names they are interested in instrumenting. Each time a class is loaded by the JVM, our transformer gets a callback containing the bytecode for the class. We then use Javassist to construct an in-memory representation of the class. If the class name matches one of the arguments given by the user, we modify the class to perform the appropriate logging.

For each method of a target class, we add code to log the parameters, add logging for the beginning and ending "SL" entries in the Events table, and add an entry for the MethodMap table. Unfortunately, compiled Java bytecode does not contain method parameter names by default, which means we have to refer to parameters as "arg1," "arg2," etc. However, if the user compiles their code with the "-g" flag, parameter names are recorded and we are able to use them as we do in MACE. As a result, function call objects can be named by their entire function prototype, unlike in our C++ version, which relies on users giving them unique names.

In addition, since there are no “typedefs” in `Java`, it is easy to tell if a parameter is a basic type and can be logged directly in its parent’s table. In `C++`, this process is much tricker and our `C++` library does not try to determine if arguments are basic types, since the checking would have to be done at run time and executed every time an object is logged. On the other hand, this testing is straightforward in `Java` and only needs to be done once when a class is loaded. The generated code will then always do the right thing with no runtime checking. This optimization allows us to reduce the number of tables for function call objects with parameters that are basic types.

In addition to function call logging for all methods, we also add Event logging to all *public* methods of a target class, since these methods are the ones that could be called from other classes and represent potential entry points into the logged code.

### 6.2.5 Extensions

So far, we have been encouraged by the power of our `Java` logging library. It very closely mimics the functionality provided by MACE, but without a stand-alone compiler. There are a number of features we have not implemented, but should be possible with our current infrastructure.

#### Return Values

To log method return values, we would generate a new wrapper method for each method whose return value we wish to log. The new method’s body would simply contain “return Log.logRv($existingFunc(\ldots)$).” This is the same approach MACE takes for logging return values.

#### Program State

Like `C++`, our `Java` logging library has no a priori way of knowing what should be logged as program state. It should be possible to register objects of interest with the logging subsystem that are logged at the end of every event, but we have not implemented such functionality.

#### Query-based Log Configuration

As described above, our current log configuration method uses a set of class names provided by the user that define what should be instrumented. However, all three types of query-based customizations should be possible. Instead of passing class names as an argument to our class transformer, the user would pass a set of queries. As each class is loaded, the transformer can generate the appropriate logging code for each class and method. Although this testing would be prohibitively expensive in `C++` since it would have to be done at run time for every call to a logging method, it only needs to be done once per class in `Java`. Furthermore, since we are able to generate new code, we can even insert conditional clauses which is not possible in our `C++` library.

All of this modification does happen at run time in `Java` though, so there will be some performance hit the first time each class is loaded, which the MACE implementation does not suffer. However, the `Java` system is more flexible since the code does not have to be recompiled to change the generated logging like it does in MACE, it only needs to be re-executed.

## 6.3 Summary

In this chapter, we presented two new logging libraries, implemented in `C++` and `Java`, as alternatives to our MACE-based approach described in Chapter 4. In both cases, our goal was to evaluate how features of the MACE logging subsystem translate to situations where no stand-alone compiler is present. The most notable differences between the three logging systems are summarized in Table 6.1.

Our `C++` library, while powerful in its own right, lacks in almost every area compared to our MACE approach. Structured logging is not type checked by the compiler and supports at most six arguments in our current implementation. Users must manually add logging statements to record function call parameters, return values, event boundaries, and program state if desired. The library could only barely support query-based configuration by checking whether each object should be logged at run time based on the objects specified in a query. Finally, our `C++` library does not attempt to determine whether arguments passed to structured logging methods or function calls are basic types

**Table 6.1**: MACE, `C++`, and `Java` Logging Comparison

| | | MACE | `C++` | `Java` |
|---|---|---|---|---|
| **Structured Logging** | **Type Checked?** | Yes | No | Yes |
| | **Unlimited Args?** | Yes | No | Yes |
| **Function Call Logging** | | Generated at Compile Time | User Added | Generated at Run Time |
| **Event Logging** | | Generated at Compile Time | User Added | Generated at Run Time |
| **Program State Logging** | | Generated at Compile Time | User Added | User Added |
| **Query-based Log Configuration** | | Full, at Compile Time | Basic Only, at Run Time | Full, at Run Time |
| **Basic Types in Parent Table** | | Educated Guess | Never | Always |

and can appear in their parents' tables, since the testing code would need to be executed every time a logging statement was made.

