FAILURE DIAGNOSIS IN DISTRIBUTED SYSTEMS

BY

EUN SOO SEO

DISSEYATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Tarek Abdelzaher, Chair
Professor Jiawei Han
Professor Nitin Vaidya
Assistant Professor Steven Y. Ko, University at Buffalo,
The State University of New York
Failures in computing systems are unavoidable. Therefore, it is important to detect and diagnose failures early to improve the reliability of systems. In this dissertation, new approaches on root-cause diagnosis for two notorious types of failures in distributed systems are introduced.

This dissertation first focuses on the failures that are caused by software bugs triggered by race conditions. Due to the non-deterministic manifestation, these bugs are much harder to diagnose, fix and test than the bugs in sequential logic. To understand the concurrency bugs, we first study the characteristics of concurrency bugs using 105 bugs of four representative open-source programs. Motivated by the interesting findings from the study, we also propose an automatic bug diagnosis tool for distributed programs that finds the minimal causal orders of related events that trigger the bugs. Our tool is a significant extension to the previous tools that can find only bug-triggering sequence of events.

The second focus of this dissertation is on the failures that are caused by propagating errors. An error started by a single network component propagates and contaminates other components. As a result, a large number of network components are infected by errors. To fix the problem, root-cause of this problem, the single component that started the error propagation, needs to be identified. It is assumed that only a limited view on the status of components — whether they are infected or not — are available through monitors, a set of pre-selected network components. For this problem, we propose two root-cause diagnosis tools. The first tool relies on a simple intuition that the root-cause component is likely to be close to the infected monitors and far from the uninfected monitors. We also compare six
different monitor selection methods. The second tool makes use of additional information — failure propagation probability and time of infections — to improve the accuracy of root-cause diagnosis. We propose approximation algorithms to calculate the likelihood that a node is the failure source. In addition, we also propose a new monitor selection algorithm that maximizes the number of infected monitors for best accuracy of root-cause diagnosis.
For Sunkyung and Jooha.
Acknowledgments

“If you have raced with men on foot and they have worn you out, how can you compete with horses? If you stumble in safe country, how will you manage in the thickets by the Jordan?”

This thesis would have been impossible without the support of several individuals. First of all, I would like to express my deep gratefulness to my advisor, Prof. Tarek Abdelzaher. For the last three years, he has been a wonderful advisor to me. Discussion with him was always inspiring, encouraging and optimistic. I am also grateful to Prof. Jiawei Han, Prof. Nitin Vaidya and Prof. Steven Ko for serving on my thesis committee and giving me invaluable comments. I also thank Prof. Jennifer Hou. In her last visit to UIUC before she passed away, she came to me, asked about my new advisor, and gave me encouragement while she was dying of cancer. She was such a caring person. During my study at UIUC, I have worked with exceptionally many advisors and would like to thank my previous advisors — Prof. Haiyun Luo, Prof. Yuanyuan Zhou and Prof. Matt Caesar — as well. I have learned a lot from them.

I have received a lot of inputs and support from many friends and colleagues including Chun-cheng Chen, Shan Lu, Soyeon Park, Kyungtae Kang and MuQun Kent Yang (at the HDF Group). I also would like to thank my friends at the Korean Church of Champaign–Urbana including Seung Hoon Sung, Yuki Kang, Young-Im Cho and Hyun Eun Choi. Whenever I visited Korea, many longtime friends have always encouraged me. I would like to express my thank to Jihoon Kang, Jinah Lee, Hyejin Hwang, Jee Yong Jung, Jinsung

1Jeremiah 12:5, NIV
Kim and Wonyoung Maeng.

Korea Foundation for Advanced Studies (KFAS) supported my study. I am grateful to all people there.

I would like to thank my family for their love and support. Especially, I thank my parents for everything they have done for me.

Finally, I would like to express my utmost love and gratitude to my wife, Sunkyung Oh and my seven-month-old daughter, Jooha Seo. Without Sunkyung’s love and encouragement, I could never complete my Ph.D. Jooha also gave me a lot of energy when I was exhausted. This dissertation is dedicated to them.
# Table of Contents

List of Tables ........................................................................ x

List of Figures ........................................................................ xi

Chapter 1 Introduction .............................................................. 1
  1.1 Failures Triggered by Race Conditions ................................. 1
  1.2 Failures Caused by Propagating Errors ................................. 3
  1.3 Contributions .................................................................. 3
  1.4 Outline ........................................................................ 5

Chapter 2 Concurrency Bug Characteristic Study ......................... 6
  2.1 Motivation ...................................................................... 6
  2.2 Methodology ................................................................... 10
    2.2.1 Bug Sources ............................................................ 10
    2.2.2 Characteristic Categories .......................................... 11
    2.2.3 Threats to Validity ................................................... 12
  2.3 Bug Pattern Study ............................................................ 13
  2.4 Bug Manifestation Study .................................................... 20
    2.4.1 How many threads are involved? .................................. 20
    2.4.2 How many variables are involved? .............................. 22
    2.4.3 How many accesses are involved? .............................. 25
  2.5 Bug Fix Study .................................................................. 27
    2.5.1 Fix Strategies ........................................................... 27
    2.5.2 Mistakes during bug fixing ........................................ 31
    2.5.3 Discussion: bug avoidance ........................................ 31
  2.6 Other Characteristics ........................................................ 35
  2.7 Related Work .................................................................. 36
  2.8 Chapter Summary ............................................................. 37

Chapter 3 Finding Complex Bug-Triggering Conditions .................. 38
  3.1 Overview ....................................................................... 38
  3.2 Definitions ..................................................................... 39
  3.3 Finding a bug-triggering event pattern ................................. 41
    3.3.1 How likely is a given pattern a bug-trigger? .................. 41
    3.3.2 Pattern Score .......................................................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3</td>
<td>Mining Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Bounding</td>
<td>46</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Branching</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Case Studies</td>
<td>50</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Virtual Cord Protocol (VCP)</td>
<td>50</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Chord</td>
<td>54</td>
</tr>
<tr>
<td>3.4.3</td>
<td>GreenGPS</td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Performance Evaluation</td>
<td>57</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Synthetic Graph Data</td>
<td>57</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Synthetic Sequence Data</td>
<td>59</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Experiment 1 – Varying Number of Execution Graphs</td>
<td>59</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Experiment 2 – Varying Number of Mixes in Execution Graphs</td>
<td>62</td>
</tr>
<tr>
<td>3.6</td>
<td>Related Work</td>
<td>64</td>
</tr>
<tr>
<td>3.7</td>
<td>Chapter Summary</td>
<td>66</td>
</tr>
</tbody>
</table>

Chapter 4  Root-Cause Identification in Error-Propagating Networks: Exploiting Component Graphs

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Error-Propagating Networks</td>
<td>67</td>
</tr>
<tr>
<td>4.2</td>
<td>Root-Cause Identification</td>
<td>68</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Method</td>
<td>69</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Monitor Selection</td>
<td>70</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Case Study</td>
<td>72</td>
</tr>
<tr>
<td>4.3</td>
<td>Estimating the Number of Error-Initiating Nodes</td>
<td>76</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Greedy Sources Set Size (GSSS)</td>
<td>76</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Maximal Distance of Greedy Information Propagation (MDGIP)</td>
<td>77</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Case Study</td>
<td>78</td>
</tr>
<tr>
<td>4.4</td>
<td>Related Work</td>
<td>84</td>
</tr>
<tr>
<td>4.5</td>
<td>Chapter Summary</td>
<td>87</td>
</tr>
</tbody>
</table>

Chapter 5  Improving Root-Cause Identification in Error-Propagating Networks: Leveraging Additional Information

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Leveraging Propagation Probability</td>
<td>88</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Definitions</td>
<td>88</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Problems</td>
<td>89</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Approximation 1: End-to-end Propagation Probability</td>
<td>91</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Approximation 2: Outcome Probability</td>
<td>92</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Finding the Error Source</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>Leveraging Infection Timestamps</td>
<td>95</td>
</tr>
<tr>
<td>5.3</td>
<td>Selecting Monitors</td>
<td>96</td>
</tr>
<tr>
<td>5.4</td>
<td>Evaluation</td>
<td>99</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Datasets</td>
<td>99</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Leveraging Propagation Probability</td>
<td>100</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Leveraging Infection Timestamps</td>
<td>104</td>
</tr>
<tr>
<td>5.5</td>
<td>Chapter Summary</td>
<td>105</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Our findings of real world concurrency bug characteristic and their implications for concurrency bug detection, concurrent program testing and concurrent programming language design. (*: All terms and categories mentioned here are explained in Section 2.2.) ............................................. 14
2.2 Applications and Bugs Used in the Bug Study ............................................. 15
2.3 Our characteristic categories and definitions. (*: The bug pattern category is determined by the root cause of a concurrency bug, i.e. what type of programmers’ synchronization intention is violated, regardless of possible bug fix strategies.) ............................................. 16
2.4 Patterns of non-deadlock concurrency bugs. (There are 3 examined bugs, whose patterns can be considered as either atomicity or order violation. Therefore, they are considered in both categories.) ............................................. 17
2.5 The number of threads (or environments) involved in concurrency bugs. .............. 22
2.6 The number of variables (resources) involved in concurrency bugs. ................. 23
2.7 The number of accesses (or resource acquisition/release) involved in concurrency bugs. (*: “1 acc.” case happens only in deadlock bugs, when one thread waits for itself. The bug triggering therefore does not depend on any inter-thread order problem.) ............................................. 25
2.8 Fix strategies for non-deadlock concurrency bugs (all categories are explained in Table 2.3). .................................................. 27
2.9 Fix strategies for deadlock bugs (all categories are explained in Table 2.3) ............................................. 30
2.10 Can TM help avoid concurrency bugs? (*: There are some concerns) ............. 33
3.1 Number of good/bad execution graphs with zero/non-zero instances ................. 42
3.2 Parameters for the synthetic data generation ............................................. 60
4.1 Classification of rumors and non-rumors (Number of Monitors: 640). .............. 83
5.1 Infected Monitors and Uninfected Monitors ............................................. 96
5.2 Used Datasets ............................................. 97
List of Figures

2.1 An atomicity violation bug from MySQL. ........................................... 15
2.2 An order violation bug from Mozilla. The program fails to enforce the programmers’ order intention: thread 2 should not read mThread until thread 1 initializes mThread. Note that, this bug could be fixed by making PR_CreateThread atomic with the write to mThread. However, our bug pattern categorization is based on the root cause, regardless of possible fix strategies. .................................................. 17
2.3 A MySQL bug that is neither an atomicity-violation bug nor an order-violation bug. The monitor thread is designed to detect deadlock. It restarts the server when a thread i has waited for a lock for more than fatal_timeout amount of time. In this bug, programmers under-estimate the workload (n could be very large), and therefore the lock waiting time would frequently exceed fatal_timeout and crash the server. .................................................. 18
2.4 A write-write order violation bug from Mozilla. The program fails to enforce the programmers’ order intention: thread 2 is expected to write io_pending to be FALSE some time after thread 1 initializes it to be TRUE. Note that, this bug could be fixed by making S1 and S2 atomic. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies. .................. 18
2.5 A Mozilla bug that violates the intended order between two groups of operations. 19
2.6 A multi-variable concurrency bug from Mozilla. Accesses to three correlated variables, mContent, mOffset and mLength, should be synchronized. ........... 24
2.7 A MySQL bug that cannot be fixed by adding/changing locks. ..................... 28
2.8 A MySQL bug Fix .......................... 28
2.9 The process of fixing the bug shown in Figure 2.5. Programmers want to make sure js_MarkAtom will not be called after js_UnpinPinnedAtom. They first added a flag variable state to fix the bug. However, that left a small window between S1 and S2 unprotected. They finally added a second flag variable gcLevel to completely fix the bug. .................. 32

3.1 Definitions .................................................. 40
3.2 Pattern Score .................................................. 44
3.3 Methods for Prepending Events .................................................. 47
3.4 Equivalent Patterns .................................................. 47
3.5 Last-event Tree .................................................. 48
3.6 Duplicate Search .................................................. 49
5.9 These three figures show the average rank of the actual error source with different ratio of monitors in dataset 4. In all simulations, the ratio of infected nodes in the network is between 0.02 and 0.03 and the average values are calculated over 100 simulations.
Chapter 1

Introduction

Humans are imperfect, so do computers and programs. Due to various reasons such as software problems (software bugs, software upgrade failures, security vulnerabilities, configuration errors, etc.) and hardware problems (power failures, manufacturing defects, device wear-out, etc.), failures in computing systems are unavoidable. Therefore, it is essential to detect and diagnose such failures quickly and accurately to improve reliability of systems.

This dissertation introduces new approaches to root-cause diagnosis for two types of notorious failures in computer systems. First, we study failures that are caused by software bugs related to complex race conditions. Due to their non-deterministic manifestation, it is especially hard to find out what the exact bug-triggering conditions are. Second, we study failures that are caused by propagating errors. From the large number of infected components, it is hard to find the original source of the errors, which is the root-cause of the problem, without knowing details about how the errors are propagated.

1.1 Failures Triggered by Race Conditions

With the advances in the networking technologies, distributed programs are becoming more popular and important. From simple client-server architecture, distributed programs have evolved to more complex forms such as peer-to-peer systems, cluster computing, cloud computing, etc. While the need for distributed programs has been increased drastically, writing correct distributed program is still a challenge.

Distributed computing systems are composed of many processes, each of which run a
part of a distributed program and interacts with other processes. In contrast to centralized sequential programs, which run on a single process, distributed programs use networking for message exchange between software components spanning multiple computers. Message delivery order is essentially non-deterministic because of many reasons such as MAC-layer contention, routing delay, packet drops, and TCP retransmissions. Bugs related to the message delivery orders may be triggered or not depending on the outcomes of message race conditions.

Non-deterministic bugs induced by message race conditions are a real challenge in program development. In the traditional bug fix scenario, user first reports the symptom of a bug and the used input. Programmer then uses the same input to reproduce the bug. To diagnose the bug, he may use interactive debuggers, set breakpoint around the part where bug manifests, examine the execution using single-stepping.

Unfortunately, these techniques cannot be used to debug message race bugs in distributed programs. Due to the non-deterministic message delivery of the network, bug reproduction is not guaranteed with the same input. In bug reports of popular open source software, it is easy to find cases that developers have difficulty in reproducing bugs that can be easily observed in the reporters’ sites. Making things even worse, traditional debuggers such as gdb cannot be used to debug distributed programs so breakpoints or single-stepping cannot be used for bug diagnosis.

Testing a distributed program is also challenging. Even if a program passes comprehensive tests many times, it is still early to declare that the program is bug-free since the space of all possible message orders is huge and almost impossible to test it entirely. Therefore, many bugs still remain after in-house testing and they are eventually exposed to the users.
1.2 Failures Caused by Propagating Errors

Modern computing systems are composed of a large number of software/hardware components and their complex interconnections. While the links between components enable communication, they also spread errors.

For example, let’s consider a system with many software components. Result of computation from one component can be fed to other components as inputs. If a component has a software bug, it may produce bad results (initial error). Since the bad results are fed to other components, they also can produce bad results (error propagation). As a result, the initial error introduced by a single buggy component propagates to many other components. To fix this problem, the single buggy root-cause component needs to be identified.

Another example of error propagation is computer virus and Internet worms. Initially, one computer is infected by a computer virus. As it contacts other computers that have security vulnerabilities, the virus may propagate to them as well. After many computers get infected, it is not easy to find the initial computer that started the virus propagation.

1.3 Contributions

Failures Triggered by Race Conditions

(1) Concurrency Bug Characteristic Study

This work provides the first (to the best of our knowledge) comprehensive real world concurrency bug characteristic study. Specifically, we examine the bug patterns, manifestations, fix strategies and other characteristics of real world concurrency bugs. Our study is based on 105 randomly selected real world concurrency bugs, including 74 non-deadlock bugs and 31 deadlock bugs, collected from 4 large and mature open-source applications: MySQL, Apache, Mozilla and OpenOffice, representing both server and client applications. For each bug, we carefully examine its bug report, corresponding source code, related patches, and
programmers discussion, all of which together provide us a relatively thorough understanding of the bug patterns, manifestation conditions, fix strategies and diagnosis processes. Our study reveals many interesting findings, which provide useful guidelines for concurrency bug detection, concurrent program testing, and concurrent programming language design.

(2) Finding Complex Bug-Triggering Conditions

In this work, we propose a diagnosis tool for non-deterministic bugs in distributed systems. It uses execution traces of a distributed program in the form of directed acyclic graphs (DAGs) of events with causal order. With a number of traces from successful executions and bug-triggered executions, the tool finds the minimal pattern of events that is highly related to bug-manifestation.

Failures Cause by Error Propagation

We also propose two tools that diagnose root-cause component in error-propagating networks.

(3) Finding the Root-Cause by Exploiting Component Graph

Our first root-cause diagnosis tool for error-propagating networks is based on a simple intuition that the root-cause component is close to the infected components and far from the uninfected components. Using this intuition, we propose an algorithm that finds highly likely source nodes in the network.

Assuming that only a limited view on the network through pre-selected monitors is available, we also compare the accuracy of source identification algorithm using the monitors selected by six different methods.
Lastly, we propose two ways to improve our root-cause identification algorithm by leveraging additional information: error propagation probability between two network components and time when monitors get infected. For each node in the network, our algorithm calculates the approximate probability that it is the source of failure and sorts all nodes using the probability.

To maximize information obtained from monitors, we also propose a new monitor selection algorithm that makes use of the propagation probabilities.

1.4 Outline

The rest of this dissertation is organized as follows. Chapter 2 and Chapter 3 focus on the non-deterministic failures that are caused by complex interaction of multiple components. Chapter 2 presents a characteristic study on concurrency bugs. Chapter 3 explained the design of a tool that automatically pinpoints bug-triggering causal order events from execution traces.

Chapter 4 and Chapter 5 focus on the error propagation problem. Chapter 4 presents an root-cause identification algorithm which makes use of the component graph that describes information flow between pairs of components. Chapter 5 presents two ways to improve the root-cause identification by exploiting (1) propagation probabilities of edges and (2) time when monitors get infected. In both chapters, it is assumed that only a small part of the entire network is visible through a small number of pre-selected monitors. Finally, Chapter 6 concludes the dissertation.
Chapter 2

Concurrency Bug Characteristic Study

To address problems caused by concurrency bugs, it is essential to understand the characteristics of the bugs well. This chapter presents a comprehensive study on 105 concurrency bugs from four open-source programs.¹

2.1 Motivation

Modern computing systems have evolved from traditional computers that run serial logic on a central processing unit to parallel systems that make use of multiple processing units simultaneously. From multi-core processing on single computer to distributed computing on clusters of computers, parallel computing has become an essential part of computation. As a result, the difficulty of concurrent programming is hitting the entire software development community, rather than just the elite few. Writing good quality concurrent programs has become critically important. Unfortunately, writing correct concurrent programs is difficult. Most programmers think sequentially and therefore they make mistakes easily when writing concurrent programs. Even worse, the notorious non-determinism of concurrent programs makes concurrency bugs difficult to repeat during interactive diagnosis. Addressing the above challenges will require efforts from multiple related directions including those listed as follows, all of which have made some progress over the past years but still have many open, unsolved issues:

¹Part of the work presented in this chapter was done in collaboration and was published earlier [1].
(1) Concurrency bug detection

Most previous concurrency bug detection research has focused on detecting data race bugs [2–7] and deadlock bugs [3, 6, 8]. Data race occurs when two conflicting accesses to one shared variable are executed without proper synchronization, e.g., not protected by a common lock. Deadlock occurs when two or more operations circularly wait for each other to release the acquired resource (e.g., locks). Recently, several approaches have also been proposed to detect atomicity-violation bugs [9–11], which are caused by concurrent execution unexpectedly violating the atomicity of a certain code region.