On the other hand, our `Java` library closely mimics the functionality present in the MACE logging system; the main difference is when logging statements are generated. Since the MACE compiler generates `C++` code, all of its code insertions happen at compile time. Similarly, query-based log configuration methods happen at compile time since they determine which logging statements should be generated. Our `Java` library, in contrast, does all of its logging code generation at run time, which allows it to be a bit more flexible than MACE. Furthermore, since `Java` has no typedefs, we can always tell when basic types are passed as arguments to structured logging methods and function calls. As a result, we generate the minimum number of tables possible, unlike our `C++` library which always generates a new table for each child variable, even if they are basic types.

Overall, we have found that using a code generator, either as a stand-alone part of the compile chain or integrated into an interpreted language, has many benefits that affect the overall flexibility and ease of use of a logging subsystem. Features such as automatic logging and query-based log customization are only possible in these scenarios.

Chapter 6, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

# Chapter 7

# Summary and Future Work

In this chapter, we provide a summary of the dissertation and give some directions for future research.

## 7.1 Summary

As websites grow in size, they increasingly rely on distributed systems for scalability. These systems run on potentially thousands of machines and implement various functionality from storage systems, to distributed computation services, to lock services, and others. Although distributed systems form the backbone of such websites, and are vital for their correct functioning, they are incredibly difficult to implement correctly.

In this dissertation, we have explored techniques for debugging distributed systems, with the goal of making debugging easier, faster, and less error-prone. Our key observation is based on the fact that debugging with ad-hoc scripts running over plain text log files has a number of drawbacks: (i) changing the log file format may cause previously-correct scripts to break, (ii) debugging often requires complex log processing scripts which become difficult to understand and maintain, and (iii) scripts can be inefficient since they may each need to process all of the log files in their entirety to complete.

In order to eliminate all three of these problems, we proposed a debugging methodology in which developers interact with a *model* of log file data, rather than directly with the data itself. We envision this model as a virtual three-dimensional cube, called the *state matrix*, organized by time, node identifier, and event type. Each element of this state matrix corresponds to a single log entry in a traditional log file, except represented as an *object* rather than a string with no inherent structure.

With this new model for log file data, developers can extract objects of interest using a high-level object-oriented query language called NYQL, developed with the express purpose of extracting objects from the state matrix. We showed that NYQL can be used to construct a wide range of useful queries, including many that would be quite difficult with traditional scripting languages.

We also described our implementation of NYQL on top of a relational database. Each object in the state matrix is split up across potentially many tables internally, and NYQL queries are translated to a set of SQL queries that are run against the database backend. Query results are then reconstructed into objects and presented to the user as trees in a graphical user interface.

Next, we covered the logging support needed for queryable objects. We presented a logging system in the context of MACE [49], a source-to-source translator and set of libraries for building distributed systems. We showed that with the aid of the translator, many kinds of logging can be automatically generated, including function call parameters and return values, event boundary markers, program state, and causal path information. We also introduced a new logging primitive called *structured logging*, which lets users define custom object-oriented logging methods.

As opposed to traditional log-level-based logging configuration, we showed how our logging system can support a new type of configuration based on queries. Users can control which objects get logged using a set of *target queries* given to the MACE translator. This type of log configuration supports dynamically generating conditional expressions into the generated source code, which cannot be done without the aid of a source-to-source translator.

In order to give some estimates as to how our query language performs, we benchmarked all aspects of the system. These benchmarks included the time to process and merge log files, the time to populate the database, and the time to execute a number of different queries. We also benchmarked an optimization that reduces query runtime if multiple queries are ever run over the same objects, which increases performance in the common case. Our final benchmarks covered the log customizations we introduced, and showed that they are indeed effective in controlling the logging overhead in a high-throughput distributed application.

Next, we evaluated NyQL as an actual debugging tool by applying it to two distributed systems written in Mace: a random tree-forming protocol and a high-performance implementation of Paxos, the classic distributed consensus algorithm. We were able to diagnose a handful of bugs in these two systems with our query interface alone, in much less time than it would have taken us without NyQL.

However, we found there were a number of situations where queries alone were clumsy to use, and a more visual representation would have been useful. In light of these observations, we developed Nebula, a new graphical user interface which integrates our query processor with an event graph representation. The event graph allows users to visualize communication patterns, which is something queries do not provide. The combination of the event graph *and* queries provides more functionality than either would alone. We demonstrated Nebula's use as a bug-fixing tool by having another student use it to debug their work-in-progress. We found Nebula was quite successful in this regard — the student was able to fix around 10 bugs with it.

The last chapter of this dissertation investigated more common distributed systems building environments — those without source-to-source translators. In these cases, we were unsure how a number of our logging features would work, since we explicitly took advantage of the translator to generate log statements for us. We investigated these scenarios by implementing two new logging libraries, one in `C++` and one in `Java`.