Although previous work has proposed effective methods to detect certain types of concurrency bugs, it is still far from providing a complete solution. In particular, several open questions about concurrency bug detection still remain: (i) Can existing bug detection tools detect all real world concurrency bugs? Specifically, what types of concurrency bugs exist in real world? Is there any type that has not been addressed yet by existing work? In addition, are the assumptions of existing tools about concurrency bugs valid? For example, most previous race detection and many atomicity bug detection techniques focus on synchronization among accesses to a single variable. How many concurrency bugs are missed by this single variable assumption? (ii) How helpful are existing tools in diagnosing and fixing the real world concurrency bugs detected by them? For example, many concurrency bug detection tools remind programmers that some conflicting accesses are not protected by the same lock. Such information can help programmers add or change lock operations. However, how often are real world bugs fixed by adding or changing lock operations? More generally, how do programmers fix real world concurrency bugs and what information do they need?

(2) Concurrent program testing and model checking

Testing is a common practice in software development. It is a critical step for exposing software bugs before release. Existing testing techniques mainly focus on the sequential aspects
of programs, such as statements, branches, etc. and can not effectively address concurrent programs’ concurrency aspects, such as multi-thread (or multi-process) interleavings [12].

The major challenge of concurrency testing is the exponential interleaving space of concurrent programs. Exposing concurrency bugs requires not only a bug-exposing input, but also a bug-triggering execution interleaving. Therefore, to achieve a complete testing coverage of concurrent programs, testing needs to cover every possible interleaving for each input test case [13], which is infeasible in practice.

To address the above challenge, an open question in concurrency testing is as follows: can we selectively test a small number of representative interleavings and still expose most of the concurrency bugs? Motivated by this problem, previous work such as the ConTest project [14,15] has proposed some methods to perturb program execution and force certain interleavings by injecting artificial delays after every synchronization point. While an inspiring attempt, it is unclear, both quantitatively and qualitatively, what portion of concurrency bugs can be exposed by such heuristics.

Ultimately, designing practical and effective test cases for concurrent programs requires a good understanding of the manifestation conditions of real world concurrency bugs. That is, we need to know what conditions are needed, besides program inputs, to reliably trigger a concurrency bug. Specifically, how many threads, how many variables, and how many accesses are usually involved in a real world concurrency bug’s manifestation?

Similar questions are also encountered in software verification and model checking [12, 16, 17] for concurrent programs. Better understanding of the manifestation of real world concurrency bugs can help model checking prioritize the program states and alleviate its state explosion problem.

(3) Concurrent programming language design

Good concurrent programming languages can help programmers correctly express their intentions and therefore avoid certain types of concurrency bugs. Along this direction, tran-
actional memory (TM) [18–25] is one of the popular trends. TM provides programmers an
easier way to specify which code regions should be atomic. Further, it automatically pro-
tects the atomicity of the specified region against other specified regions through underlying
hardware and software support.

Although TM shows great potential, there are many open questions, including (i) What
portion of bugs can be avoided by using TM? (ii) What are the real world concerns that
TM design needs to pay attention to? (iii) Besides TM, what other programming language
supports will be useful for programmers to write correct concurrent programs?

Addressing the open questions in all of the above directions will significantly benefit from
a better understanding of real world concurrency bug characteristics — basically, we can
learn from the common mistakes programmers are making in writing concurrent programs.
For example, if many real world concurrency bugs involve multiple shared variables, we
need to extend concurrency bug detection techniques to address multi-variable concurrency
bugs; if the manifestation of most real world concurrency bugs are guaranteed by a partial
order among only two threads, concurrent program testing only needs to cover pairwise
interleavings for every pair of program threads; if there are some concerns in avoiding real
world concurrency bugs with existing synchronization primitives, we can extend transactional
memory model or design new language support to further ease writing concurrent programs;
if a certain type of information is frequently used by programmers in fixing real world
concurrency bugs, bug detection tools can be extended to provide such information and thus
become more useful in practice.

In the past, many empirical studies on general program bug characteristics (not specific
to concurrency bugs) have been done. Their findings have provided useful guidelines and
motivations for bug detection, testing and programming language design. For example, the
study of bug types in IBM software systems [26] in 1990’s demonstrated the importance of
memory bugs and has motivated many commercial and open-source memory bug detection
tools such as Purify [27], Valgrind [28], CCured [29], etc. A recent study of operating system
bugs [30] revealed that copy-paste was an important cause of semantic bugs, and has inspired a tool called CP-Miner that focused on detecting copy-pasted code and semantic bugs related to copy-paste [31].

Unfortunately, few studies have been conducted on real world concurrency bug characteristics. Previously, researchers realizing the importance of such a study have conducted a preliminary work on concurrency bug characteristics [32]. However, they built their observations upon programs that were intentionally made buggy by students for the characteristic study.

The lack of a good real-world concurrency bug characteristic study is mainly due to the following two reasons:

1. It is difficult to collect real world concurrency bugs, especially since they are usually under-reported. As observed in previous work [33], the non-determinism hindered the users from reporting concurrency bugs, and made concurrency bug reports difficult to get understood and solved by programmers. Therefore, it is time-consuming to collect a good set of real world concurrency bugs.

2. Concurrency bugs are not easy to understand. Their patterns and manifestations usually involve complicated interactions among multiple program components, and are therefore hard to understand.

### 2.2 Methodology

#### 2.2.1 Bug Sources

**Applications:** We select four representative open source applications in our study: MySQL, Apache, Mozilla, and OpenOffice. These are all mature (with 9–13 years development history) large concurrent applications (with 1–4 million lines of code), with well maintained bug databases. These four applications represent different types of server applications (database
and web server) and client applications (browser suite and office suite). Concurrency is used for different purposes in these applications. Server applications mostly use concurrency to handle concurrent client requests. They can have hundreds or thousands of threads running at the same time. Client and office applications mostly use concurrency to synchronize multiple GUI sessions and background working threads.

Bugs: We randomly collect concurrency bugs from the bug databases of these applications. Since these databases contain more than five hundred thousand bug reports, in order to effectively collect concurrency bugs from them, we used a large set of keywords related to concurrency bugs, for example, race(s), deadlock(s), synchronization(s), concurrency lock(s), mutex(es), atomic, compete(s) and their variations. From the thousands of bug-reports that contain at least one keyword from the above keyword set, we randomly pick about five hundred bug reports with clear and detailed root cause descriptions, source codes, and bug fix information. Then, we manually check them to make sure that the bugs are really caused by programmers’ wrong assumptions about concurrent execution, and finally get 105 concurrency bugs.

We separately study two types of concurrency bugs: deadlock bugs and non-deadlock concurrency bugs. These two types of bugs have completely different properties, and demand different detection, recovery approaches. Therefore, we separate them for the ease of investigation.

Finally, we collect 105 concurrency bugs, including 74 non-deadlock concurrency bugs and 31 deadlocks bugs. The details are shown in Table 2.2.

2.2.2 Characteristic Categories

In order to provide guidance for future research on concurrent program reliability, we focus on three aspects of concurrency bug characteristics: bug pattern, manifestation, and bug fix strategy. Other characteristics, such as failure impact and bug diagnosis process, will be briefly discussed at the end.
Along the bug pattern dimension, we classify non-deadlock concurrency bugs into three categories (atomicity-violation bugs, order-violation bugs and the other bugs) based on their root causes, i.e., what types of synchronization intentions are violated. Detailed definitions are shown in Table 2.3. Here, we do not classify data race as a bug pattern. The reason is that, a data race may indicate a concurrency bug, but it can also be a benign race in many cases, e.g., while-flag. Furthermore, data-race free does not mean concurrency bug free [9, 10]. We do not further break deadlocks into subcategories, as most of them are relatively similar and simple.

For the manifestation characteristics, we study the required condition for each concurrency bug to manifest (denoted as manifestation condition, defined in Table 2.3), and then discuss concurrency bugs based on how many threads, how many variables (resources), and how many accesses are involved in their manifestation conditions.

For the bug fix strategy, we study both the final patch’s fix strategy and the mistakes in intermediate patches. We also evaluate how transactional memory can help avoid these bugs. All the related classification is shown in Table 2.3.

2.2.3 Threats to Validity

Similar to the previous work, real world characteristic studies are all subject to a validity problem. Potential threats to the validity of our characteristic study are the representativeness of the applications, concurrency bugs used in our study, and our examination methodology.

As for application representativeness, our study chooses four server and client-based concurrent applications written in C/C++, which are the popular programming languages for these types of applications. We believe that these four applications well represent server and client-based concurrent applications, which are two large classes of concurrent applications. However, our study may not reflect the characteristics of other types of applications, such as scientific applications, operating systems, or applications written in other programming
As for bug representativeness, the concurrency bugs we studied are randomly selected from the bug database of the above applications. They provide good samples of the fixed bugs in those applications. While characteristics of non-fixed or non-reported concurrency bugs might be different, these bugs are not likely as important as the reported and fixed bugs that are examined in our study.

In terms of our examination methodology, we have examined every piece of information related to each examined bug, including programmers’ clear explanations, forum discussions, source code patches, multiple versions of source codes, and bug-triggering test cases. In addition, we are also familiar with the examined applications.

Overall, while our conclusions cannot be applied to all concurrent programs, we believe that our study does capture the characteristics of concurrency bugs in two large important classes of concurrent applications: server-based and client-based applications. In addition, most of these characteristics are consistent across all four examined applications, indicating the validity of our evaluation methodology to some degree. Additionally, we do not emphasize any quantitative characteristic results. Finally, we also warn the readers to take our findings together with above methodology and selected applications.

2.3 Bug Pattern Study

Different bug patterns usually demand different detection and diagnosis approaches. In Table 2.4, we classify the patterns of the examined non-deadlock concurrency bugs into three categories: Atomicity, Order, and Other, which are described in Table 2.3. Note that the categories are distinguished from each other by the root cause of a bug, regardless of the possible bug fix strategies.
### Findings on Bug Patterns (Section 2.3)

<table>
<thead>
<tr>
<th>Implications</th>
<th>Findings on Bug Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Almost all (97%) of the examined non-deadlock bugs belong to one of the two simple bug patterns: atomicity-violation or order-violation.</td>
<td>Concurrency bug detection can focus on these two bug patterns to detect most concurrency bugs.</td>
</tr>
<tr>
<td>(2) About one third (32%) of the examined non-deadlock bugs are order-violation bugs, which are not well addressed in previous work.</td>
<td>New concurrency bug detection tools are needed to detect order-violation bugs, which are not addressed by existing atomicity violation or race detectors.</td>
</tr>
</tbody>
</table>

### Findings on Bug Manifestation (Section 2.4)

<table>
<thead>
<tr>
<th>Implications</th>
<th>Findings on Bug Manifestation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3) Almost all (96%) of the examined concurrency bugs are guaranteed to manifest if certain partial order between 2 threads is enforced.</td>
<td>Pairwise testing on concurrent program threads can expose most concurrency bugs, and greatly reduce the testing complexity.</td>
</tr>
<tr>
<td>(4) Some (22%) of the examined deadlock bugs are caused by one thread acquiring resource held by itself.</td>
<td>Single-thread based deadlock detection and testing techniques can help eliminate these simple deadlock bugs.</td>
</tr>
<tr>
<td>(5) Many (66%) of the examined non-deadlock concurrency bugs’ manifestation involves concurrent accesses to only one variable.</td>
<td>Focusing on concurrent accesses to one variable is a good simplification for concurrency bug detection, which is used by many existing bug detectors.</td>
</tr>
<tr>
<td>(6) One third (34%) of the examined non-deadlock concurrency bugs’ manifestation involves concurrent accesses to multiple variables.</td>
<td>New detection tools are needed to address multiple variable concurrency bugs.</td>
</tr>
<tr>
<td>(7) Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most two resources.</td>
<td>Pairwise testing on the acquisition/release sequences to two resources can expose most deadlock concurrency bugs, and reduce testing complexity.</td>
</tr>
<tr>
<td>(8) Almost all (92%) of the examined concurrency bugs are guaranteed to manifest if certain partial order among no more than 4 memory accesses is enforced.</td>
<td>Testing partial orders among every small group of accesses can expose most concurrency bugs, and simplify the interleaving space from exponential to polynomial.</td>
</tr>
</tbody>
</table>

### Findings on Bug Fix Strategies (Section 2.5)

<table>
<thead>
<tr>
<th>Implications</th>
<th>Findings on Bug Fix Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>(9) Three quarters (73%) of the examined non-deadlock bugs are fixed by techniques other than adding/changing locks. Programmers need to consider correctness, performance and other issues to decide the most appropriate fix strategy.</td>
<td>Bug detection and diagnosis tools need to provide more bug pattern and manifestation information, besides lock information, to help programmers fix bugs.</td>
</tr>
<tr>
<td>(10) Many (61%) of the examined deadlock bugs are fixed by preventing one thread from acquiring one resource (e.g. lock). Such fix can introduce non-deadlock concurrency bugs.</td>
<td>Fixing deadlock bugs might introduce non-deadlock concurrency bugs. Special help is needed to ensure the correctness of deadlock bug fixes.</td>
</tr>
</tbody>
</table>

### Findings on Bug Avoidance (Section 2.5.3)

<table>
<thead>
<tr>
<th>Implications</th>
<th>Findings on Bug Avoidance</th>
</tr>
</thead>
<tbody>
<tr>
<td>(11) Transactional memory (TM) can help avoid about one third (39%) of the examined concurrency bugs.</td>
<td>Transactional memory (TM) is a promising language feature for programmers.</td>
</tr>
<tr>
<td>(12) TM could help avoid over one third (42%) of the examined concurrency bugs, if some concerns are addressed.</td>
<td>TM designers may need to pay attention to some concerns, such as how to protect hard-to-rollback operations.</td>
</tr>
<tr>
<td>(13) Some (19%) of the examined concurrency bugs cannot benefit from basic TM designs, because of their bug patterns.</td>
<td>Better programming language features to help express “order” semantics in C/C++ programs are desired.</td>
</tr>
</tbody>
</table>

Table 2.1: Our findings of real world concurrency bug characteristic and their implications for concurrency bug detection, concurrent program testing and concurrent programming language design. (*: All terms and categories mentioned here are explained in Section 2.2.)
The Finding (1) can be explained by the fact that programmers generally put their intentions on atomic regions and execution orders, but it is not easy to enforce all these intentions correctly and completely in implementation.

Since programmers think sequentially, they tend to assume that small code regions will be executed atomically. For example, in Figure 2.1, programmers assume that if S1 reads a non-NULL value from thd->proc_info, S2 will also read the same value. However, such an atomicity assumption can be violated by S3 during concurrent execution, and it leads to a program crash.

It is also common for programmers to assume an order between two operations from different threads, but programmers may forget to enforce such an order. As a result, one of
### Definitions Related to Bug Pattern Study

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Category</th>
<th>Description</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug Pattern*</td>
<td>Atomicity Violation</td>
<td>The desired serializability among multiple memory accesses is violated. (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution.)</td>
<td>Atomicity</td>
</tr>
<tr>
<td></td>
<td>Order Violation</td>
<td>The desired order between two (groups of) memory accesses is flipped. (i.e. A should always be executed before B, but the order is not enforced during execution.)</td>
<td>Order</td>
</tr>
<tr>
<td></td>
<td>Other</td>
<td>Concurrency bugs other than the atomicity violation and order violation. *</td>
<td>Other</td>
</tr>
</tbody>
</table>

### Definitions Related to Bug Manifestation Study

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug Manifestation</td>
<td>Manifestation</td>
<td>A specific execution order among a smallest set (S) of memory accesses. As long as that order is enforced, no matter how, the bug is guaranteed to manifest.</td>
</tr>
<tr>
<td></td>
<td># of threads involved</td>
<td>The number of distinct threads that are included in S.</td>
</tr>
<tr>
<td></td>
<td># of variables involved</td>
<td>The number of distinct variables that are included in S.</td>
</tr>
<tr>
<td></td>
<td># of accesses involved</td>
<td>The number of accesses that are included in S.</td>
</tr>
</tbody>
</table>

### Definitions Related to Bug Fix Strategy

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Category</th>
<th>Description</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-deadlock</td>
<td>Condition Check</td>
<td>(1) While-flag; or (2) optimistic concurrency with consistency check.</td>
<td>COND</td>
</tr>
<tr>
<td>Fix Strategy</td>
<td>Code Switch</td>
<td>Switch the order of certain statements in the source code.</td>
<td>Switch</td>
</tr>
<tr>
<td></td>
<td>Design Change</td>
<td>Change the design of data structures or algorithms.</td>
<td>Design</td>
</tr>
<tr>
<td></td>
<td>Lock Strategy</td>
<td>(1) Add/change locks; or (2) adjust the region of critical sections.</td>
<td>Lock</td>
</tr>
<tr>
<td></td>
<td>Other</td>
<td>Strategies other than the above ones.</td>
<td>Other</td>
</tr>
<tr>
<td>Deadlock</td>
<td>Give up resource</td>
<td>Not acquiring a resource (lock, etc.) for certain code region.</td>
<td>GiveUp</td>
</tr>
<tr>
<td>Fix Strategy</td>
<td>Split Resource</td>
<td>Split a big resource to smaller pieces to avoid competition.</td>
<td>Split</td>
</tr>
<tr>
<td></td>
<td>Change acquisition order</td>
<td>Switch the acquisition order among several resources.</td>
<td>AcqOrder</td>
</tr>
<tr>
<td></td>
<td>Other</td>
<td>Strategies other than the above ones.</td>
<td>Other</td>
</tr>
<tr>
<td>Concerns on Transactional Memory</td>
<td>Very long code</td>
<td>A code region is too long to be put into a transaction.</td>
<td>Long</td>
</tr>
<tr>
<td></td>
<td>Rollback Problem</td>
<td>Some I/O and system calls are hard to roll back.</td>
<td>Rollback</td>
</tr>
<tr>
<td></td>
<td>Code Nature</td>
<td>Source code with certain design is hard to turn to transaction.</td>
<td>Nature</td>
</tr>
</tbody>
</table>

Table 2.3: Our characteristic categories and definitions. (*: The bug pattern category is determined by the root cause of a concurrency bug, i.e. what type of programmers’ synchronization intention is violated, regardless of possible bug fix strategies.)
Table 2.4: Patterns of non-deadlock concurrency bugs. (There are 3 examined bugs, whose patterns can be considered as either atomicity or order violation. Therefore, they are considered in both categories.)

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>Atomicity</th>
<th>Order</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>14</td>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache</td>
<td>13</td>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>41</td>
<td>29</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>74</td>
<td>51</td>
<td>24</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 2.2: An order violation bug from Mozilla. The program fails to enforce the programmers’ order intention: thread 2 should not read \texttt{mThread} until thread 1 initializes \texttt{mThread}. Note that, this bug could be fixed by making \texttt{PR\_CreateThread} atomic with the write to \texttt{mThread}. However, our bug pattern categorization is based on the root cause, regardless of possible fix strategies.

the two operations may be executed faster (or slower) than the programmers’ assumption, and it makes the order bug manifest. In the Mozilla bug shown in Figure 2.2, it is easy for programmers to assume wrongly that thread 2 would dereference \texttt{mThread} after thread 1 initializes it, because thread 2 is created by thread 1. However, in real execution, thread 2 may be very quick and dereference \texttt{mThread} before \texttt{mThread} is initialized. This unexpected order leads to program crash. Note that even though the bug can be fixed with locks, the root cause of the bug is not an atomicity violation, but an order violation.

Concurrence bugs violating other types of programmers’ intentions also exist, but are much rarer as shown in Table 2.4. Figure 2.3 shows an example. In one version of MySQL, programmers use a timeout threshold \texttt{fatal\_timeout} to detect deadlock. The server will
Figure 2.3: A MySQL bug that is neither an atomicity-violation bug nor an order-violation bug. The monitor thread is designed to detect deadlock. It restarts the server when a thread i has waited for a lock for more than fatal_timeout amount of time. In this bug, programmers under-estimate the workload (n could be very large), and therefore the lock waiting time would frequently exceed fatal_timeout and crash the server.

Figure 2.4: A write-write order violation bug from Mozilla. The program fails to enforce the programmers' order intention: thread 2 is expected to write io_pending to be FALSE some time after thread 1 initializes it to be TRUE. Note that, this bug could be fixed by making S1 and S2 atomic. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies.

crash, if any thread waits for a lock for more than fatal_timeout amount of time. However, when programmers set the threshold, they under-estimate the workload. As a result, users found that the MySQL server keeps crashing under heavy workload (with 2048 worker-thread setting). Such a performance-related assumption is neither atomicity intention nor order intention. This bug is fixed by limiting the number of worker-threads.
Figure 2.5: A Mozilla bug that violates the intended order between two groups of operations.

**Finding (2):** A significant number (24 out of 74) of the examined non-deadlock concurrency bugs are order bugs, which are not addressed by previous bug detection work.

**Implications:** New bug detection techniques are desired to address order bugs.

As we discussed above, it is common for programmers to assume a certain order between two operations from two threads. Specifically, programmers can have an order intention i) between a write and a read (Figure 2.2) to one variable; ii) between two writes (Figure 2.4) to one variable; or iii) between two groups of accesses to a group of variables (Figure 2.5). In Figure 2.4, programmers expect S2 to initialize io pending before S4 assigns a new value, `FALSE`, to it. However, the execution of the asynchronous read can be very quick and S4 may be executed before S2, contrary to the expectation of programmers. This makes thread 1 to hang. In another example shown in Figure 2.5, `js_UnpinPinnedAtom` frees all elements in the atoms array. This set of memory accesses to the whole array is expected to happen after `js_MarkAtom`, which may access some elements in atoms.

Note that the above order bugs are different from data race bugs and atomicity violation bugs. Even if the same lock protects two memory accesses to the same variable or two conflicting code regions are atomic to each other, the execution order between them still may not be guaranteed. We should also note that some order-violation bugs could be fixed using coarser-grained locking, as in example Figure 2.2 and Figure 2.4; some others cannot be fixed by locks, as in example Figure 2.5 and Figure 2.7 (will be discussed later).
not related to the bug root cause, and does not affect our bug pattern classification.

Although important and common, order-violation bugs have not been well studied by previous research. Existing concurrency bug detectors, which mainly focus on race bugs or atomicity bugs, will miss many order bugs. New techniques are desired for solving the order problems.

2.4 Bug Manifestation Study

Manifestation condition of a concurrency bug is usually a specific order among a set of memory accesses or system events. In this section, we study the characteristics of real world concurrency bug manifestation, following the methodology defined in Table 2.3. We will discuss guidance for concurrent program testing and concurrency bug detection based on our observations.

2.4.1 How many threads are involved?

| Finding (3): The manifestation of most (101 out of 105) examined concurrency bugs involves no more than two threads. |
| Implications: Concurrent program testing can pairwise test program threads, which reduces testing complexity without losing bug exposing capability much. |

Finding (3) tells us that even though the examined server programs use hundreds of threads, in most cases, only a small number (mostly just two) of threads are involved in the manifestation of a concurrency bug.

The underlying reason for this is that most threads do not closely interact with many others, and most communication and collaboration is conducted between two or a small group of threads. As a result, manifestation conditions of most concurrency bugs do not involve many threads. For examples, all of the bugs presented in
Section 2.3, except the one shown in Figure 2.3, are guaranteed to manifest if their execution follow certain partial orders (marked by dotted lines in the figures) between two threads.

We should note that this finding is not opposite to the common observation that concurrency bugs are sometimes easier to manifest at a heavy-workload (concurrent execution of many threads). In many cases, the manifestation condition involves only two threads. Heavy-workload increases the resource competition and context switch intensity. It therefore increases the possibility of hitting certain orders between the two threads that can trigger the bug. The manifestation condition still involves just two threads.

Our finding implies that testing can focus on execution orders among accesses from every pair of threads. Such pairwise testing technique can prevent the testing complexity from increasing exponentially with the number of threads. At the meantime, few concurrency bugs would be missed.

There are also cases where the bug manifestation relies on not only memory accesses within the program, but also environmental events (as shown in column ‘Env’ in Table 2.5). For example, one Mozilla bug cannot be triggered unless another program modifies the same file concurrently with Mozilla. Exposing such bugs needs special system support.

**Finding (4)**: The manifestation of some (7 out of 31) deadlock concurrency bugs involves only one thread.

**Implications**: This type of bug is relatively easy to detect and avoid. Bug detection and programming language techniques can try to eliminate these simple bugs first.

It usually happens when one thread tries to acquire a resource held by itself. Detecting and analyzing this type of bugs are relatively easy, because we do not need to consider the contention from other concurrent execution components.
### Table 2.5: The number of threads (or environments) involved in concurrency bugs.

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>Env.</th>
<th>&gt; 2 threads</th>
<th>2 threads</th>
<th>1 thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Apache</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>41</td>
<td>1</td>
<td>0</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>74</td>
<td>2</td>
<td>1</td>
<td>71</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>Env.</th>
<th>&gt; 2 threads</th>
<th>2 threads</th>
<th>1 thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Apache</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>31</td>
<td>0</td>
<td>1</td>
<td>23</td>
<td>7</td>
</tr>
</tbody>
</table>

### 2.4.2 How many variables are involved?

Are concurrency bugs synchronization problems among accesses to one variable or multiple variables? To answer this question, we examine the number of variables (or resources) involved in the manifestation of each concurrency bug. The examination result is shown in Table 2.6.

**Finding (5):** 66% (49 out of 74) of the examined non-deadlock concurrency bugs involve only one variable.

**Implications:** Focusing on concurrent accesses to one variable is a good simplification for concurrency bug detection.

Finding (5) confirms our intuition. Flipping the order of two accesses to different memory locations does not directly change the program state, and therefore is less likely to cause problems. Figure 2.1, 2.2, and 2.4 are all examples of single variable concurrency bugs: their manifestation can be guaranteed by certain order among accesses to one variable. This finding supports the single-variable assumption taken by many existing bug detectors. For example, data race bug detection [6, 7] checks the synchronization among accesses to one variable.
Table 2.6: The number of variables (resources) involved in concurrency bugs.

variable; some atomicity violation bug detection tools also focus on atomic regions related to one variable [10,11].

Finding (6): A non-negligible number (34%) of non-deadlock concurrency bugs involve more than one variable.

Implications: We need new concurrency bug detection tools to address multiple variable concurrency bugs.

Multiple variable concurrency bugs usually occur when unsynchronized accesses to correlated variables cause inconsistent program state. Semantic connections among variables are common, and therefore, multiple variable concurrency bugs are common too. Figure 2.6 shows an example of multiple variable concurrency bug from Mozilla. In this example, mOffset and mLength together mark the region of useful characters stored in dynamic string mContent. Thread 1 and 2’s concurrent accesses to these three variables should be synchronized; otherwise thread 1 might read inconsistent values and access invalid memory address. Here, controlling the order of memory accesses to any single variable, cannot guarantee the bug to manifest. For example, it is not wrong for thread 1 to read mContent either before
Figure 2.6: A multi-variable concurrency bug from Mozilla. Accesses to three correlated variables, mContent, mOffset and mLength, should be synchronized.

or after thread 2’s modification to all of three variables. The required condition for the bug manifestation is that thread 1 uses the three correlated variables in the middle of thread 2’s modification to these three variables.

As discussed above, most existing bug detection tools only focus on single-variable concurrency bugs. Although this simplification provides a good starting point for concurrency bug detection, future research should not ignore the problem of multi-variable concurrency bugs. The difficulty of detecting multiple variable concurrency bugs is that it is hard to infer which accesses, to different variables, should be well synchronized. Solving this problem will not only benefit automatic concurrency bug detection, but also provide useful hints for programmers to specify correct transactions or atomic regions for transactional memory or atomicity bug detection tools [9].

Finding (7): 97% (30 out of 31) of the examined deadlock concurrency bugs involve at most two resources.

Implications: Deadlock-oriented concurrent program testing can pairwise test the order among acquisition and release of two resources.

Among the examined deadlock bugs, three threads circularly waiting for three resources
trigger only one bug. Leveraging this finding, pairwise testing on resources can prevent the
testing complexity from increasing exponentially with the total number of resources.

2.4.3 How many accesses are involved?

We find that the manifestation of most concurrency bugs involves only two threads and a
small number of variables. However, the number of accesses from one thread to each variable
can still be huge. Therefore, we need to study how many accesses are involved in the bug
manifestation.

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>1 acc.*</th>
<th>2 acc.</th>
<th>3 acc.</th>
<th>4 acc.</th>
<th>&gt; 4 acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Apache</td>
<td>13</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>41</td>
<td>0</td>
<td>12</td>
<td>18</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>74</td>
<td>0</td>
<td>22</td>
<td>33</td>
<td>12</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>1 acc.*</th>
<th>2 acc.</th>
<th>3 acc.</th>
<th>4 acc.</th>
<th>&gt; 4 acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Apache</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>16</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>31</td>
<td>7</td>
<td>3</td>
<td>20</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.7: The number of accesses (or resource acquisition/release) involved in concurrency
bugs. (*: “1 acc.” case happens only in deadlock bugs, when one thread waits for itself.
The bug triggering therefore does not depend on any inter-thread order problem.)
Finding (8.1): 90% (67 out of 74) of the examined non-deadlock bugs can deterministically manifest, if certain orders among at most four memory accesses are enforced.

Finding (8.2): 97% (30 out of 31) of the examined deadlock bugs can deterministically manifest, if certain orders among at most four resource acquisition/release operations are enforced.

Implications: Concurrent program testing can focus on the partial order among every small groups of accesses. This simplifies the interleaving testing space from exponential to polynomial regarding to the total number of accesses, with little loss of bug exposing capability.

The Finding (8.1) can be easily understood, considering that most of the examined concurrency bugs have simple patterns and involve a small number of variables. Most of the exceptions come from those bugs that involve more than two threads and/or more than two variables.

The Finding (8.2) is also natural, considering that most of our examined deadlock bugs involve only two resources.

The above findings have significant implication for concurrent program testing. The challenge in concurrent program testing is that the number of all possible interleavings is exponential to the number of dynamic memory accesses, which is too big to thoroughly explore. Our finding provides support to a more effective design of interleaving testing [34]: exploring all possible orders within every small groups of memory accesses, e.g. groups of 4 memory accesses. The complexity of this design is only polynomial to the number of dynamic memory accesses, which is a huge reduction from the exponential-sized all-interleaving testing scheme. Furthermore, the bug exposing capability of this design is almost as good as exploring all interleavings. It would miss only few bugs in our examination.

A recent model checking work [12] uses the heuristic to start the checking from interleav-
ings with small numbers of context switches. Our study provides support for this heuristic.

Of course, enforcing a specific partial order among a set of accesses is not trivial. The program input and many accesses need to be controlled to achieve that. How to leverage our finding to enable practical and powerful concurrent program testing and model checking remains as future work.

## 2.5 Bug Fix Study

### 2.5.1 Fix Strategies

Before we check how the real world bugs were fixed, our guess was that adding or changing locks should be the most common way to fix concurrency bugs. However, the characteristic result is contrary to our guess, as shown in Table 2.8.

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>COND</th>
<th>Switch</th>
<th>Design</th>
<th>Lock</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Apache</td>
<td>13</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>41</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>74</td>
<td>19</td>
<td>10</td>
<td>19</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2.8: Fix strategies for non-deadlock concurrency bugs (all categories are explained in Table 2.3).

**Finding (9):** Adding or changing locks is not the major fix strategy. It is used for only 20 out of 74 non-deadlock concurrency bugs that we examined.

**Implication:** There is no silver bullet for fixing concurrency bugs. Just telling programmers that certain conflicting accesses are not protected by the same lock is not enough to fix concurrency bugs.

There are two reasons for this controversy. First of all, locks cannot guarantee to enforce some synchronization intentions, such as $A$ should happen before $B$. Therefore, adding/changing locks cannot fix certain types of bugs. Figure 2.5 shows such an example.
The Finding (8.1) can be easily understood, considering that most of the examined concurrency bugs have simple patterns. Some of these patterns are difficult to fix by adding/changing locks and can lead to bugs like the one shown in Figure 2.7. A MySQL bug that cannot be fixed by adding/changing locks.

![Figure 2.7: A MySQL bug that cannot be fixed by adding/changing locks.](image)

Here we show another simple example in Figure 2.7. Secondly, even if adding/changing locks can fix a bug, in many cases, it is not the best strategy, because it may hurt the performance or introduce new bugs, such as deadlock bugs.

In the following, we describe the different strategies, other than adding/changing locks, used by programmers. We will see that these strategies usually require deep understanding of program semantics. At the mean time, they usually have better performance than corresponding lock-based fixes, if existing.

1. Condition check (denoted as COND). Condition check can be used in different ways to help fix concurrency bugs. One way is to use while-flag to fix order-related bugs, such as the bug shown in Figure 2.5. The other way is to add consistency check to monitor the bug-related program states. This enables the program to detect buggy interleavings
and restore program states. For example, to fix the bug shown in Figure 2.6, the program does consistency check \( \text{if}(\text{strlen(mContent)} \geq mOffset+mLength) \) before it executes \texttt{putc} function. The \texttt{putc} will be skipped if the consistency check fails. In another example shown in Figure 2.8, condition \((n! = \text{block} \rightarrow n)\) is checked to see whether the shared variable \texttt{block} \rightarrow \text{n} has been overwritten since the last time it was read. If \text{n} is not consistent with \texttt{block} \rightarrow \text{n}, the program rolls back and reads \texttt{block} \rightarrow \text{n} again. Note that, above fix strategy does not eliminate the buggy interleaving, which is usually the purpose of lock-based fixes. Instead, it focuses on detecting buggy interleavings and makes sure the program states corrupted by the buggy interleavings can be recovered in time. It has better performance than corresponding lock-based fixes.

(2) Code switch (denoted as Switch). Switching the order of certain code statements can fix some order-related bugs. For example, the order bug shown in Figure 2.4 is fixed by switching statements S1 and S2, so that S2 is always executed before S4.

(3) Algorithm/Data-structure design change (denoted as Design). This includes different types of algorithm changes and data structure changes that help to achieve correct synchronization. Some design changes are simple, just modifying a few data structures. For example, in the MySQL bug #7209, the bug is caused by unprotected conflicting accesses to a shared variable \texttt{HASH::current} record. Programmers recognize that this variable does not need to be shared. They simply move the field current record out of the class \texttt{HASH}, making it a local variable for each thread, and fix the bug. As another example, in Mozilla bug #201134, one thread needs to conduct a series of operations on a shared variable \texttt{nsCertType}. In order to enforce the atomicity of that series of operations, programmers simply let program read \texttt{nsCertType} into a local variable, conduct operations on the local variable, and store the value back to \texttt{nsCertType} at the end. Some design changes are more complicated, involving algorithm re-design. For example, in Mozilla bug #131447, programmers changed a message handling and queueing algorithm to tolerate special timing when a reply message arrives before its corresponding callback function is ready.
As we can see, fixing concurrency bugs is much more complicated than just adding or changing lock operations. Race detection tools can help programmers conduct those lock-related fixes, but this is not enough. It is desired to have more tools to help programmers figure out the bug pattern, the consistency condition associated with each bug, etc. For example, if programmers know that the bug is an order-violation bug and they also know what the consistency condition is, it is easy to come out with a condition check fix. This is the challenge for future research on concurrency bug detection and diagnosis.

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>GiveUp</th>
<th>Split</th>
<th>AcqOrder</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>9</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Apache</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>16</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>31</td>
<td>19</td>
<td>1</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.9: Fix strategies for deadlock bugs (all categories are explained in Table 2.3)

**Finding (10):** The most common fix strategy (used in 19 out of 31 cases) for the examined deadlock bugs is to let one thread give up acquiring one resource, such as a lock. This strategy is simple, but it may introduce other non-deadlock bugs.

**Implication:** We need to pay attention to the correctness of some “fixed” deadlock bugs.

In many cases, programmers find it unnecessary or not worthwhile to acquire a lock within certain program context. Therefore, they simply drop the resource acquisition to avoid the deadlock. However, this strategy could introduce non-deadlock concurrency bugs. In some of our examined bug reports, programmers explicitly say that they know the fix would introduce a new non-deadlock concurrency bug. They still adopt the fix, because they gamble that the probability for the non-deadlock bug to occur is small. In the future, techniques combining optimistic concurrency and rollback-reexecution, such as TM, can help fix some deadlock bugs. Of course, using these techniques should also be careful, because
they might introduce live-lock problems.

2.5.2 Mistakes during bug fixing

Fixing bugs is hard. Some patches released by programmers are still buggy. In order to investigate the nature of buggy patches, we collect all the distinct buggy patches of the 57 Mozilla concurrency bugs. Specifically, we first gather all the intermediate (non-final) patches submitted by Mozilla programmers for these 57 bugs. We then manually check each patch and filter out non-bug fixing patches, which only change comments or code structures for maintenance purpose. Our study finds that 17 out of the 57 Mozilla bugs have at least one buggy patches. On average, 0.4 buggy patches were released before every final correct patch. Among all the 23 distinct buggy patches, 6 of them only decrease the occurrence probability of the original concurrency bug, but fail to fix the original bug completely (an example is shown in Figure 2.9). 5 of them introduce new concurrency bugs. The other 12 introduce new non-concurrency bugs. Programmers need help to improve the quality of their patches.

2.5.3 Discussion: bug avoidance

Good programming languages should help avoid some bugs during implementation. Transactional memory (TM) is a popular trend of programming language feature for easing concurrent programming. To estimate its benefit and what more are needed along this direction, we study the 105 concurrency bugs to see how many of them can potentially be avoided with TM support. Furthermore, we study what are the issues that future concurrent programming language design needs to address. Again, our analysis should be interpreted with our examined applications and evaluation methodology in mind, as discussed in Section 2.2.3. In addition, since different TM designs may have different features, in our discussion, we

\[\text{We focus on Mozilla, because it has the best maintenance of patch update information.}\]
focus on the basic atomicity and isolation properties of TM. We discuss the benefits and concerns in general, based on such basic TM designs [19,21–23]. It is definitely possible for advanced TM designs to address some of the concerns we will discuss, which is exactly the purpose of our discussion: provide more real-world information and help improve the design of TM.

![Diagram of code examples]

(a) an incomplete fix for the bug shown in Figure 5.
This fix left a small window between S1 and S2 unprotected.

(b) a final correct fix.
Now the order between js_MarkAtom and js_UnpinPinnedAtom is enforced.

Figure 2.9: The process of fixing the bug shown in Figure 2.5. Programmers want to make sure js_MarkAtom will not be called after js_UnpinPinnedAtom. They first added a flag variable state to fix the bug. However, that left a small window between S1 and S2 unprotected. They finally added a second flag variable gcLevel to completely fix the bug.

**Finding (11):** TM can help avoid many concurrency bugs (41 out of the 105 concurrency bugs we examined).

**Implication:** Although TM is not a panacea, it can ease programmers correctly expressing their synchronization intentions in many cases, and help avoid a big portion of concurrency bugs.
Table 2.10: Can TM help avoid concurrency bugs? (*: There are some concerns)

Atomicity violation bugs and deadlock bugs with relatively small and simple critical code regions can benefit the most from TM, which can help programmers clearly specify this type of atomicity intention. Figure 2.8 shows an example, where programmers use a consistency check with re-execution to fix the bug. Here, a transaction (with abort, rollback and replay) is exactly what programmers want.

Finding (12): TM can potentially help avoid many concurrency bugs (44 out of the 105 concurrency bugs we examined), if some concerns can be addressed, as shown in Table 2.10.

Implication: TM design can combine system supports and other techniques to solve some of these concerns, and further ease the concurrent programming.