We found that our stand-alone `C++` library was able to mimic most of the functionality of our Mace implementation, although we were not able to implement query-based log configuration due to the reliance on the source-to-source translator. In addition, all of the logging statements that are generated by the Mace compiler have to be added by the user in the `C++` version. This alone makes the `C++` version much more difficult to use.

On the other hand, we were able implement a logging system that is nearly identical to the Mace logging system in `Java` due to its interpreted nature. While most features behave the same, the main difference is that `Java` does its code generation at run time, while MACE does all of its code generation at compile time. This difference has the potential to make the `Java` version slightly slower, but only when an application is first run and the classes are loaded. Although we did not implement the query-based log configuration for our `Java` library, we are confident all features of the MACE version are possible.

## 7.2 Future Work

Our experiences with query-based debugging have been promising. Our prototype system has proved to be efficient enough to debug many real-world bugs, and user-friendly enough to be used by others. However, query-based debugging is still in its infancy. There are a number of issues that need to be addressed before this approach would be ready for the large distributed systems that back today's large websites.

### 7.2.1 Issues of Scale

The most glaring limitation with this work is that our implementation has not been designed to scale well. Our system has shown that using queries for debugging is promising, but many real systems contain thousands of nodes. Although many such systems can be scaled down to manageable levels and still exhibit bugs, there will be occasions where this is not practical. When bugs can only be found at large scale, collecting logs in a central location, ordering them, and then inserting them all into a relational database is not a feasible solution.

We hypothesize that a publicly-available storage system like Hadoop's HBase [22], which provides an interface similar to BigTable [34], could replace the relational database of our implementation, allowing the system to scale to more nodes. However, since HBase does not support a relational query language, NYQL's entire implementation would need to be reworked, as would the representation of stored objects.

Research in this direction would also need to investigate the tradeoffs between online and offline logging. Should log statements be written directly to HBase, or should they still be logged to local disks and inserted into HBase afterwards? The former will incur more runtime overhead, but the latter will have a potentially large startup cost once a run is complete.

System scale would also affect the way the event graph is visualized. A small number of nodes are easily visible simultaneously, but thousands of nodes are not. The event graph would need more options for controlling which nodes are visible at any one time, perhaps aggregating multiple, similarly-functioning nodes together. Subsystems could also be treated as black boxes if they were known to function properly, allowing the event graph to ignore the messages and events executed by them.

### 7.2.2 Language Features

Although NYQL has been useful for us, it is hardly a mature language. It needs to be used in larger systems and refined. One possible language extension would be to support aggregation statements in "where" clauses. Currently, NYQL only supports them in "output" statements, but supporting them in "where" clauses would open up new kinds of queries. For example, we could write the query shown in Figure 7.1.

```
foreach b in ObjectType where (b.val1 < max(ObjectType.val1) / 2) {
  output b;
}
```

**Figure 7.1**: Query with "where" Clause Aggregate Functions

This query returns all instances of *ObjectType* whose *val*1 child is less than half the maximum value of any *ObjectType*'s *val*1. Currently, this query is not possible in NYQL.

Another language features that might prove useful would be an explicit iterator over the "node" dimension of the state matrix. Although NYQL supports slicing based on node, it does not provide a way to iterate over nodes directly. For example, we could write the query shown in Figure 7.2.

This query outputs the first instance of *ObjType* for each node in the system. Here, the ">@" would work like "@," except instead of choosing the first smallest timestamp, it chooses the first largest. Iterating over the "node" dimension could also be valuable in computing statistics for each node separately.

```
foreach n in nodes {
  let b = ObjType[node == n, time >@ $START];
  output b order by n;
}
```

**Figure 7.2**: Query With a Loop Over Nodes

### 7.2.3 Logging Customization

A third area of future work could be in the area of logging customization. One of the most difficult issues that log-based debugging faces is trying to decide whether each statement should be logged. Although we have covered a number of different ways to control what gets logged, there is always room for more advanced systems. For example, a developer may only want to see log message 2 if log message 1 is logged first, or log events on an error path once an error has been discovered.

Ideally, a system would log nothing except for the necessary state on the critical path to a bug. Currently, no one knows how to build a system like this. Of course, there is always a tradeoff between the power of the logging subsystem and its runtime cost. The “smarter” the system gets in choosing what to log at runtime, the slower it will run. Being able to tune this balance to an application’s needs would be an interesting line of future work, as well as developing adaptive logging systems in general.