One concern, not a surprise, is I/O operations. As operations like I/O are hard to roll back, it is hard to use TM to protect the atomicity of code regions that include such operations. Take the concurrency bug in Figure 2.1 as an example. Since S2 involves a file operation, TM might need non-trivial undo techniques to protect the S1–S2 atomic region. Other concerns, such as atomic region size and special code nature, also exist. For example, the atomic code regions of several Mozilla bugs include the whole garbage collection process. These regions could have too large memory footprint to be effectively handled by hardware-TM. Many of the above concerns are addressable in TM, but with higher overhead and complexity. For example, some of the roll-back concerns can be addressed using system supports. Very long transactions can be addressed by combining software and hardware
Finding (13): 20 out of the 105 concurrency bugs that we examined cannot benefit from the basic TM designs, because the violated programmer intentions, such as order intentions, cannot be guaranteed by the basic TM.

Implications: Apart from atomicity intentions, there is also a significant need for concurrent programming language features to help programmers express order intentions easily.

Programmers’ order intention is the major type of intention that cannot be easily enforced by the basic TM design or locks. In general, the basic TM designs cannot help enforce the intention that $A$ has to be executed before $B$. Therefore, they cannot help avoid many related order-violation bugs. Among all order-violation bugs, we find a sub-type of order intentions that are extremely hard to be enforced by basic TM designs: $A$ must be either executed before $B$ or not executed at all. In other words, programmers do not want $B$ to wait for $A$. They simply skip $A$ if $B$ is already executed. For example, in one Mozilla bug, thread 1 keeps inserting entities to a cache and thread 2 would destroy the cache at some moment. Based on the description in the bug report, programmers do not want thread 2 to wait for thread 1 to finish all insertions. The program simply skips any insertion attempt after the cache is destroyed. This happens for 7 bugs.

In order to help avoid above 20 bugs, the semantic design, instead of implementation schemes, of the basic TM needs to be enhanced. Recently, some TM designs [35, 36] are equipped with rich semantics (such as watch/retry, retry/orElse) and can help enforce some of the above synchronization intentions. We hope our bug characteristic study can help future research to decide the best TM design.

---

Some order-violation bugs can be avoided by TM. In those cases, order intentions can be enforced as side effects while TM enforces the atomicity of related code regions (an example is shown in Figure 2.2).
2.6 Other Characteristics

**Bug impacts** Among our examined concurrency bugs, 34 of them can cause program crashes and 37 of them cause program hangs. This validates that concurrency bug is a severe reliability problem.

**Some concurrency bugs are very difficult to repeat.** In one bug report (Mozilla bug #52111), the reporter complained that “I develop Mozilla full time all day, and I get this bug only once a day”. In another bug report (Mozilla bug #72599), the reporter said that “I saw it only once ever on g (never on other machines). Perhaps the dual processor of g makes it occur.”

**Test cases are critical to bug diagnosis** Programmers’ discussions show that a good test case to repeat a concurrency bug is very important for diagnosis. In Mozilla bug #73291, the programmers once gave up on this bug and closed the bug report, because they could not repeat the bug. Fortunately, somebody worked out a way to reliably repeat the bug, and the bug was fixed subsequently. In another Mozilla bug report (Mozilla bug #72599), the programmers finally gave up repeating the bug and simply submit a patch based on their “guessing”, and this led to a wrong fix.

**Programmers lack diagnosis tools** From the bug reports, we notice that many concurrency bugs are diagnosed simply by programmers reading the source code. For example, for 29 out of the 57 Mozilla bugs, the bug reports did not mention that the programmers ever leveraged any information from any tools, core dumps, or stack traces, etc. Sometimes programmers tried gdb, but could not get useful information. We have never seen programmers mentioned that they used any automatic diagnosis tools. In contrast, in many bug reports about memory bugs, programmers mentioned that they got help from Valgrind, Purify, etc [37].
2.7 Related Work

**Bug characteristic studies**  A lot of work has been done to study the bug characteristics in large software systems. Many of them provide precious information to help improve software reliability from different aspects, such as bug detection [26,30], fault tolerance [38], failure recovery [33], fault prediction and testing [39], etc. In a recent work [37], people also studied how the recent trends (availability of commercial tools, open-source, etc) affect the general bug characteristics (bug distribution, fixing time) for all bugs.

Unfortunately, concurrency bugs have not been studied extensively, probably because real world concurrency bugs are hard to collect and analyze. For example, in a previous study [33], only 12 concurrency bugs were collected from three applications: MySQL, GNOME and Apache. Under this situation, a previous concurrency bug pattern study [32] had to ask students to purposely write concurrent programs containing bugs, which cannot well represent the real world bug characteristics. Unlike previous work, we study the bug pattern, manifestation, and fix of 105 real world concurrency bugs from 4 large open source applications. Our study provides many findings and implications for addressing the correctness problems in concurrent programming.

**Improving concurrent program reliability**  Techniques to improve the quality of concurrent programs are related to our work. Due to space limit, here we briefly discuss the work that has not been discussed in previous sections.

In software testing, people proposed different coverage criteria in order to selectively test concurrent program interleavings. Unfortunately, these proposals are either too complicated [13] or based on heuristics [14, 15]. Our study of concurrency bug manifestation can help understand the trade-off between testing complexity and bug exposing capability and help design better coverage criteria.

In programming language area, designs other than transactional memory are also studied. AtomicSet [40] associates synchronization constraints with data instead of code region. This
design can help avoid some multiple variable related concurrency bugs. Autolocker [41] eases programmers specifying atomic regions by automatically assigning locks. Our characteristics study provides more motivation for these new language features.

2.8 Chapter Summary

This chapter provides a comprehensive study of the real world concurrency bugs. It summarizes bug patterns, manifestation conditions, common bug fix strategies and other characteristics. Our study is based on 105 real world concurrency bugs, randomly collected from 4 representative open-source programs: MySQL, Apache, Mozilla, and OpenOffice. The result of our study includes many interesting findings and implications for concurrency bug detection, testing and concurrent programming language design.

Among all the presented findings, some motivates our bug diagnosis tool presented in Chapter 3. More specifically, our tool can find bug-triggering conditions of both atomicity-violation and order-violation (Finding (1) and (2)). The tool can find the conditions quickly since bug triggers are usually related to a small number of threads, resources and accesses (Finding (3), (6) and (8)). More details about the tool are described in Chapter 3.
Chapter 3

Finding Complex Bug-Triggering Conditions

Previous chapter concentrates on the characteristics of concurrency bugs in various open-source programs. The presented findings include

- Atomicity-violation and order-violation are the most common bugs. While atomicity violation has been studied extensively, order-violation has not been studied much. (Finding (1) and (2))

- For bug manifestation, only a small number of threads, resources and accesses are necessary. (Finding (3), (6) and (8))

Motivated by the above findings, this chapter presents a tool that finds complex bug-triggering conditions.\(^1\) The tool targets distributed programs where non-deterministic message delivery is the source of race conditions\(^2\), which has not been studied much by previous research.

3.1 Overview

Non-deterministic message delivery makes it hard to debug concurrency bugs in distributed programs. To diagnose such non-deterministic bugs, programmers need to know the bug-triggering conditions. A causal order among multiple distributed events over multiple processes constitutes a directed acyclic graph (DAG). Hence, to diagnose the bug, a bug-triggering event DAG needs to be identified.

\(^1\)Part of the work presented in this chapter was done in collaboration and was published earlier [42].
\(^2\)With simple straightforward modifications, it can also be used for multi-threaded programs as well.
All events (such as message transmission and reception) that occur during distributed program execution can be logged and represented as a large DAG. It can be obtained by combining the sequential event logs of all processes using sequence numbers of corresponding transmitted/received messages. The transmission event and reception event of the same message at different processes have a causal relation, so there is an edge between them. In this chapter, we present a graph-mining tool, PopMine, which finds bug-triggering conditions (represented as event DAGs) that characterize the bug-triggered executions from the successful ones.

Our focus on the automatic diagnosis of software bugs is not new. Prior work developed diagnostic tools that identify bug-triggering conditions automatically. These conditions included bug triggering events, groups of events, or event sequences. Researchers have studied finding bug-triggering control flow within a process [43], finding faulty software components [44], and finding bug-triggering sequences of events in networked systems [45], to name a few.

In this chapter, we extend the scope of automatic diagnosis to the class of bugs which are triggered by the more complex execution conditions represented by a DAG of events. This chapter is the first in identifying event graphs correlated with occurrences of problems. We believe that the extension is of great value in distributed systems where execution constitutes a distributed event graph, not a sequential order. Case studies and performance results confirm that PopMine can expose the minimal bug-triggering conditions that cannot be found by previous approaches.

3.2 Definitions

In this chapter, we consider three types of events: message transmission (t), message reception (r) and local event (l). Each event is expressed as a vertex in an execution graph and it is labeled with two attributes: event type (t, r, l) and event attribute (details about
transmitted/received message or local event). An event label $tA$, for example, means a transmission event of message $A$.

An execution event graph is a directed acyclic graph of the events of all processes that occur during a distributed program execution. Figure 3.1a shows an example of an execution event graph. Events that occur at the same process are positioned chronologically along a vertical line (earliest event is at the top). Process names can be shown on top of the lines for the ease of explanation. A pair of events with a direct causal relation (transmission and reception of the same message) is connected by a directed edge. Edges are not labeled since message information is embedded in the event labels. The execution event graph in Figure 3.1a has four processes and 10 unique event labels ($lX$, $lY$, $\cdots$, and $rD$).

The ultimate goal of PopMine is to find the bug-triggering event DAG pattern. For example, consider a message race condition, induced by two transmission events without any causal relation from two senders to the same receiver. The outcome of the race condition is the order of the two corresponding reception events. Figure 3.1b shows a race condition and one of the two possible outcomes. Depending on the outcome (whether $rA$ happens before $rB$ or $rB$ happens before $rA$), the bug may manifest or not.

The bug-triggering condition can be captured in a rooted event DAG pattern, in which the root event happens after all other events in the pattern. The process at which the root event happens is called the root process. The root event can be considered as the result
(that is, bug manifestation) and all other events (happening before the root event) in the pattern can be considered as its possible causes. From now on, we’ll use the term “pattern” to refer to a rooted event DAG pattern.

A pattern has zero or more instances in an execution graph. For example, the pattern $p$ in Figure 3.1b has one instance (marked by shaded events) in the execution graph in Figure 3.1a. For a “subset” of an execution graph to be an instance of a pattern, it needs to be isomorphic to the pattern. Corresponding events, of course, must have the same labels. Note that, events in a pattern instance do not have to be consecutive. In the example in Figure 3.1a, $rA$ and $rB$ of $p$’s instance are separated by an event $tC$, which does not belong to $p$’s instance.

We require that each instance cannot skip the same events that are included in the instance at each process. At process $P_2$, for example, $tC$ is skipped and $rA$ and $rB$ are included in the instance. Since they have nothing in common, shaded events form a valid instance.

### 3.3 Finding a bug-triggering event pattern

Input of the proposed mining algorithm is a number of labeled (good or bad\textsuperscript{3}) execution graphs. They can be obtained from multiple execution traces. Using the labeled execution graphs, PopMine finds the bug-triggering conditions in the form of event DAGs as presented in this section.

#### 3.3.1 How likely is a given pattern a bug-trigger?

To quantify the likelihood that a given event pattern is a bug-trigger, we use information gain as a discriminative measure as other data mining algorithms. Information gain, $IG(Y|X)$,\textsuperscript{3}

\textsuperscript{3} “Good” execution means that the program execution is successful and no bug is triggered. “Bad” execution means that a bug is triggered.
represents the reduction of uncertainty about $Y$ when $X$ is known. It is defined as

$$IG(Y|X) = H(Y) - H(Y|X)$$

where $Y$ is the label (good or bad) of the execution graph and $X$ represents whether the number of instances of the given pattern is zero or not. $H(Y)$ is the entropy of the events that the bug is triggered or not. $H(Y|X)$ is the conditional entropy of $Y$ when $X$ is known.

Suppose that the numbers of good/bad execution graphs which have zero/non-zero instances of a pattern $p$ are in Table 3.1a.

<table>
<thead>
<tr>
<th>Instances</th>
<th>0</th>
<th>&gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>Bad</td>
<td>$c$</td>
<td>$d$</td>
</tr>
</tbody>
</table>

(a) $p$

<table>
<thead>
<tr>
<th>Instances</th>
<th>0</th>
<th>&gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>$a+\alpha$</td>
<td>$b-\alpha$</td>
</tr>
<tr>
<td>Bad</td>
<td>$c+\beta$</td>
<td>$d-\beta$</td>
</tr>
</tbody>
</table>

(b) $q$, a super pattern of $p$

Table 3.1: Number of good/bad execution graphs with zero/non-zero instances

Then $H(Y) = -u \log u - (1-u) \log(1-u)$ where $u = P(Y = \text{Good}) = (a+b)/(a+b+c+d)$, and $H(Y|X) = P(X = 0) \cdot H(Y|X = 0) + P(X > 0) \cdot H(Y|X > 0)$ where

$$P(X = 0) = (a+c)/(a+b+c+d)$$

$$P(X > 0) = 1 - P(X = 0)$$

$$H(Y|X = 0) = -v \log v - (1-v) \log(1-v)$$

$$v = P(Y = \text{Good}|X = 0) = a/(a+c)$$

$$H(Y|X > 0) = -w \log w - (1-w) \log(1-w)$$

$$w = P(Y = \text{Good}|X > 0) = b/(b+d).$$

As shown above, information gain of a pattern is a function of four parameters that describe the number of good/bad execution graphs with zero/non-zero instances.

Since it is highly likely that the bug trigger is in the bad execution graphs and not in
the good execution graphs, it can be exposed by finding the event patterns with the largest information gain. Due to the symmetry of information gain, the patterns that are in the good execution graphs and not in the bad execution graphs are also found.

3.3.2 Pattern Score

There may exist multiple patterns with the maximum information gain. Inspired by Occam’s razor, PopMine generally prefers smaller patterns (with fewer events) than larger patterns (with more events). For example, if pattern $A_1$ and $A_2$ in Figure 3.2 have the maximum information gain, PopMine chooses $A_2$ since it has fewer events.

However, there is an exception when PopMine does not choose the smallest pattern. Suppose pattern $A_1$ and $A_3$ in Figure 3.2 have the maximum information gain. Note that $A_3$ contains two reception events ($rA$ and $rC$) which do not have matched transmission events. We call a pattern with one or more unmatched receptions an incomplete pattern. PopMine favors complete patterns over incomplete ones since complete ones may help developers to understand the bug more.

Due to the above reasons, we define a metric, called Pattern Score, used as a tiebreaker among patterns with the same information gain. It is given by $-N_e - 2 \cdot N_r$ where $N_e$ is the number of events and $N_r$ is the number of unmatched reception events. With this metric, $A_1, A_2$ and $A_3$ in Figure 3.2 have pattern scores of $-6$, $-4$ and $-8$, respectively. Therefore, if all three patterns have the same maximum information gain, PopMine selects $A_2$. If only $A_1$ and $A_3$ have the maximum information gain, PopMine selects $A_1$.

In summary, PopMine chooses small/complete patterns over large/incomplete patterns using the pattern score among the patterns with the same information gain.
3.3.3 Mining Algorithm

Our mining algorithm starts from singleton patterns (patterns which are composed of only one event) and grows them by prepending events repeatedly. To reduce the search space, we use the branch-and-bound search technique (Section 3.3.4 and 3.3.5) and canonical pattern growth (Section 3.3.5).

Algorithm 1 Mining Algorithm

1: \( E \leftarrow \) All unique labels of all events
2: \( S \leftarrow \{\} \) \( \triangleright \) Patterns to search
3: \textbf{for all } \( p \in E \) \textbf{do} \( \triangleright \) \( p \) is a singleton pattern
4: \( I_g \leftarrow \) Find all instances of \( p \) in good execution graphs
5: \( I_b \leftarrow \) Find all instances of \( p \) in bad execution graphs
6: \( \text{Add } (p, I_g, I_b) \text{ to } S \)
7: \textbf{end for}
8: \( F \leftarrow \{\} \) \( \triangleright \) Found patterns
9: \( g \leftarrow 0, s \leftarrow 0 \)
10: \textbf{while } \( S \) is not empty \textbf{do} \( \triangleright \) Info. Gain and Pattern Score of \( F \)
11: \( \text{Pop } (p, I_g, I_b) \text{ from } S \)
12: \( (F, g, s) \leftarrow \text{PopMine}(F, g, s, p, I_g, I_b) \)
13: \textbf{end while}
14: \textbf{Return } \( F \)

In the beginning of Algorithm 1, all unique event labels are obtained (line 1). Each unique event label forms a singleton pattern and all its instances are found (line 4 and 5). This is a simple linear search. These events in the initial patterns become the root events of searched patterns.

During the search, \( F \) stores best patterns (judged by the information gain and pattern score) among those that have been searched so far. For each singleton pattern \( p \), PopMine(\)

Figure 3.2: Pattern Score
is called to search all patterns that can be grown from $p$. Whenever PopMine() returns, it updates $F$, $g$ and $s$ and they are fed again to the next call of PopMine().

Algorithm 2 PopMine($F$, $g$, $s$, $p$, $I_g$, $I_b$)

1: $p_g \leftarrow p$’s Information Gain (calculated from $I_g$ and $I_b$)
2: $p_s \leftarrow p$’s Information Score
3: if ($p_g > g$) or ($p_g = g$ and $p_s > s$) then
4:     $F' \leftarrow \{p\}$, $g' \leftarrow p_g$, $s' \leftarrow p_s$
5: else if ($p_g = g$ and $p_s = s$) then
6:     $F' \leftarrow F \cup \{p\}$, $g' \leftarrow p_g$, $s' \leftarrow p_s$
7: else
8:     $F' \leftarrow F$, $g' \leftarrow g$, $s' \leftarrow s$
9: end if
10: if number of events in $p < \text{MAX\_EVENTS}$ then
11:     if InfoGainUpperBound($p$) $\geq g'$ then \hspace{1cm} \text{▷ Sec. 3.3.4}
12:         $C \leftarrow$ Generate $p$’s child patterns \hspace{1cm} \text{▷ Sec. 3.3.5}
13:     for all $c \in C$ do
14:         $(I_g^c, I_b^c) \leftarrow$ Find instances of $c$ using $I_g$ and $I_b$
15:         $(F', g', s') \leftarrow$ PopMine($F'$, $g'$, $s'$, $c$, $I_g^c$, $I_b^c$)
16:     end for
17: end if
18: end if
19: Return ($F'$, $g'$, $s'$)

Algorithm 2 shows the PopMine() function. For each pattern $p$ to search, we first check whether $p$ is better than the patterns found so far by comparing information gain and information score. If $p$ is better, $p$ becomes the so-far best pattern. If $p$ is same as the patterns in $F$, $p$ is added to $F$.

To prevent patterns from growing indefinitely, we limit the number of events in grown patterns to $\text{MAX\_EVENTS}$.$^4$ If $p$ does not reach the limit, and upper bound of information gain of $p$ and its descendants is larger than or equal to that of the found patterns, $p$’s child patterns are generated, their instances are found, and PopMine() is called recursively.

$^4$In our code, we set $\text{MAX\_EVENTS}$ to 10.
3.3.4 Bounding

Suppose \( p \) is a pattern and \( q \) is a super pattern of \( p \), meaning that \( q \) contains \( p \). Since every instance of \( q \) contains an instance of \( p \), the number of instances of \( q \) in an execution graph is always smaller than or equal to the number of instances of \( p \). Therefore, the number of good/bad execution graphs with zero/non-zero instances of \( p \) and \( q \) can be expressed as Table 3.1a and 3.1b with non-negative integers \( \alpha \) and \( \beta \).

With different values of \( \alpha \) and \( \beta \), information gain is maximized when 1. non-zero instance always means good execution graph and zero instance mostly means bad execution 2. non-zero instance always means bad execution graph and zero instance mostly means good execution. Therefore, among all possible values of \( \alpha \) and \( \beta \), the ones which maximize information gain are \((\alpha = b, \beta = 0) \) or \((\alpha = 0, \beta = d) \). We calculate both cases and take the maximum as the upper bound of the information gain among all super patterns of \( p \).

3.3.5 Branching

Prepending an Event

When growing a pattern by adding a new event, PopMine only prepends the new event, which means that it does not happen after any existing event.

Figure 3.3 shows the four different methods for prepending a new event. Events with solid lines are in the current pattern. Four events circled with dotted lines represent four different ways to prepend a new event. First, a transmission (\( tD \)) from an existing process can be added. This new transmission event is matched with one existing reception event (unicast) or multiple ones (broadcast/multicast). Second, a new reception (\( rA \)) can be prepended in the beginning of some existing process. At this time, the corresponding transmission has not been determined. Third, a local event (\( lE \)) from an existing process can be added. Fourth, a transmission (\( tC \)) from a new process can be added. In this case, the newly added event becomes the last event of the newly added process.
After prepending a new event to a pattern, instances of the new pattern should be found. This can be done incrementally from the instances of the old pattern. For example, for \textit{rA} in Figure 3.3, we start from \textit{rD}'s instance and search upwards until \textit{rA} is found. The instance of \textit{tC} can be found directly from \textit{rC}'s instance. While adding new events to an instance, we make sure that the skipped events and the events included in the instance do not overlap as explained in Section 3.2. If they overlap, the instance is discarded.

**Canonical Process Order**

Two patterns are \textit{equivalent} if they can be made the same by swapping processes. For example, Figure 3.4 shows three equivalent patterns. In general, a pattern with \( n \) processes can have \( n! \) equivalent patterns. Although they look different, they all carry the same information. Therefore, there is no good in finding all such patterns. It is sufficient to find just one of them. To do that, we define \textit{canonical process order} denoted by \(<\).

From each process in an execution graph, only the last events of all processes, their
adjacent edges and their matched events are considered to get a last-event tree. For example, Figure 3.5 shows the last-event tree of an execution graph. Shaded events denote the last events of processes. Root event of the execution graph becomes the root of the last-event tree.

Then, the last-event tree is traversed in depth-first manner. Numbers next to the events in Figure 3.5 are the order of traversal. It starts depth-first traversal from the root event. At the third event (at $P_3$), there are two choices: another event at the same process ($P_3$) or an event at a different process ($P_4$). At such events, different process is chosen first, so the order in Figure 3.5 is obtained.

Finally, from the order of event traversal, the order of last event in all processes is used. In the example, last events are 1st, 2nd, 4th, 6th, 8th and 10th events. Corresponding process order is $P_6 \prec P_3 \prec P_4 \prec P_5 \prec P_2 \prec P_1$.

The patterns in Figure 3.4 have the canonical process order of $P_2 \prec P_3 \prec P_1$. We enforce all patterns to have left-to-right process order of the canonical process order. Therefore, among the three patterns in Figure 3.4, PopMine allows the one in the middle only.

**Canonical Pattern Growth**

Note that generating all possible patterns by prepending a new event can lead to redundant searches. Figure 3.6 shows that there can be two ways (left and right) to grow the given pattern with three events. If both ways are allowed, the pattern will be searched twice.
More complex patterns may have much higher redundancy. To prevent this, PopMine uses the *canonical event order*.

Canonical event order (denoted by \(<\)) gives a total order of all events in a given pattern as follows: 1. if event \(e_1\) happens before \(e_2\), \(e_2 < e_1\), 2. if there is no happen-before relation between \(e_1\) (happening at process \(P_1\)) and \(e_2\) (happening at process \(P_2\)), \(e_1 < e_2\) if \(P_1 < P_2\).

Figure 3.7a shows the canonical order of all events in a pattern. Since the canonical order is \(rA < rB < tA\), they are marked by 1, 2 and 3, respectively.

Figure 3.6 shows that there are many ways to grow a pattern by prepending events. Among all the same patterns that are grown through different ways, only one should be allowed. For a particular pattern growth to be canonical, the order in which events have
been added to the pattern and the canonical event order should be same. Figure 3.7b shows the order of event additions for the right boxed pattern in Figure 3.6. Note that, Figure 3.7a and 3.7b are different, meaning that it is not canonical, and right boxed pattern in Figure 3.6 is pruned. On the other hand, left boxed pattern is canonical, so it is not pruned.

This pruning technique using canonical pattern growth guarantees that any pattern is searched only once and no pattern is missed. Due to space limitation, we omit the proof.

3.4 Case Studies

Our implementation of PopMine consists of around 3400 lines of C++ code. PopMine reads execution graphs (described in a simple text format) from successful runs and failed runs and finds the bug-triggering pattern.

To generate execution graphs, we instrumented the original programs to log events locally. These logs are collected and combined together using message sequence numbers.

3.4.1 Virtual Cord Protocol (VCP)

Protocol

VCP [46] is a routing protocol for wireless sensor networks that maintains a virtual ordered list of all nodes. Each node maintains a predecessor node pointer and a successor node pointer which are neighbors of the node. Nodes in the linked list from the start node to the end node have ascending position numbers. A position number is a floating-point number in [0, 1].

Figure 3.8 shows how a VCP network changes when nodes join the network. Nodes’ position numbers are also shown. Arrows represent successor links (predecessor links are not shown) and dotted lines represent neighbor relations. Each node periodically broadcasts Hello messages that contain its position number and predecessor/successor information. When a data packet is routed, each node forwards the packet to one of its neighbors closest
to the destination based on the position numbers. For example, when node $D$(Pos:0) in Figure 3.8d sends a message to node $C$(Pos:1), it is first forwarded to $B$ which is its only neighbor, then $B$ forwards it to $A$ since $A$ has the largest position number less than the destination position number among all the neighbors. Then $A$ forwards the packet to $C$, which is the final destination.

![Diagram](image)

(a) Before $D$ joins  
(b) Before $E$ joins  
(c) Before $F$ joins  
(d) Final Network

Figure 3.8: VCP examples

Position numbers are assigned when nodes join and may be re-assigned when other nodes join. Figure 3.8a and 3.8b show the position number assignment before/after node $D$ joins. To maintain the global list structure, $D$ assigns its position number to 0 and reassigns $B$’s position number to the number in the middle of $B$’s original position number and $A$’s ($B$’s successor) position number. Then $D$ sets its successor to $B$.

Nodes can also join between two nodes. Figure 3.8b and 3.8c show the position number assignment before/after node $E$ joins. Since $E$ has two neighbors ($B$ and $A$) which are consecutive (meaning that $B$ and $A$ are each other’s predecessor/successor), $E$ can join between $B$ and $A$. To do that, $E$ assigns its position number to middle number of $B$’s and
A’s position numbers. E also sets its predecessor/successor to B/A. Then it updates B and A’s successor and predecessor pointers to E.

Figure 3.9 shows the pseudo code for one case of VCP join operation, described in the original chapter. When node D joins in Figure 3.8a, this pseudo code is executed. Other cases (joining between two nodes, etc.) are similar.

```
MyPosition←0
Successor←Neighbor
Predecessor←NULL
if NeighborSuccessor=NULL then
  NewNeighborPosition←1.0
else
  NewNeighborPosition←position(0, NeighborSuccessorPosition)
endif
SendUpdatePredecessor(Neighbor, NewNeighborPosition)
```

Figure 3.9: VCP Join Operation

Implementation

We implemented VCP on TinyOS using TOSSIM with meyer-heavy noise model. Nine message types (Hello, UpdatePred, UpdatePredACK, UpdateSucc, UpdateSuccACK, CreateVirtual, CreateVirtualACK, Data and DataACK) are defined. UpdatePred, UpdateSucc, CreateVirtual and Data messages are transmitted reliably using ACK and retransmissions.

Observations

We first generated a topology of 7 nodes. One node is initially joined to the network and starts sending Hello messages periodically. Period for Hello messages is 100 ms. Each not-yet-joined node with a joined neighbor starts its join operations after a random delay.

After executing the program multiple times, we observed that, in most cases, a global list of nodes is successfully maintained as in Figure 3.8. However, we also observed that, in some cases, the list structure is broken as Figure 3.10.
Diagnosis of the VCP bug

We executed the VCP program repeatedly until we got at least 10 successful runs and at least 10 failed runs. In a log, each node has around 400 events. With the collected labeled execution graphs, we executed PopMine. At first, the found results turned out to be false positives. This is because multiple different bugs were triggered in the bad executions.

Executions with incorrect results can be categorized by the symptoms. For example, some networks have multiple start nodes (Figure 3.10a). We repeated program executions until we get at least 10 runs in which multiple start nodes are observed. Then, we executed PopMine again.

Figure 3.11a shows the bug triggering pattern exposed by PopMine. It shows that, if node A receives two UpdatePred messages and sends a Hello message, a bug is triggered. When the first UpdatePred is received by $N_1$, $N_3$ becomes the predecessor of $N_1$. At this time, node $N_2$ sends an UpdatePred to node $N_1$. Between the two receptions of UpdatePred, $N_1$ do not send any HELLO message. Therefore, $N_2$ is not aware of $N_1$'s new position number and $N_1$'s new predecessor $N_3$. As a result, both $N_2$ and $N_3$ have the same ID and have node $N_1$ as their successor. This violates the VCP network model, which maintains a global list of nodes with unique position numbers.

After repeating the same experiment, we found that, some execution graphs which are from the executions generating multiple start nodes do not contain any instance of the
pattern in Figure 3.11a. We collected such execution graphs and re-executed PopMine on them only. This exposed another bug-triggering pattern in Figure 3.11b. In this pattern, $N_1$ sends a Hello message between the two reception events of UpdatePred messages. However, this Hello message is received by node 2 after node 2 sends the UpdatePred message, this pattern still triggers the bug.

Note that, by sequence-mining using the sequences of events occurring at each node, the bug-triggering pattern in Figure 3.11b can hardly be diagnosed since the three sequences of events (Rx: UpdatePred, Tx: Hello, Rx: UpdatePred, Rx: Hello) (Tx: UpdatePred, Rx: Hello, Tx: Hello) and (Tx: UpdatePred) are frequently observed in the normal runs. Section 3.5 explains more about the compared sequence-mining approach.

3.4.2 Chord

Chord [47] is a DHT protocol that maintains a virtual ring among all joined nodes. Each node has an ID and all nodes are placed along the virtual ring in ascending order of node IDs. Each node also keeps its successor node in the virtual ring. Nodes maintain other information (e.g., finger table) as well, but we omit such details for brevity. We used PopMine to diagnose a non-deterministic bug that we injected in our implementation of Chord.

When a node $m$ joins to the ring, it first finds its predecessor node $p$ after which $m$ will be placed. For this, $m$ sends a FIND_PRED message to a known node that is already in the
Chord ring. That message is routed through the nodes in the Chord ring and finally arrives at $p$. Then $p$ sends a message FIND_PRED_RSP back to $m$. Then $m$ gets the successor node $s$ of $p$ through another message exchange (GET_NEXT and GET_NEXT_RSP). After that, $m$ sets its successor to $s$ and sets $m$’s successor to $m$ itself by sending and receiving SET_NEXT and SET_NEXT_RSP to/from $m$.

This can handle individual joins well, but cannot handle concurrent joins since two concurrent joins may break the ring structure. Therefore, each node in the Chord ring does periodic stabilization in which a node checks whether the node’s successor’s predecessor is itself or not. If not, it fixes the successor pointer.

![Diagram](image)

Figure 3.12: Bug-trigger in Chord (injected bug)

We injected a bug into a Chord implementation by removing stabilization. As expected, concurrent join operations could not be processed properly. Using five execution graphs in which there is a missing node in the Chord ring and five good execution graphs, we executed PopMine and found the pattern depicted in Figure 3.12. It clearly shows that, when a node’s join operation (GET_NEXT and SET_NEXT) is interleaved by another node’s join operation, the bug is triggered.

### 3.4.3 GreenGPS

Recently published GreenGPS [48] is another example showing that PopMine can be used for diagnosing problems in distributed systems. GreenGPS leverages built-in sensors in cars to collect data and share it to build a community database over time that helps understand
vehicular fuel consumption. The OBD-II (On-Board Diagnostic) interface provides a mechanism to collect different engine parameters such as speed, instantaneous fuel consumption and engine RPM.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Year</th>
<th>Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Honda Civic</td>
<td>2002</td>
<td>1.7L</td>
</tr>
<tr>
<td>2</td>
<td>Chevy Prizm</td>
<td>1998</td>
<td>1.8L</td>
</tr>
<tr>
<td>3</td>
<td>Mazda 626</td>
<td>2001</td>
<td>2.0L</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>17</td>
<td>Ford Torus</td>
<td>2001</td>
<td>3.0L</td>
</tr>
</tbody>
</table>

Figure 3.13: Cars Used in the Modeling

To build a model that predicts fuel consumption, GreepnGPS used measured data from many cars as shown in Figure 3.13. Several models of fuel consumption were built depending on the year, make and engine size. While most models showed high accuracy, there were a few inaccurate models.

Figure 3.14: GreepnGPS: A Bad Model (Year 2000–2004)

For each accurate or inaccurate model, we first built a graph that shows which nodes (cars) contributed data to model calculation. Figure 3.14, for example, shows the one bad model based on the cars in 2000–2004. In the graph, $N_0$ represents the data processing module. Each pair of $t$ and $r$ and the connecting arrow shows which car data is used in the model. For example, an arrow from $tM_1$ at $N_1$ to $rM_1$ at $N_0$ shows that data from a Honda...
Civic 2002 with 1.7L engine (ID: 1) is used in the model. After generating all good/bad model graphs, we applied PopMine to find the cause of the bug and found that the two patterns exist in all bad model graphs and in no good model graphs. They are patterns with node 10 and 12 that represent Toyota Prius 2004 with 1.5L engine and Toyota Celica 2000 with 1.8L engine. Therefore, the results mean that, if any of those two cars are used in the model, the model is not trustworthy.

Further manual examination identified that the OBD interface in the Toyota Prius was actually reporting car speed in two different units; in mile/hour and in km/hour alternatively. This caused the prediction model, which expected consistent units, to malfunction. The other car, Toyota Celica, was turned out to be a false positive.

### 3.5 Performance Evaluation

To the author’s knowledge, PopMine is the first tool that considers bug-triggering conditions in the form of event graphs. Therefore, we compare PopMine and other mining approaches that consider bug-triggering conditions in the form of event sequences.

For the experiments, we used a PC equipped with a dual-core Intel Pentium 4 3.00GHz processor and 2GB of RAM and running Linux 2.6.18.

#### 3.5.1 Synthetic Graph Data

For more detailed evaluation of the performance of the proposed algorithm, we developed a synthetic data generator. To mimic the behavior of a real distributed program, we first define an *operation* as a set of events that are associated by some basic function in a protocol. For example, “Join” in VCP is an operation which triggers a set of related events. The synthetic data generator first generates a number of random operations as shown in the left part of Figure 3.15.

Second, when two uncoordinated operations are interleaved in a specific way, we assume
that a bug is triggered. The synthetic data generator makes a *mix* by picking two operations and interleaving them randomly. After producing a number of mixes, it marks only one mix as bad. All other mixes represent “safe” interleavings. This is shown in the middle of Figure 3.15.

Third, an execution graph is composed of many mixes. It is generated by concatenating a number of mixes. If an execution graph contains a bad mix, the bug is triggered and the execution graph is marked as bad. Otherwise, it is marked as good. This is depicted in the right part of the Figure 3.15.

Using the good/bad execution graphs, we execute PopMine and examine the found patterns. Note that PopMine does not have to find the whole bad mix since its subset may be enough to identify the bad execution graphs. If a found pattern is contained in the bad mix, we consider that it is a true bug trigger. If not, we assume that it is not a bug trigger.
3.5.2 Synthetic Sequence Data

To use sequence-mining algorithms, generated graphs need to be converted to sequences. An execution graph can be converted to many event sequences by grouping events by the nodes at which the events occur. The graph in Figure 3.1a, for example, is converted to four sequences: \((tA, tA, 1Y), (tA, tC, rB)\), etc. This way, good/bad graphs are converted to good/bad sequences.

To compare with PopMine, we present results of two approaches based on sequence-mining. First, DustMiner [45] is a bug diagnosis tool based on discriminative sequence-mining with several extensions. Second, RPMiner [49] is a sequence-mining tool that finds repetitive gapped subsequences from a set of sequences. We first find frequent subsequences from bad sequences and filter out the ones which are also frequent in the good sequences. Since the found sequences are frequent in bad sequences and infrequent in good sequences, they are likely bug-triggers. We call this approach Seq.

To determine whether the found sequence is a true bug trigger or not, we take a similar approach to what we did for PopMine. We first convert the bad mix into sequences. If a found subsequence is contained in any of the sequences of the bad mix, we consider that it is a true bug trigger. If not, we assume that it is not a bug trigger.

3.5.3 Experiment 1 – Varying Number of Execution Graphs

In the first experiment, the numbers of good/bad execution graphs are changed while all other parameters are kept fixed. We keep the two numbers the same. The parameters we used are summarized in Exp. 1 of Table 3.2. According to the parameters, an execution graph contains 180 events \((12 \text{ events/mix} \times 15 \text{ mixes/run})\). For each data point, we generated 20 test cases, executed mining algorithms and calculated the average result.

Figure 3.16 shows the average execution time. As the number of execution graphs in-
<table>
<thead>
<tr>
<th>Name</th>
<th>Exp. 1</th>
<th>Exp. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of unique local events</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Number of unique messages</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Maximum number of nodes in an operation</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Number of events in an operation</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Number of unique operations</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Maximum number of nodes in a mix</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Number of good mixes</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td><strong>Number of mixes in an execution graph</strong></td>
<td>15</td>
<td>Varies</td>
</tr>
<tr>
<td>Maximum number of nodes in an execution graph</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>Number of good execution graphs</strong></td>
<td>Varies</td>
<td>10</td>
</tr>
<tr>
<td><strong>Number of bad execution graphs</strong></td>
<td>Varies</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.2: Parameters for the synthetic data generation

Figure 3.16: Experiment 1: Execution Time (log scale)

creases, overall execution time also increases. However, PopMine uses many methods to prune the search space and finish search quickly.

The average number of found patterns is plotted in Figure 3.17. Note that, in a real usage scenario, the programmer needs to check the found patterns manually to extract the real bug triggers. PopMine returns only a manageable number of patterns so the programmer can check them in a short time. In contrast, DustMiner and SEQ return too many sequences.

Note that not all found patterns are real bug-triggers. Figure 3.18 shows the ratio of non-bug triggers among the found patterns. To interpret this figure, Figure 3.17 needs to be considered as well. For example, when the number of good/bad execution graphs is 5,
PopMine finds 2.85 patterns on average (Fig 3.17). About 56% (Fig 3.18) of them are not bug triggers. Overall, PopMine shows low false positive rate. In contrast, many sequences found by DustMiner and SEQ are not bug triggers. This is caused by the large number of found patterns by them as shown in Figure 3.17. Only a small number of them are real bug-triggers.

Figure 3.19 shows the fraction of test cases in which the true bug triggering condition cannot be found. When the number of execution graphs is small, PopMine may miss bug triggers since there may exist other patterns that are, by chance, discriminative and small. However, as more execution graphs are added to the mining process, the odds of such a
coincidence are reduced. Note that, even though SEQ finds a large number of discriminative sequences, it cannot find real bug triggers in more than half of the test cases. This is because, as shown in Section 3.4.1, subsequences of bug-triggering graph patterns may be frequent in good runs. In that case, it is impossible for sequence mining-based approaches to find the bug-trigger since no event sequence is discriminative. In other words, the true bug trigger cannot be expressed as a sequence of events at a node, but rather as a distributed event graph.