## 7.3 Final Thoughts

Hopefully, this dissertation has made a convincing argument for query-based debugging. There are a number of challenges left that need to be addressed, but we believe it can become an invaluable tool. We hope the groundwork laid out in this dissertation can help move distributed systems debugging away from being a black art, and get closer to being a principled science.

Chapter 7, in part, is currently being prepared for submission for publication of the material. Braud, Ryan; Killian, Charles; Deutsch, Alin; Vahdat, Amin. The dissertation author was the primary investigator and author of this material.

# Bibliography

[1] Amazon Gets Downtime, No Explanations Yet. http://www.nwinnovation.com/amazon_gets_downtime_no_explanations_yet/s-0029570.html.

[2] Boost.Preprocessor. http://www.boost.org/doc/libs/release/libs/preprocessor.

[3] DB2 pureXML - Intelligent XML database management. http://www-01.ibm.com/software/data/db2/xml.

[4] DDD - Data Display Debugger. http://www.gnu.org/software/ddd.

[5] Debugging in Visual Studio. http://msdn.microsoft.com/en-us/library/sc65sadd.aspx.

[6] Eclipse - The Eclipse Foundation open source community website. http://www.eclipse.org.

[7] Facebook Downtime Means Real-Life Repercussions for Blogosphere. http://blogs.forbes.com/velocity/2010/09/24/facebook-downtime-means-real-life-repercussions-for-blogosphere.

[8] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb.

[9] Generics in the Java Programming Language. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[10] Getting to XML. http://www.oracle.com/technetwork/issue-archive/2005/05-may/o35xml-094515.html.

[11] Gmail Outage Marks Sixth Downtime in Eight Months. http://www.pcworld.com/article/160153/gmail_outage_marks_sixth_downtime_in_eight_months.html.

[12] gnuplot homepage. http://www.gnuplot.info.

[13] Instrumentation (Java Platform SE 6). http://download.oracle.com/javase/6/docs/api/java/lang/instrument/Instrumentation.html.

[14] Java Annotations. http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html.

[15] Javassist. http://www.javassist.org.

[16] nm(1) - Linux man page. http://linux.die.net/man/1/nm.

[17] The ISC Domain Survey|Internet Systems Consortium. http://www.isc.org/solutions/survey.

[18] unittest - Unit testing framework - Python v2.7 documentation. http://docs.python.org/library/unittest.html.

[19] Variadic macro to count number of arguments. http://cplusplus.co.il/2010/07/17/variadic-macro-to-count-number-of-arguments.

[20] va_start(3) - Linux man page. http://linux.die.net/man/3/va_start.

[21] Welcome to Apache Hadoop! http://hadoop.apache.org.

[22] Welcome to HBase! http://hbase.apache.org.

[23] Welcome to JUnit.org! http://www.junit.org.

[24] White Paper: What's New for XML in SQL Server 2008. http://www.microsoft.com/sqlserver/2008/en/us/wp-sql-2008-whats-new-xml.aspx.

[25] AGANS, D. J. Debugging. Amacom, 2006.

[26] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance Debugging for Distributed Systems of Black Boxes. In SOSP (2003).

[27] ALTEKAR, G., AND STOICA, I. ODR: Output-Deterministic Replay for Multicore Debugging. In SOSP (2009).

[28] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a Needle in Haystack: Facebook's photo storage. In OSDI (2010).

[29] BECK, K., AND ANDRES, C. Extreme Programming Explained, second ed. Addison-Wesley Professional, 2004.

[30] BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, January 2007.

[31] BRAUN, P. Parallel Program Debugging Using Scalable Visualization. ICAAP 2 (1995), 699–708.

[32] BURROWS, M. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In OSDI (2006).

[33] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic Instrumentation of Production Systems. In USENIX (2004).

[34] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In OSDI (2006).

[35] CHENG, Y.-P., CHEN, J.-F., CHIU, M.-C., LAI, N.-W., AND TSENG, C.-C. xDIVA: A Debugging Visualization System with Composable Visualization Metaphors. In OOPSLA Companion (2008).

[36] CLUET, S. Designing OQL: Allowing objects to be queried. Information Systems 23 (1998), 279–305.

[37] CONRADIE, L., AND MOUNTZIA, M.-A. A Relational Model for Distributed Systems Monitoring using Flexible Agents. In SDNE (1996).

[38] CONSENS, M. P., HASAN, M. Z., AND MENDELZON, A. O. Using Hy+ for Network Management and Distributed Debugging. Proceedings of Centre for Advanced Studies on Collaborative Research: Software Engineering 1 (1993), 450–471.

[39] DAO, D., ALBRECHT, J., KILLIAN, C., AND VAHDAT, A. Live Debugging of Distributed Systems. In CC (2009).