3.5.4 Experiment 2 – Varying Number of Mixes in Execution Graphs

In this experiment, we vary the number of mixes in execution graphs. By increasing the number of mixes, the average number of events in a node increases. This represents the situation that the distributed program is executed and logged for a longer time. All other parameters are shown in Exp. 2 of Table 3.2. Since the number of mixes in an execution graph varies from 8 to 14, the number of total events in an execution graph varies from 96 (12 events/mix × 8 mixes) to 168 (12 events/mix × 14 mixes).

Figure 3.20 shows the average execution time. They grow exponentially over the size of
mixes, but PopMine shows less steep slope. Number of patterns found by SEQ also increase exponentially as the number of mixes increases (Figure 3.21). Therefore, it is not scalable to examine found sequences manually. However, PopMine returns only a small number of patterns, so that programmers can easily examine them. DustMiner returns a moderate number of results.

Figure 3.22 and 3.23 show the ratio of false bug triggers among the found ones and the ratio of test cases in which no true bug trigger is found. As in the previous experiment, PopMine shows reasonably small numbers. In contrast, DustMiner and SEQ returns many false bug triggers and they may fail to find any bug triggers.
Figure 3.22: Experiment 2: Ratio of False Bug Triggers among the found

Figure 3.23: Experiment 2: Ratio of tests that no true bug trigger is found

### 3.6 Related Work

Debugging distributed programs is a hard task. To deal with the non-determinism caused by network conditions, deterministic replay tools \[50–52\] are used to log non-deterministic events occurring in the program execution. While replay tools can reproduce bugs, programmers still need to diagnose the bug to understand how the bug is triggered in the replay runs.

Instead of relying on replay debuggers, many systems focus on logging useful information during runtime that may not be sufficient for replaying program executions, but still car-
ries valuable information for debugging. With PIP [53], the programmer first specifies the expectation of various properties of the system. During the execution, PIP logs the actual behavior of the program and helps programmer to check the difference of expectation and the actual behavior. X-trace [54] constructs task trees, which summarizes how a request is processed by all layers at all nodes. If there is a faulty layer or node, it can be observed in the task tree.

Monitoring tools can also detect and diagnose simple problems in the network. Pinpoint [44], for example, finds faulty components in distributed systems. For each request, all components that are involved in the request processing are logged. Pinpoint finds the components that are related to the failed requests. Sympathy [55] checks several predefined metric data and finds problems of node, network path or sink in a sensor network. \(D^3S\) [56] does predicate checking on a deployed distributed systems. While these monitoring tools may tell whether the system is in a good state and what is the faulty component if the system is not working correctly, they do not expose what conditions trigger failures.

Researchers have proposed to use data mining techniques to understand the behavior of software for various purposes such as failure detection [57], modeling program behavior [49, 58]. While these techniques can be effective for the software on single system, they do not model networked systems well. DustMiner [45] uses a sequence-mining technique to find a subsequence of events that may be the bug-triggering events. PopMine extract information from more complex structure: partially ordered event over many nodes.

Statistical debugging [59, 60] uses correlation of program’s behavior and failures to find the root cause of bugs. Rather than message race bugs, they are targeting bugs within a single program. To reduce the logging overhead, they use statistical sampling. PopMine, which requires each process to provide full message logs, can be made more light-weight with the statistical sampling techniques.

Many researchers have also studied performance analysis of MPI-based message passing parallel programs by analyzing execution traces [61, 62]. PopMine does trace analysis for a
different purpose: bug diagnosis.

Dynamic bug detection techniques [63–65] such as PopMine analyze program execution for any buggy behavior. It means that, if a bug is not triggered, it cannot be detected. Due to the execution overhead, it may not be possible to have a large number of test executions enough to reveal all bugs.

Static tools [66–70], on the other hand, find bugs without actual executions. They analyze source code to find any potential problem. It generally covers more execution space than dynamic bug detection tools. However, they require access to source code, that is not always available. Dynamic tools, including PopMine, can be used to diagnose bugs even when the source code is not available.

3.7 Chapter Summary

Diagnosis of non-deterministic distributed bugs in networked systems is a challenge. In this chapter, we propose a new automatic diagnosis tool that can pinpoint bug-triggering event DAGs embedded within much larger execution graphs. This is a significant extension to previous work, which is concerned with finding bug triggering events, group of events, or event sequences, but not event DAGs, hence presenting possible deficiencies in identifying complex protocol bugs in distributed systems. Since our algorithm uses causal order among events, it does not require a global clock.

Our experience shows that (i) the minimal bug-triggering event DAG can be found quickly, (ii) false positive rates of results are low, and (iii) even when each operation is composed of a large number of events, the bug-triggering event DAG which characterizes the bug can be much smaller than the operations. The chapter constitutes an advance to the types of bugs that can be diagnosed in distributed systems such as various protocol implementations or MPI programs.
Chapter 4

Root-Cause Identification in Error-Propagating Networks: Exploiting Component Graphs

Previous two chapters focus on the micro-scale root-cause diagnosis of software bugs related to race conditions. In chapter 4 and 5, we explore issues in more macro-scale root-cause diagnosis in error-propagating networks.¹

4.1 Error-Propagating Networks

In this chapter, we consider the problem of finding root-cause of propagating errors in a network. When a component introduces an error to the network, the error may propagate to other components. Examples of such networks and the problems of root-cause identification are as follows.

Finding Buggy Software Module  Large software is composed of many modules that have complex inter-dependency. A software bug in a module can produce incorrect values in computation that are fed to other modules, which may also generate incorrect values. When many modules produce bad computation results due to a bug in one module, how can the buggy module be found?

Finding Broken Sensor  If a sensor in a sensor network is broken, it may produce bad measurement data. Since its measurements are fed to other computation units in the network, many computation units can produce bad intermediate/final results. Then, how can the broken sensor be identified?

¹Part of the work presented in this chapter was done in collaboration and was published earlier [71].
**Finding Initial Spreader of Computer Virus**  Computer virus or Internet worms use security vulnerabilities in software to propagate themselves from one computer to other computers. In a network with many infected computers, how can the initial virus spreader be found?

**Finding Rumor Source in Social Networks** Social networks are popular media for sharing information. Online social networks enable large-scale information dissemination in a very short time, often not matched by traditional media [72,73]. Mis-information and false claims can also propagate rapidly through social networks. This is exacerbated by the fact that (i) anyone can publish (incorrect) information and (ii) it is hard to tell who the original source of the information is [74]. Without the provenance information of error propagation, how can the initial rumor spreader be found?

In this chapter, an error-propagating network as the above examples is modeled as a directed graph $G = (V, E)$ where $V$ is the set of all network components (or nodes) and $E$ is the set of edges where each edge represents the direction of error propagation between two nodes. An error, started from a node (called error source or root-cause node), propagates to other nodes. The nodes at which propagated errors arrive are called infected nodes. We assume that $k$ pre-selected nodes $M$ ($M \subseteq V$) are monitors, which have self-diagnostic logic and reports whether they are infected or not. We denote the set of infected monitor by $M^+$, and the set of uninfected monitor by $M^-$ (where $M^+, M^- \subseteq M$). For non-monitor nodes, it is impossible to know whether they are infected or not.

### 4.2 Root-Cause Identification

The first question we are studying in this chapter is as follows: if an error-propagation is started by a single root-cause component in a network, how can it be identified?
4.2.1 Method

For this problem, we use the intuition that the root-cause node must be close to the infected monitors but far from the uninfected monitors. Hence, for each node $x$, our algorithm calculates the following four metrics:

(1) **Reachability to all infected monitors**  We first calculate how many infected monitors are reachable from $x$. For a node $x$ to be the root-cause node of the error propagation, $x$ must have paths to all monitors in $M^+$. If those paths do not exist, $x$ cannot be the root-cause node.

(2) **Distance to infected monitors**  Among those nodes that can reach all infected monitors, nodes that are closer, on average, are preferred. In other words, for each node $x$, we calculate the total distance

$$\sum_{m \in M^+ \text{ and } m \text{ is reachable from } x} d(x, m).$$

where $d(x, m)$ is the distance from $x$ to $m$, and sort the suspected sources by increasing total distance from infected monitors.

(3) **Reachability to uninfected monitors**  Among the nodes that can reach all nodes in $M^+$ and have the same total distance to these infected monitors, we use reachability to uninfected monitors as a third metric. For each such node $x$, we count of monitors in $M^-$ that are not reachable from $x$ and prefer larger counts.

(4) **Distance to uninfected monitors**  As a last metric, we also use the distance to uninfected monitors. For each node $x$, we calculate the total distance

$$\sum_{m \in M^- \text{ and } m \text{ is reachable from } x} d(x, m).$$
It is more natural that uninfected monitors are far from the root-cause node, so nodes with large values of total distance are preferred.

Using the above four metrics – number of reachable infected monitors, sum of distances to reachable infected monitors, number of reachable uninfected monitors, sum of distances to reachable uninfected monitors –, all nodes in the network are sorted lexicographically. That is, \( i \)-th metric is used only when there is a tie in all metrics before it. Note that, for the first and last metric, large numbers are preferred while small numbers are preferred for the second and third. Our implementation converts the sign of first and fourth metrics to make sorting easy.

In the sorted list, the top suspect is the first node.

### 4.2.2 Monitor Selection

For best accuracy, it is important to choose monitors wisely. In this chapter, we compare the following six monitor selection methods.

1. **Random**  
   Random selection method selects \( k \) monitors randomly. This means that, for any node \( x \in V \), the probability that \( x \) is selected as a monitor is \( \frac{k}{|V|} \).

2. **Inter-Monitor Distance (Dist)**  
   This requires any pair of monitors to be at least \( d \) hops away. To do that, it first randomly shuffles the list of all nodes. Then, from the first node, it checks whether it is at least \( d \) hops away from all already-selected monitors. If it is, the node is selected as a monitor and the next node is checked. Note that the first node is always selected as a monitor. This is repeated until \( k \) monitors are selected or it is impossible to select any more monitors. Dist selection method finds the largest \( d \) which can choose \( k \) monitors. To do that, it starts with a large value of \( d \) and decrements it every time it fails to choose \( k \) monitors, starting over with the smaller \( d \) until it can find \( k \) monitors.
(3) **Number of Incoming Edges (NI)** In this method, the number of incoming edges of each node is counted. Then, the top $k$ nodes that have largest counts are chosen.

(4) **NI+Dist** This method combines NI and Dist. Nodes with a large number of incoming edges are preferred as monitors, but the algorithm also considers inter-monitor distance. To do that, it first sorts all nodes in the descending order by the number of incoming edges of each node, then Dist is used to choose monitors. In other words, nodes in the sorted list are examined one by one and a node is chosen as a monitor if it is at least $d$ hops away from all previously selected monitors. As in Dist, this method finds the largest $d$ that can choose $k$ monitors.

(5) **Betweenness Centrality (BC)** This method calculates betweenness centrality [75] for each node $v$, which is defined as

$$C(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where $\sigma_{st}$ is the number of shortest paths from $s$ to $t$ and $\sigma_{st}(v)$ is the number of shortest paths from $s$ to $t$ that pass through $v$. Then, the $k$ nodes that have the largest betweenness centrality are chosen as monitors.

(6) **BC+Dist** This method combines BC and Dist. Nodes are first sorted by their betweenness centrality, then Dist is used to choose monitors.

The above six monitor selection algorithms produce different sets of monitors, which result in different accuracy in the root-cause node identification. Section 4.2.3 compares these algorithms in detail.
4.2.3 Case Study

In this case study, we apply our root-cause node identification method in a social network to find the initial spreader (source) of a rumor.

Dataset

To apply our algorithm to a real social network, we extracted a graph from Twitter. First, we obtained 159271 tweets written in Dec. 2011 containing a special keyword\(^2\). These tweets were written by 39567 twitter accounts.

In Twitter, tweets are propagated by retweets. When a user \(y\) retweets another user \(x\)’s tweet (or retweet), we assume that there is an edge from \(x\) to \(y\). In total, we obtained 102796 edges from the crawled data. The undirected version of this graph has 9243 connected components. In this evaluation, we focus on the largest connected component \(G\) that has 30146 nodes and 102608 edges. Maximum in-degree and out-degree among all nodes in \(G\) is 193 and 2264, respectively. The undirected version of \(G\) has a diameter of 12 hops.

Besides the topology, we also calculated Retweet probability of each edge \(x \rightarrow y\) as the ratio of “\(x\)’s tweets retweeted by \(y\)” to “all tweets of \(x\).” Calculated Retweet probabilities were used to simulate random propagation of rumors.

Finding the Root-Cause Node

We first evaluate the accuracy of our root-cause identification method. To simulate random error propagation, we do the following: (1) A random error source is selected, (2) Using propagation probability of each edge, the error is propagated, (3) if an error does not reach more than 1% of all nodes, it is consider it as negligible error and discard the result. Then a new error source is selected and the same procedures are repeated. For each simulation, we use our root-cause identification algorithm using different number of monitors:

\(^2\)The actual used keyword is “Kim, Geuntae (in Korean),” a Korean politician who died in Dec. 2011. Instead of using Twitter API directly, we downloaded already-crawled tweets from Tweetrend.com. It is a third-party twitter web site which shows the trend of popular keywords and the actual tweets in Korean.
20, 40, 80, \cdots, 5120. All results are averaged over 200 simulations.

**Rank of the Actual Source** Using the method presented in Section 4.2, all nodes are sorted in the likelihood that they are the actual error source. Figure 4.1 shows the average rank of the actual source in the output. In the ideal case, the rank should be one that means that the top suspect is actually the error source. Note that, regardless of the monitor selection method, the rank of the true source generally decreases (i.e., improves by becoming closer to 1) as the number of monitors increases. Dist and NI+Dist generally show a bad accuracy. Random also performs poorly when the number of monitors is small, but it improves as more monitors are added. NI, BC and BC+Dist show better performance than the others. When the number of monitors is very large, the choice of monitor selection does not matter that much anymore, and all algorithms converge.

![Figure 4.1: Average Rank of Actual Source in the output (out of 30146 nodes)](image)

One of the important factors that affect the accuracy of error source identification is the number of infected monitors. Figure 4.2 shows the ratio of experiments in which no monitor received the rumor. In all monitor selection methods, the ratio decreases as the number of monitors increases. Among the four methods compared, the Dist selection method has the highest ratio. Dist basically maximizes inter-monitor distance, so it tends to choose nodes on the boundary of the graph. Therefore, monitors selected by Dist have low probability of
hearing rumors. The Random selection method also has a high ratio of uninfected monitors when the number of monitors is small. The other methods (NI, NI+Dist, BC, BC+Dist) have small ratio compared to Dist and Random. When no monitor hears the rumor, it is very hard to find the source accurately as shown in Figure 4.1 (Random and Dist when the number of monitors is 20, for example).

<table>
<thead>
<tr>
<th>Number of Monitors</th>
<th>Random</th>
<th>NI</th>
<th>Dist</th>
<th>NI+Dist</th>
<th>BC</th>
<th>BC+Dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.6850</td>
<td>0.5200</td>
<td>0.3050</td>
<td>0.1200</td>
<td>0.0150</td>
<td>0.0000</td>
</tr>
<tr>
<td>40</td>
<td>0.0050</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>80</td>
<td>0.0600</td>
<td>0.0050</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>160</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>320</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>640</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1280</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>2560</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>5120</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

This figure shows the ratio of experiments (out of 200 experiments) that no sensor hears the rumor. In random selection, it decreases as the number of sensors increases.

However, a larger number of infected monitors do not always lead to a more accurate result. Figure 4.3 shows the average number of infected monitors when the number of monitors is 160. Figure 4.1 shows that BC has best accuracy, NI has second best, and the others are worse. Note that, the number of infected monitors in NI is almost double of that in BC, but BC is more accurate. This means that the accuracy of source finding algorithm does not increase monotonically with the number of infected monitors. Also, if the number of infected monitors is very low (as in BC+Dist, NI+Dist, Random and Dist), source-finding algorithm outputs very inaccurate results.

**Distance from Top Suspect to the True Source** Figure 4.4 shows the distance\(^3\) between the top suspect and the actual source. In the ideal case, the distance should be zero,

\(^3\)This distance is calculated over the undirected version of the graph.
Figure 4.3: Number of Infected Monitors (Number of Monitors: 160)

meaning the the top suspect is the source. Figure 4.4 shows a similar tendency as Figure 4.1. The distance decreases as more monitors are added. Dist shows the largest distance of all monitor selection methods. Random has large distances with a small number of monitors, but the distance decreases drastically as the number of monitors increase. BC and BC+Dist generally show the smallest distance between the top suspect and the actual source.

Figure 4.4: Average Distance between the top suspect and the actual source
4.3 Estimating the Number of Error-Initiating Nodes

The other problem we are studying in this chapter is estimation of the number of error-initiating nodes. Though it is impossible to estimate the exact number of error-initiating nodes, it may be good enough to classify whether there are a large number of error-initiating nodes or just a small number of them in some cases.

For example, in social networks, if one person initiates a rumor intentionally, it is not independently corroborated by others. Hence, in the absence of collusion, there is only one source of the error propagation in the network. If a rumor is initiated by a small colluding group of people, the number of independent sources is just the size of the group. Conversely, if a piece of information is not a rumor, there may be many independent sources of the information. Therefore, it is important to estimate the number of independent sources correctly.

For the classification, we calculate the following two metrics: GSSS and MDGIP.

4.3.1 Greedy Sources Set Size (GSSS)

A set of nodes $C$ is a valid source set if the following is satisfied: for all $m \in M^+$, there exists $n \in C$ which satisfies $d(n, m) \neq \infty$. The question is, what is the minimum size of set $C$? Instead of calculating the exact solution with an exponential algorithm\(^4\), we use the greedy approximation algorithm in Algorithm 3 to get an approximate minimal source set. The algorithm calculates the set of candidate sources ($C$). For each candidate source $x$, it also calculates the set of infected monitors ($P_x$) covered by $x$ that are used in Section 4.3.2.

Initially, $C$ and $P_m$ for all $m \in V$ are initialized to empty sets. At each iteration, a node $x$ which can reach the largest number of elements in $M^+$ is chosen as a source. Node $x$ is considered as one of the candidate sources and all monitors in $M^+$ that are reachable from $x$ are assumed to have received the information from $x$. Then $x$ is added to $C$. The reachable

\(^4\)For each node $m$ in $M^+$, we define $S_m$ the set of nodes which have a path to $m$. Then calculating the minimal $C$ is exactly same as the minimal hitting set problem among $\{S_m\}_{m \in H}$.
Algorithm 3 A Greedy algorithm for calculating an approximate minimal source set.

\[
C \leftarrow \emptyset
\]
For each \( m \in V \), \( P_m \leftarrow \emptyset \).
For each \( m \in M^+ \), \( S_m \leftarrow \) the set of nodes which have a path to \( m \).
\[
\text{while } M^+ \neq \emptyset \text{ do}
\]
Let \( x \) be one of the most frequent elements in all \( S_m \)'s where \( m \in M^+ \).
Add \( x \) to \( C \).
For each \( m \in M^+ \), add \( m \) to \( P_x \) and remove \( m \) from \( M^+ \) if \( d(x, m) \neq \infty \).
\end{while}

monitors are removed from \( M^+ \) and put into \( P_x \). In the final state, \( C \) becomes the Greedy Source Set (GSS), which is an approximate minimal source set. For each node \( x \in C \), \( P_x \) is the set of monitors that are expected to receive the information from \( x \).

![Diagram of Greedy Source Selection](image)

Figure 4.5: Greedy source selection overestimates error propagation distance.

4.3.2 Maximal Distance of Greedy Information Propagation (MDGIP)

The previous greedy algorithm tries to assign as many infected monitors as possible to each source, so the resulting greedy information propagation trees tend to become larger than the real ones. Figure 4.5 shows an example with three original source nodes (black circles). Information from the sources propagates along the solid arrows. Note that, the dotted edges are not used for the actual error propagation. Suppose all nodes (black or white) are monitors. The greedy approach in Section 4.3.1 finds that it is possible to cover all infected monitors using only one source node. This means that \( C = \{1\} \) and \( P_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \).
As a result, a new large propagation tree is generated instead of the actual three small trees.

To estimate possible disparity between the actual propagation tree and the one constructed by the above greedy algorithm, we use a second metric. Namely, given a greedy source set $C$ and the set of nodes that receive information from $x (P_x)$ for all $x \in C$, we define Maximal Distance of Greedy Information Propagation (MDGIP), calculated as:

$$\max_{x \in C, y \in P_x} d(x, y)$$

where $d(x, y)$ is the distance from $x$ to $y$. Note that, when there are many actual sources, the estimated MDGIP tends to become large since many small propagation trees are combined into one greedy propagation tree.

Previous two metrics – Greedy Source Set Size (GSSS) and Maximal Distance of Greedy Information Propagation (MDGIP) – increase as the number of actual sources increases. In the following section, we present a case study on rumor classification in social networks.

### 4.3.3 Case Study

In this case study, we use the dataset used in Section 4.2.3 and classify rumors from non-rumors in social networks. Rumors are usually initiated by a small number of people. In contrast, true information can be reported by many people independently. In this case study, we assumed that rumors have a small number (1 or 10) of sources and non-rumors have a large number (100 or 1000) of sources and show how their propagation features are different.

For each number of sources (1, 10, 100 and 1000), monitor selection method, and number of monitors (20, 40, 80, 160, 320 and 640), we repeated rumor identification 200 times and used the results for rumor identification only when there is at least one monitor that receives the rumor.
Figure 4.6 shows average GSSS and MDGIP when Dist monitor selection method is used. Left figure shows that, as the number of real sources increases, GSSS also increases. Right figure also shows that, as the number of real sources increases, MDGIP also increases. These two graphs show that GSSS and MDGIP can be used to classify a piece of information as a rumor or not since it is directly related to the number of sources.

**Figure 4.6: GSSS and MDGIP (Monitor Selection: Dist)**

Figure 4.7 shows the results from Random monitor selection. Overall, Figure 4.7 looks similar to Figure 4.6, but the difference of GSSS and MDGIP values with different number of sources is smaller.

**Figure 4.7: GSSS and MDGIP (Monitor Selection: Random)**

**Random** Figure 4.7 shows the results from Random monitor section. Overall, Figure 4.7 looks similar to Figure 4.6, but the difference of GSSS and MDGIP values with different number of sources is smaller.

**NI** Figure 4.8 shows the results from NI monitor selection. Contrary to the previous two monitor selection methods, it shows that GSSS does not change over different number of
monitors or different number of sources. In the experiments, GSSS is alway one, which means that only one candidate source can cover all infected monitors. It also shows that MDGIP and the number of sources do not have strictly consistent relation. Overall, GSSS and MDGIP for the NI monitor selection algorithm do not give much information for rumor identification.

![Figure 4.8: GSSS and MDGIP (Monitor Selection: NI)](image)

**NI+Dist** Figure 4.9 shows the results from NI+Dist monitor selection. It shows the overall tendency that GSSS and MDGIP increase with the number of sources.

![Figure 4.9: GSSS and MDGIP (Monitor Selection: NI+Dist)](image)

**BC** Figure 4.10 shows the results from BC monitor selection. Similar to NI, GSSS does not change and MDGIP shows inconsistent values with different number of sources.
Figure 4.10: GSSS and MDGIP (Monitor Selection: BC)

**BC+Dist** Figure 4.11 shows the results from BC+Dist monitor selection. Similar to Dist, Random and NI+Dist, it shows the overall tendency that GSSS and MDGIP increase with the number of sources.

Figure 4.11: GSSS and MDGIP (Monitor Selection: BC+Dist)

**Classification**

Previously, we have shown that GSSS and MDGIP can be used to classify a piece of information as rumor or non-rumor. Figure 4.12 visualizes GSSS/MDGIP and rumor classification in more detail. Four figures compare the cases in which rumors have 1 or 10 sources and non-rumors have 100 or 1000 sources when Dist monitor selection algorithm is used and the number of monitors is 640. When there is a large difference in the number of sources of rumors and non-rumors (two right figures), it is shown that rumors and non-rumors are
clearly separated. However, when the difference is smaller (two left figures), rumors and non-rumors overlap in some cases that causes inaccuracy in classification.

Using the experimental data, we evaluated how accurately logistic regression can classify rumor and non-rumors. We used the first half of experimental data as training set and the second half as test set. Table 4.1 summarizes the results when the number of monitors is 640. The first column of Table 4.1 shows the monitor selection algorithm. The second and third columns show the number of sources of a rumor and a non-rumor. Next three columns (4–6) show the classification results of 100 rumors. They are classified as rumors (True Positive, Column 4) or non-rumors (False Negative, Column 5). If no monitor hears the rumor, our classification algorithm cannot work, so it cannot be classified (Column 6). Similarly, last three columns (7–9) show the classification results of 100 non-rumors. They are classified as non-rumors (True Negative, Column 7) or rumors (False Positive, Column 8). If no monitor
Table 4.1: Classification of rumors and non-rumors (Number of Monitors: 640).

<table>
<thead>
<tr>
<th>Monitor Selection</th>
<th># Sources (Rumor)</th>
<th># Sources (Non-rumor)</th>
<th>Rumors</th>
<th>NonRumors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>True Positive</td>
<td>False Negative</td>
</tr>
<tr>
<td>Dist</td>
<td>1</td>
<td>100</td>
<td>70</td>
<td>0</td>
</tr>
<tr>
<td>Random</td>
<td>1</td>
<td>100</td>
<td>82</td>
<td>18</td>
</tr>
<tr>
<td>NI+Dist</td>
<td>1</td>
<td>100</td>
<td>84</td>
<td>13</td>
</tr>
<tr>
<td>NI</td>
<td>1</td>
<td>100</td>
<td>37</td>
<td>63</td>
</tr>
<tr>
<td>BC</td>
<td>1</td>
<td>100</td>
<td>51</td>
<td>49</td>
</tr>
<tr>
<td>BC+Dist</td>
<td>1</td>
<td>100</td>
<td>64</td>
<td>36</td>
</tr>
<tr>
<td>Dist</td>
<td>1</td>
<td>1000</td>
<td>70</td>
<td>0</td>
</tr>
<tr>
<td>Random</td>
<td>1</td>
<td>1000</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>NI+Dist</td>
<td>1</td>
<td>1000</td>
<td>97</td>
<td>0</td>
</tr>
<tr>
<td>NI</td>
<td>1</td>
<td>1000</td>
<td>37</td>
<td>63</td>
</tr>
<tr>
<td>BC</td>
<td>1</td>
<td>1000</td>
<td>67</td>
<td>33</td>
</tr>
<tr>
<td>BC+Dist</td>
<td>1</td>
<td>1000</td>
<td>87</td>
<td>13</td>
</tr>
<tr>
<td>Dist</td>
<td>10</td>
<td>100</td>
<td>68</td>
<td>10</td>
</tr>
<tr>
<td>Random</td>
<td>10</td>
<td>100</td>
<td>81</td>
<td>19</td>
</tr>
<tr>
<td>NI+Dist</td>
<td>10</td>
<td>100</td>
<td>91</td>
<td>6</td>
</tr>
<tr>
<td>NI</td>
<td>10</td>
<td>100</td>
<td>34</td>
<td>66</td>
</tr>
<tr>
<td>BC</td>
<td>10</td>
<td>100</td>
<td>54</td>
<td>46</td>
</tr>
<tr>
<td>BC+Dist</td>
<td>10</td>
<td>100</td>
<td>51</td>
<td>49</td>
</tr>
<tr>
<td>Dist</td>
<td>10</td>
<td>1000</td>
<td>78</td>
<td>0</td>
</tr>
<tr>
<td>Random</td>
<td>10</td>
<td>1000</td>
<td>99</td>
<td>1</td>
</tr>
<tr>
<td>NI+Dist</td>
<td>10</td>
<td>1000</td>
<td>97</td>
<td>0</td>
</tr>
<tr>
<td>NI</td>
<td>10</td>
<td>1000</td>
<td>34</td>
<td>66</td>
</tr>
<tr>
<td>BC</td>
<td>10</td>
<td>1000</td>
<td>67</td>
<td>33</td>
</tr>
<tr>
<td>BC+Dist</td>
<td>10</td>
<td>1000</td>
<td>89</td>
<td>11</td>
</tr>
</tbody>
</table>
hears anything, it cannot be classified (Column 9).

From the table, we can observe that, if rumors and non-rumors have very large difference in the number of sources, rumor classification can be done with very high accuracy. As the difference in the number of sources of rumors and non-rumors decreases, it gets harder to classify rumors and non-rumors accurately.

Another observation from the table is that the algorithms which show good results in error source identification (BC and BC+Dist) do not always work well in rumor classification. This is because the two tasks have different conditions for best performance. In error source identification, it is best to have monitors near the error source, so that they receive the rumor and estimate the error source based on their locations. In rumor classification, it is best to have monitors in various places in the error propagation trees so that GSSS and MDGIP can be estimated accurately. We leave finding a monitor selection algorithm that is good for both tasks as future work.

4.4 Related Work

Finding Root Cause of Distributed System Errors

A number of tools [44, 54, 76, 77] trace causal path of request processing. They visualize request trees and show related statistics to support failure diagnosis. Component-based approaches [44, 78] tracks requests that are processed by various components in the system and collect successes/failures and their related components. Using statistical analysis, it correlates components with failures to find the root cause.

Sherlock [79] builds an inference graph that describes the propagation of errors. When the actual errors occur, it finds the most probable error sources using the states of network services accessed by observation node and the inference graph. While Sherlock has an approach that is similar to ours, there are some differences. First, Sherlock assumes that there is no loop in the inference graphs and errors propagate from parent nodes to child
nodes. Therefore, error propagation probabilities can be calculated easily. In real situations, some nodes may have circular dependency. In our model (Section 4 and 5), we allow cycles, but use approximations for error propagation. Second, the authors states that the accuracy of Sherlock system is “affected by the number of vantage points” (in our terms, monitors), but do not study how to place vantage points strategically. There is a trade-off between the overhead of vantage points and the accuracy of the final result. For this problem, our work can help to choose them. NetMedic [80] is another tool that considers error propagation, but assumes that all nodes are monitors.

To detect re-occurrences of previously diagnosed problems, Cohen et al. uses a statistical approach to metric selection [81]. They build signatures from various reference metrics such as CPU/memory utilization.

In some tools, anomaly is considered as a sign of errors. For example, Kasick et al. proposed a fault localization tool [82] for the Parallel Virtual File System (PVFS) that uses CPU instruction-pointer samples and function-call traces to detect anomalous behaviors of culprit servers. Tan et al. developed a tool [83] that analyzes Hadoop logs, extracts state-machine views in MapReduce behavior, and creates a unified view of MapReduce program behavior. It focuses on the slowest causal flow to find the causes of the slow processing. With this method, problems in MapReduce processing steps or problems in specific hosts can be found.

**Rumors in Social Networks**

Shah et al. studied the problem of rumor source finding [84]. They model rumor spreading with a variant of an SIR model [85] and define rumor centrality, the number of ways (order or infections) that a virus can spread, to evaluate the likelihood that each node is the actual rumor source. This work has the same goal as our work, but there are some significant differences. First, they assume that the virus graph is known, meaning that the set of all infected nodes and the edges through which rumors are transferred are known. Second, it
is also assumed that all edges are equivalent. In contrast, we assume that the states of only a small subset of nodes are known. We also assume that we do not know the edges used for rumor spreading. Lastly, we assume that edges are different (they can have different propagation probabilities). Various notions of network centrality has been proposed [86–88] but it has not been studied how accurate they can be as an rumor source estimator.

Online social networks have emerged as a new medium for information sharing [72, 89]. Contrary to the traditional media, anyone can share information and it can be delivered to a large number of people in a very short time [73]. Unfortunately, it also carries undesirable information such as rumors [90].

A way to avoid rumors is to subscribe only trustworthy information sources. Adler and Alfaro proposed a content-driven reputation system for Wikipedia authors [91]. For each preserved edit or rollback of a user’s edit, reputation is adjusted. Zhao et al. also proposed SocialWiki in which social context including each user’s interest and trust is used to select trustworthy contributors [92] and TrustWiki in which conflicts are resolved by matching compatible editors and readers [93]. Canini et al. proposed a system that ranks users using topical content and social network structure [94]. Nel et al. tackled the problem of rumor detection by monitoring publishing behavior of information sources [74]. They cluster groups of sources that have similar publishing behaviors. Our approach uses social graph topology and monitors to detect rumors and it is orthogonal to user reputation systems.

Morris et al. studied how people feel about the credibility of new tweets [95]. They showed that people use various heuristics to assess the credibility – whether the tweet is retweeted, author’s expertise, etc. This can be used to improve the quality of Twitter search engine.

Mendoza et al. studied tweets about 2010 earthquake in Chile and found that rumors and non-rumors are retweeted in a different way [96]. That is, people question more about rumors and affirm more about non-rumors when they retweet. By making use of the comments of users in retweets, they have shown that it is possible to classify rumors and non-rumors.
Castillo et al. used a machine leaning technique that makes use of text in tweets, user characteristics and tweet propagation pattern to classify rumors and non-rumors [97]. Our methods could enhance such leaning techniques by exploiting social network graph structure even when the investigator has a limited view on the rumor propagation. Ratkiewicz et all developed a tool that visualizes tweet propagation and can be used to detect abusive behaviors [98]. This tool assumes that full provenance about information propagation is known, which is not used by our method. In another work of the authors, they focus on features related to information diffusion (e.g., number of nodes, number of edges, max number of in/out edges, etc.) and showed that truthiness classification is possible [99]. Gupta et al. focused on the credibility of events instead of individual tweets by a event graph-based optimization [100].

4.5 Chapter Summary

In this chapter, we proposed an approach for (i) finding the root-cause node of error propagation when there is only error source node and (ii) estimating the number of error sources when there can be many of them. Our approach uses a very small amount of provenance information; namely, which of a set of monitors are infected. To find the error source, our algorithm evaluates the likelihood of each node to be the source, calculated from node connectivity and shortest path distances. To estimate the number of error sources, we proposed two metrics – Greedy Source Set Size (GSSS) and Maximal Distance of Greedy Information Propagation (MDGIP) – and used logistic regression. To evaluate the proposed approach, we performed a case study involving a real social network crawled from Twitter. The algorithm shows good potential to help users in identifying error sources and estimating the number of error sources.
Chapter 5

Improving Root-Cause Identification in Error-Propagating Networks: Leveraging Additional Information

In this chapter, we extend our root-cause diagnosis presented in the previous chapter by leveraging additional information.¹

5.1 Leveraging Propagation Probability

First, we study how propagation probability – the probability that an error propagates through the given edge – can be used to improve root-cause identification in error-propagating networks.

5.1.1 Definitions

Network Graph

We model a error-propagating network as a directed graph $G = (V, E)$. Vertices in the graph $V = \{v_1, v_2, \cdots, v_n\}$ represent components in the network and directed edges $E = \{e_{ij} | e_{ij} \text{ is an edge from } v_i \text{ to } v_j\}$ represent directions of error propagation. Each edge $e_{ij}$ has a propagation probability $p_{ij}$ that reflects the probability that an error propagates from $v_i$ to $v_j$.

When a node $v_s$ fails and starts error propagation, it uses its outgoing edges to propagate the error to its neighbors. For each edge, error propagates with the associated propagation probability. This propagation is repeated until there is no more remaining edge that can propagate the error.

¹Part of the work presented in this chapter is submitted for publication [101].
Monitors

When an error propagates in a network, some nodes get infected and others do not. This is very useful information that gives hint for error source identification. However, in a very large network, it is not always possible to check all nodes if they are infected or not. Therefore, we assume that a set of nodes $M$ is pre-selected as monitors and they report whether they are infected or not. For a given error, $M$ is partitioned into two disjoint sets: infected monitors $M^+$ and uninfected monitors $M^-$ that have not. The sets of infected monitors and uninfected monitors $(M^+, M^-)$ are called an outcome of an error.

Outcome Probability

Given that a node $v_x$ is the error source, outcome probability $P(M^+, M^-|v_x)$ is the probability that all infected monitors receive the error and no uninfected monitors receive the error.

5.1.2 Problems

Finding Error Source

The problem of error source identification can be summarized as follows:

Given a graph $G = (V, E)$, a propagation probability function $p : E \to [0, 1]$ and an outcome of error propagation $(M^+, M^-)$ from pre-selected monitors $M$, find the node $v_x$ which maximizes $P(M^+, M^-|v_x)$.

For some special types of network topologies such as trees, this problem can be solved efficiently. However, for general graphs, the calculation of exact outcome probability, which leads to the optimal solution, has exponential time complexity.

Figure 5.1 shows two examples of graphs. In both graphs, we assume that node 1 is the source and we want to calculate the probability that node 4 receives the rumor. In
In the examples, we denote the probability that node $x$ is infected, but $y$ is not infected by $P(x^+y^-)$. We also denote the edge propagation probability from node $x$ to $y$ by $p_{xy}$.

Figure 5.1 (a) is a tree. For node 4 to get infected ($P(4^+)$), it should receive the error from node 2, so node 2 should be infected as well ($P(2^+4^+)$). Since node 1 is the source, nodes 1, 2 and 4 are infected ones ($P(1^+2^+4^+)$). It does not matter whether node 3 is infected or not. Therefore, $P(4^+)$ is calculated as follows:

\[
P(4^+) = P(2^+4^+) = P(1^+2^+4^+) = p_{12} \cdot p_{24}.
\]

Figure 5.1(b) is a complete graph of four nodes. Since node 4 can receive error from any of node 1, 2 and 3, $P(4^+)$ is calculated as follows:

\[
P(4^+) = P(1^+2^−3^−4^+) + P(1^+2^−3^+4^+) + P(1^+2^−3^−4^+) + P(1^+2^+3^+4^+)
\]

\[= \ldots\]

As the number of nodes increases, the number of terms in the above equation increases exponentially.

In this paper, instead of calculating the exact probability that has exponential time complexity for general graphs, we propose an efficient approximation algorithm that still
can find the error source precisely and efficiently.

5.1.3 Approximation 1: End-to-end Propagation Probability

Given a source node \( v_x \) and a monitor \( v_m \), there might be a large number of paths from \( v_x \) to \( v_m \). This makes computation of exact end-to-end propagation probability \( P(v_x \rightarrow v_m) \) expensive.

Instead of calculating the exact probability, we approximate the probability by taking a path that has the maximum propagation probability as follows:

\[
P'(v_x \rightarrow v_m) = \max_{t \text{ is a path from } v_x \text{ to } v_m} \prod_{e_{ij} \text{ is in } t} p_{ij}.
\]

Note that the path propagation probability \( \prod_{e_{ij} \text{ is in } t} p_{ij} \) is calculated by multiplying all edge propagation probabilities in the path. The above equation implies that errors are assumed to propagate through best paths (in terms of the path propagation probability) only. Since error can actually propagate through non-best paths, \( P'(v_x \rightarrow v_m) \) is only a lower bound of the exact error propagation probability \( P(v_x \rightarrow v_m) \).

To calculate the maximum path propagation probability between two nodes, we use a trick as follows. For each edge \( e_{ij} \), an edge weight \( w_{ij} \) is calculated as follows:

\[
w_{ij} = -\log_2 p_{ij},
\]

Since \( p_{ij} \) is in \([0, 1]\), \( w_{ij} \) is non-negative. Then, the path with maximum propagation probability from \( v_x \) to \( v_m \) is the shortest path from \( v_x \) to \( v_m \) with respect to the newly calculated edge weights \( w_{ij} \). The shortest path length from \( v_i \) to \( v_j \) is denoted as \( d_{ij} \). All shortest paths mentioned in this chapter are calculated over the edge weights \( w_{ij} \).
Approximation 2: Outcome Probability

Based on the previous approximation, we also approximate outcome probability $P(M^+, M^-|v_x)$, the probability that all infected monitors and no uninfected monitors receive the error from a given source $v_x$. To do that, we first calculate the shortest path tree $T$ from $v_x$ to all monitors in $M$ using Dijkstra’s algorithm.