[40] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In OSDI (2004).

[41] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon’s Highly Available Key-Value Store. In SOSP (2007).

[42] DEUTSCH, A., FERNÁNDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. XML-QL: A Query Language for XML. http://www.w3.org/TR/NOTE-xml-ql, August 1998.

[43] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-Trace: A Pervasive Network Tracing Framework. In NSDI (2007).

[44] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global Comprehension for Distributed Replay. In USENIX (2007).

[45] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay Debugging for Distributed Applications. In USENIX (2006).

[46] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In SOSP (2003).

[47] GODEFROID, P. Model Checking for Programming Languages using VeriSoft. In POPL (1997).

[48] HOLZMANN, G. J. The Model Checker SPIN. IEEE Transactions on Software Engineering 23, 5 (1997), 279–295.

[49] KILLIAN, C., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. Mace: Language Support for Building Distributed Systems. In PLDI (June 2007).

[50] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, Death, and the Critical Transition: Detecting Liveness Bugs in Systems Code. In NSDI (2007).

[51] KILLIAN, C., NAGARAJ, K., PERVEZ, S., BRAUD, R., ANDERSON, J. W., AND JHALA, R. Finding Latent Performance Bugs in Systems Implementations. In FSE (2010).

[52] KRANZLMÜLLER, D., GRABNER, S., AND VOLKERT, J. Message Passing Visualization with ATEMPT. In PARCO (1995).

[53] KULKARNI, D., BOLOGNESE, L., WARREN, M., HEJLSBERG, A., AND GEORGE, K. LINQ to SQL: .NET Language-Integrated Query for Relational Data. http://msdn.microsoft.com/en-us/library/bb425822.aspx, March 2007.

[54] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21, 7 (1978), 558–565.

[55] LAMPORT, L. The Part-Time Parliament. ACM TOCS 16, 2 (1998), 133–169.

[56] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. D3S: Debugging Deployed Distributed Systems. In NSDI (2008).

[57] LIU, X., LIN, W., PAN, A., AND ZHANG, Z. WiDS Checker: Combating Bugs in Distributed Systems. In NSDI (2007).

[58] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing Declarative Overlays. In SOSP (2005).

[59] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Binary Instrumentation. In PLDI (2005).

[60] MCCONNELL, S. Code Complete. Microsoft Press, 1993.

[61] MUSUVATHI, M., AND QADEER, S. Fair Stateless Model Checking. In PLDI (2008).

[62] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and Reproducing Heisenbugs in Concurrent Programs. In OSDI (2008).

[63] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A Not-So-Foreign Language for Data Processing. In ACM SIGMOD (2008).

[64] PARK, S., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., LU, S., AND ZHOU, Y. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In SOSP (2009).

[65] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the Data: Parallel Analysis with Sawzall. In Scientific Programming Journal (2005), pp. 227–298.

[66] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the Unexpected in Distributed Systems. In NSDI (2006).

[67] SHANMUGASUNDARAM, J., TUFTE, K., HE, G., ZHANG, C., DEWITT, D., AND NAUGHTON, J. Relational Databases for Querying XML Documents: Limitations and Opportunities. In VLDB (1999).

[68] SINGH, A., MANIATIS, P., ROSCOE, T., AND DRUSCHEL, P. Using Queries for Distributed Monitoring and Forensics. In EuroSys (2006).

[69] SNODGRASS, R. A Relational Approach to Monitoring Complex Systems. ACM TOCS 6, 2 (1988), 157–195.

[70] VERBOWSKI, C., KICIMAN, E., KUMAR, A., DANIELS, B., LU, S., LEE, J., WANG, Y.-M., AND ROUSSEV, R. Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management. In OSDI (2006).

[71] VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. Model Checking Programs. In ASE (2000).

[72] WALRAED-SULLIVAN, M., MYSORE, R. N., TEWARI, M., MIRI, P., MARZULLO, K., AND VAHDAT, A. Automated, Scalable, Decentralized Naming for the Data Center. In submission to NSDI, 2010.

[73] WOLFSON, O., SENGUPTA, S., AND YEMINI, Y. Managing Communication Networks by Monitoring Databases. IEEE Transactions on Software Engineering 17, 9 (1991), 944–953.

[74] YABANDEH, M., KNEŽEVIĆ, N., KOSTIĆ, D., AND KUNCAK, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In NSDI (2009).

[75] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Úlfar Erlingsson, Gunda, P. K., and Currey, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In OSDI (2008).

[76] Zeller, A. Why Programs Fail: A Guide to Systematic Debugging, second ed. Morgan Kaufmann, 2009.