Figure 5.2 shows four cases of Dijkstra’s shortest path trees that need to be considered. In the examples, $v_x$ is considered as error source and the root of the shortest path tree $T$. Nodes labeled as ‘+’ or ‘-’ are infected or uninfected monitors. Unlabeled nodes are non-monitors. For the ease of explanation, we mark some nodes as $a$, $b$ or $c$. Each edge is labeled with its propagation probability.

Figure 5.2a shows the most simple case. To make all infected monitors to receive the error, edges labeled with $p_1$, $p_2$ and $p_3$ need to carry the error. To make the uninfected monitor not to receive the error, the $p_4 - p_5$ path should not transfer the error. Therefore, its outcome probability is $p_1 \cdot p_2 \cdot p_3 \cdot (1 - p_4 \cdot p_5)$.

Figure 5.2b has two uninfected monitors. In this example, the fact that the internal uninfected monitor $a$ does not receive the error implies that the leaf uninfected monitor $b$ does not receive the error. Therefore, for the calculation of outcome probability, $p_3 - p_4$ part do not need to be considered. It make the outcome probability to be $p_1 \cdot (1 - p_2)$.

Figure 5.2c shows an example that has a infected monitor as a descendent of a uninfected monitor. Within this tree, it is impossible to make all infected monitors and no uninfected monitor to receive the error from the source $v_x$. However, we should now make the outcome probability to 0 because errors can actually propagate through edges that are not in the shortest path tree. We believe that, as more monitors are added, approximate outcome probability should become more accurate. Therefore, for the best approximation, in this case, such uninfected monitors ($a$) are not considered as monitors. This makes the approximate outcome probability to be $p_1 \cdot p_2 \cdot p_3 \cdot p_4$. 
Figure 5.2: Examples of Shortest Path Trees to Monitors
Figure 5.2d shows a corner-case that also needs to be considered, which was found while we debug our code. In this example, \( a \) not a monitor node, \( b \) is a uninfected monitor and there is an edge from \( a \) to \( b \) with propagation probability of 1.0. Since the propagation probability of \( a \rightarrow b \) edge \((p_3)\) is 1.0, the fact that \( b \) does not receive the error implies that \( a \) does not receive the error, either. However, \( c \) which is a child of \( a \) is a infected monitor that need to receive the error. Since these are contradictory, our original code calculated the approximate outcome probability as 0. Because of the same reasons as case 3, this is not desirable. In our updated method, the entire branch rooted at \( b \) is ignored. Therefore, the approximate outcome probability is \( p_1 \cdot p_2 \cdot p_4 \).

\[
\text{Algorithm 4 } f(T, v_s, M^+, N^-)
\]

1: Input \( T \): Shortest path tree from \( v_s \) to all monitors
2: Input \( v_s \): Root of error propagation
3: Input \( M^+ \) and \( M^- \): Positive/uninfected monitors
4: \( p \leftarrow 1 \triangleright \) Probability that all reachable infected monitors and no reachable uninfected monitors receive the error
5: \( n_+ \leftarrow 0 \triangleright \) # of reachable pos. monitors from \( v_s \).
6: \( n_- \leftarrow 0 \triangleright \) # of reachable neg. monitors from \( v_s \).
7: for each child \( v_c \) of \( s \) in \( T \) do
8: \( (p', n'_+, n'_-) \leftarrow f(T, v_c, M^+, M^-) \)
9: if \( p_{sc} = 1.0 \) and \( p' = 0 \) then
10: Ignore this branch. \( \triangleright \) Case 4
11: else
12: \( n_+ \leftarrow n_+ + n'_+ \)
13: \( n_- \leftarrow n_- + n'_- \)
14: if \( n'_+ > 0 \) then
15: \( p \leftarrow p \cdot p_{sc} \cdot p' \)
16: else if \( n'_- > 0 \) then
17: \( p \leftarrow p \cdot ((1 - p_{sc}) + p_{sc} \cdot p') \)
18: end if
19: end if
20: end for
21: if \( s \) is in \( M^+ \) then
22: \( n_+ \leftarrow n_+ + 1 \)
23: else if \( s \) is in \( M^- \) and \( n_+ = 0 \) then
24: \( p \leftarrow 0 \)
25: end if
26: return \((p, n_+, n_-)\)
Algorithm 5 Approximate Outcome Probability

1: Input: $T, v_x, M^+, N^-$
2: $(p, n_+, n_-) = f(T, v_x, M^+, M^-)$ \hspace{1cm} \triangleright Algorithm 4
3: if $n_+ = |M^+|$ then
4: \hspace{1cm} return $p$
5: else
6: \hspace{1cm} return 0
7: end if

Algorithm 4 and 5 summarizes the algorithms that calculate the approximate outcome probability. First, $f$ in algorithm 4 gets shortest path tree from $v_s$ ($T$), root of shortest path tree $v_s$, infected monitors $M^+$ and uninfected monitors $M^-$ and returns $(p, n_+, n_-)$ where $p$ is the probability that all reachable infected monitors receive the error and no reachable uninfected monitors receive the error, $n_+$ is the number of reachable infected monitors from $v_s$ and $n_-$ is the number of reachable uninfected monitors from $v_s$. For each child of $v_s$, it calls $f$ recursively and checks all cases mentioned above. In the final calculation of approximate outcome probability (Algorithm 5), it should be checked if all infected monitors are reachable. If not, the final probability should be zero.

5.1.5 Finding the Error Source

For each node $v_x$, we calculate the Dijkstra’s shortest path tree from $v_x$ and use it to calculate the outcome probability. Then, all nodes are sorted using the outcome probability in the descending order. The final output of the error source finding algorithm is this sorted list of nodes that has the top suspect in the beginning.

5.2 Leveraging Infection Timestamps

In the previous section, we assume that outcome probability is calculated when the network status converges (meaning that status of monitors does not change any more). In this section, we assume that infection timestamps describing when a node gets infected are reported by
the infected monitors. Therefore, it is possible to calculate outcome probability in realtime.

Suppose $v'_1, v'_2, \cdots, v'_r$ are infected monitors ($M^+$) in the order of infection. In other words, for $i < j$, $v'_i$ gets infected before $v'_j$. Let $M^-$ be the set of negative monitors. Whenever a new $i$-th monitor gets infected, the sets of infected monitors ($M^+_i$) and uninfected monitors ($M^-_i$) at that time are updated as shown in Table 5.1.

<table>
<thead>
<tr>
<th>Number of Infected Monitors</th>
<th>Infected Monitors ($M^+_i$)</th>
<th>Uninfected Monitors ($M^-_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$M^+_1 = {v'_1}$</td>
<td>$M^-_1 = M - M^+_1$</td>
</tr>
<tr>
<td>2</td>
<td>$M^+_2 = {v'_1, v'_2}$</td>
<td>$M^-_2 = M - M^+_2$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>$r-1$</td>
<td>$M^+_{r-1} = {v'_1, v'<em>2, \cdots, v'</em>{r-1}}$</td>
<td>$M^-<em>{r-1} = M - M^+</em>{r-1}$</td>
</tr>
<tr>
<td>$r$</td>
<td>$M^+_r = {v'_1, v'_2, \cdots, v'_r} = M^+$</td>
<td>$M^-_r = M - M^+ = M^-$</td>
</tr>
</tbody>
</table>

For example, when the first monitor gets infected, the set of infected monitor is $M^+_1 = \{v'_1\}$. At this time, the set of uninfected monitors is $M^-_1 = M - \{v'_1\}$. Similarly, when the $i$-th monitor gets infected ($1 \leq i \leq r$), the set of infected monitors is $M^+_i = \{v'_1, v'_2, \cdots, v'_i\}$ and the set of uninfected monitors is $M^-_i = M - M^+_i$.

Whenever $v'_i$ gets infected, we calculate the accumulated outcome probability for each node $v_x$, which is defined as

$$\prod_{k=1}^{i} P'(M^+_k, M^-_k | v_x).$$

This is used to locate error source even when the network status has not converged yet. Section 5.4.3 presents details of performance results of the accumulated outcome probability.

### 5.3 Selecting Monitors

For best accuracy in the error source finding, it is important to choose monitors wisely. From simulations, we observe that the number of infected nodes (individuals that receive the error) is much smaller than the number of negative nodes (individuals that do not receive
Table 5.2: Used Datasets

<table>
<thead>
<tr>
<th>Dataset No.</th>
<th>Domain</th>
<th>Number of Nodes</th>
<th>Number of Edges</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Social Network</td>
<td>13273</td>
<td>22837</td>
<td>Politician 1</td>
</tr>
<tr>
<td>2</td>
<td>Social Network</td>
<td>11413</td>
<td>26599</td>
<td>Politician 2</td>
</tr>
<tr>
<td>3</td>
<td>Software Dependency</td>
<td>308</td>
<td>1127</td>
<td>Gimp</td>
</tr>
<tr>
<td>4</td>
<td>Software Dependency</td>
<td>453</td>
<td>1663</td>
<td>VLC</td>
</tr>
</tbody>
</table>

the error). This means that, infected monitors carry more information that uninfected monitors. Therefore, our monitor selection algorithm aims to maximize the number of infected monitors.

Expected number of Received Errors (ERE) of node $v_x$ is defined as follows:

When each node propagates a error, how many errors can $v_x$ receive?

Since we assume that errors propagate through shortest paths only (Section 5.1.3), the expected number of errors that $v_x$ receives from $v_y$ (as the error source) is $P'(v_y \rightarrow v_x)$. Therefore, $ERE(v_x)$ can be calculated as follows:

$$ERE(v_x) = \sum_{v_y \in V} P'(v_y \rightarrow v_x)$$

To calculate $ERE(v_x)$ efficiently, we first reverse the direction of all edges in the graph. Then we use Dijkstra’s algorithm to calculate the shortest path lengths from $v_x$ to all other nodes. After that, end-to-end propagation probability from $v_x$ to each node is calculated as $(2^{-d})$ where $d$ is the shortest path length. Finally, all end-to-end propagation probabilities are summed.

To select $k$ monitors, we select the top $k$ nodes with the largest EREs.
Figure 5.3: Datasets
5.4 Evaluation

5.4.1 Datasets

To evaluate our algorithms, we use four graphs from two different domains as summarized in Table 5.2.

Twitter Data

First two datasets are extracted from crawled Twitter data. Instead of using Twitter API that has a rate limit, we downloaded already-crawled tweets containing special keywords from Tweetrend.com, which is a Korean third-party twitter web site that shows the trend of popular keywords and the actual tweets. For both datasets, we used famous Korean politicians’ names as search keyword and downloaded tweets written in Dec. 2011. We use retweet probability (probability that a person retweets another person’s tweet) as propagation probability of an edge. Graphs obtained from the raw tweets have a large number of separate connected components. For this evaluation, we select the largest connected component only. Number of nodes and edges are shown in Table 5.2.

Figure 5.3a and 5.3b show the (in-degree, out-degree) of all nodes in dataset 1 and 2. Most nodes are centered near the origin. We also observe that some nodes have large in-degrees or out-degrees and a few nodes have very large out-degrees.

Figure 5.3e and 5.3f show the distribution of propagation probability of all edges. It is observed that, as the propagation probability increases, frequency decreases in general. One notable exception is the last bucket [0.9, 1.0]. Most of these are the cases that users have only one tweet which happen to be a retweet. Therefore, the retweet probability is 1.0.

Software Module Dependency Data

The other two datasets are obtained from Ubuntu Linux. Using debtree [102] that shows all software packages that a given software package is related to, we obtain two graphs
related to Gimp (The GNU Image Manipulation Program) [103] and VLC (VideoLAN media player) [104]. In the graphs, there is an edge from package \( X \) to package \( Y \) if \( Y \) depends on \( X \). This means that, if \( X \) is buggy, it may produce incorrect calculation results that may be fed to \( Y \), which may lead to incorrect results at \( Y \) as well. Note that it is not easy to get propagation probabilities in these examples. To make it simple, we assign random numbers in \([0, 1]\) as propagation probabilities. Figure 5.3c and 5.3d show the (in-degree, out-degree) of all nodes in dataset 3 and 4.

### 5.4.2 Leveraging Propagation Probability

#### Error Propagation

Figure 5.4 shows the cumulative distribution of the number of infected nodes (nodes that receive the error) from many error spreading simulations \((y = P(\text{NumPositiveNodes} \leq x))\). For each figure, more than one million errors are spread from random sources.

Datasets from Twitter (Dataset 1 and 2 in Figure 5.4a and 5.4b) have very small number of infected nodes in most cases. In contrast, datasets from software module dependency (Dataset 3 and 4 in Figure 5.4c and Figure 5.4d) have relatively large ratio of infected nodes than the previous two datasets.

#### Effectiveness of ERE

ERE is developed to maximize the number of infected monitors. In this section, we compare ERE and two other methods: NI and BC. NI stands for Number of Incoming edges. As the name explains, the metric counts the number of incoming edges and the ones with large number of incoming edges are chosen as monitors. BC stands for betweenness centrality [105]. This method prefers the nodes in the center of the graph rather than the nodes in the border of the graph. NI and BC have been used in previous work for monitor selection [71].

Each sub-figure in Figure 5.5 shows the average number of infected monitors of each
monitor selection method as the ratio of monitors increases. It also limits the range of ratio of infected nodes (RPN). For example, figure 5.5a has a RPN range of [0.009, 0.012). This means that only the random error propagations that have RPN in that range are used to calculate the average number of infected monitors. Each data point in the figures is an average of 200 simulations.

Figure 5.5a and 5.5b show that NI and ERE yields similar number of infected monitor and they are better than BC in dataset 1 and 2. In figure 5.5c and 5.5d, ERE shows much larger number of infected monitors than the others. In all cases, ERE is the best choice to
maximize the number of infected monitors.

**Finding the Source**

We compare the proposed source-finding method (denoted as SPT) that makes use of edge propagation probability and a previous method (denoted as HOP, presented in Chapter 4). For each node $v_x$ in the network, HOP calculates four metrics: (1) number of reachable infected monitors ($n_+$), (2) sum of distances to reachable infected monitors ($d_+$), (3) number of reachable uninfected monitors ($n_-$) and (3) sum of distances to reachable uninfected monitors ($d_-$). Then all nodes are sorted lexicographically in the descending order of four tuples ($n_+, -(d_+), -(n_-), d_-$).

With the monitors chosen by NI and BC, we use HOP source finding method. With the proposed ERE, we use SPT. Figure 5.6 compares the rank of real source in the node lists returned by the three combinations (BC+HOP, NI+HOP and ERE+SPT) with the four
datasets.

In all four datasets in Figure 5.6, ERE+SPT shows the best results among the three combinations. Especially, in figure 5.6a, rank of real source found by ERE+SPT is an order of magnitude smaller than the others when the ratio of monitors is small. The difference generally decreases as the ratio of monitors increases.

Figure 5.7 also shows the accuracy of source finding by presenting the average error distance (hops) from the top suspect to the real source. Note that the hop distance is calculated in the undirected version of the graph. In contrast to our expectation, ERE+SPT is not always show the smallest error distance in all cases. We think that this result is less meaningful than the rank of real sources in figure 5.6 since it focuses on the top suspect only and ignores all others.
(a) Dataset 1 (0.009 ≤ RPN < 0.012)  (b) Dataset 2 (0.006 ≤ RPN < 0.009)
(c) Dataset 3 (0.05 ≤ RPN < 0.06)  (d) Dataset 4 (0.03 ≤ RPN < 0.04)

Figure 5.7: Distance from the Top Suspect to the Real Source (hops)

5.4.3 Leveraging Infection Timestamps

To evaluate the effectiveness of the method in Section 5.2 that leverages monitor infection timestamps, we simulate error propagation with time considered. At time 0, error source starts propagating errors. At each edge, error propagation is delayed by some random time modeled by an exponential random variable with rate $\lambda = 1$. This makes the mean delay 1 time unit.

In the simulation, whenever a new monitor gets infected, the accumulated outcome probability of each node is re-calculated. Every 0.5 time unit, the average rank of the actual error source from the most recent result is calculated, if any. Figure 5.8 (a), (b) and (c) show the rank of the actual error source that varies over time in dataset 3. Solid horizontal lines in the three figures show the rank of actual error source obtained from the method that does not use infection timestamp information (Section 5.1). Three figures have different ratio of
monitors. Figure 5.8 (a), which has the smallest ratio of monitors, show that infection timestamps are very useful information that improves the accuracy of error source identification especially when there are only small number of monitors. In contrast, Figure 5.8 (c) has the largest ratio of monitors, which makes the original method to produce quite good results (average rank of the actual source is less then 5). In this case, timestamp-based method cannot beat the original method. Figure 5.8 (b) shows a case in-between.

Figure 5.9 (a), (b) and (c) also show similar results in dataset 4.

5.5 Chapter Summary

In this chapter, we improve the source finding method in Chapter 4 by using additional information: propagation probability and infection timestamps. We propose efficient methods that calculate approximate end-to-end propagation probability and outcome probability for a given set of monitors. The approximate outcome probability is used to evaluate the likelihood that each node is the actual error source. We also describe a monitor selection method ERE that maximizes the number of infected monitors. With four graphs extracted from networks and software dependency data, we showed that it out-performs other previous algorithms (BC+HOP and NI+HOP) in most cases.

We also presented accumulated outcome probability that can be used to find error source through online analysis. As evaluation results show, it helps to improve the accuracy of error source identification especially when the number of monitors is small.
Figure 5.8: These three figures show the average rank of the actual error source with different ratio of monitors in dataset 3. In all simulations, the ratio of infected nodes in the network is between 0.02 and 0.03 and the average values are calculated over 100 simulations.
Figure 5.9: These three figures show the average rank of the actual error source with different ratio of monitors in dataset 4. In all simulations, the ratio of infected nodes in the network is between 0.02 and 0.03 and the average values are calculated over 100 simulations.
This dissertation has proposed new approaches on root-cause diagnosis for two types of notorious failures in distribute systems: software bugs caused by race conditions and propagating errors.

For the first problem, we started with a characteristic study on concurrency bugs. We chose 105 concurrency from the bug databases of four representative open-source software and examined the bug reports, fix attempts, discussion, and final bug patches. Through this process, we summarized bug patterns, manifestations, fix strategies and other characteristics. Our study revealed many interesting findings and also gave motivations to our bug-diagnosis tool.

With the findings from the characteristic study, we proposed a root-cause diagnosis tool that can find complex bug-triggering conditions in distributed systems. Our tool takes DAG of events from bug-triggered executions and successful executions and find the minimal pattern that characterizes the difference between the two groups of DAGs. Since the tool takes generic DAG of events, it can be easily extended for other types of programs such as multi-threaded programs.

For the problem of propagating errors, we proposed two different root-cause diagnosis tools. The first tool is based on a simple intuition that the failure source is close to the positive monitors and far from negative monitors. This leads to a node-sorting algorithm that uses four metrics: reachability to positive/negative monitors and sum of distances to reachable positive/negative monitors. We also compared six different monitor selection methods.
To improve the accuracy of root-cause diagnosis tools, our next tool leverages additional information: propagation probabilities of edges and infection timestamps. Since the calculation of exact end-to-end propagation probability and outcome probability is expensive, we proposed efficient approximation algorithms whose accuracy increases with additional monitors. We also proposed accumulated outcome probability that makes use of timestamp information of the infected monitors. In addition, we presented a new monitor selection method that maximizes the number of positive monitors, which gives more information for the error source identification.
References


114


