Using stack traces to identify failed executions in a Java distributed system

by

John Lambert

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

Advisor: Dr. H. Andy Podgurski

Department of Electrical Engineering and Computer Science

Case Western Reserve University

August 2002

Built on June 2, 2002 0:22

Using stack traces to identify failed executions in a Java distributed system

CASE WESTERN RESERVE UNIVERSITY
GRADUATE STUDIES
June 3, 2002

We hereby approve the thesis of

John Lambert

candidate for the **Master of Science**

degree. ⋆

Committee Chair____

Dr. H. Andy Podgurski
Thesis Advisor
Professor, Department of Electrical Engineering & Computer Science

Committee____

Dr. Gŭltekin Özsoyoğlu

Department of Electrical Engineering & Computer Science

Committee____

Dr. Lee White

Department of Electrical Engineering & Computer Science

⋆We also certify that written approval has been obtained for any proprietary material contained therein.

DEDICATION

This thesis is dedicated to my mother and father for their support and encouragement.

Contents

List of Tables

List of Figures

List of Algorithms

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Andy Podgurski, for his help in preparing this work.

The author of CLUTO, George Karypis, was kind enough to answer several questions.

I would like to thank my friends Ken Alverson, Brad Bower, Mike Camp, Chris Connelly, Aaron Jack, Tim Schnabel, and Rick Wash for their ideas and encouragement.

Without Melinda Minch's support, encouragement, and understanding, I would not have been able to complete this work.

Using stack traces to identify failed executions in a Java distributed system

by

John Lambert

Abstract

Observation-based testing says that given a large set of program executions, we can use cluster analysis on the profiles to filter the executions and identify unusual profiles. The few executions with unusual profiles can then be checked manually for compliance, reducing the testing effort needed to reveal defects. While observation-based testing has been successfully applied to large stand-alone programs, it has not been applied to distributed systems.

This thesis presents `Ixor`, a system which collects, stores, and analyzes stack traces in distributed Java systems. When combined with third-party clustering software and adaptive cluster filtering, unusual executions can be identified.

Several methods of comparing executions are presented and their effectiveness in identifying failed executions is evaluated against two non-trivial sample applications. The results suggest that this approach is highly effective in correctly identifying failed executions.

Chapter 1

INTRODUCTION

1.1 Overview of thesis

Distributed systems present two challenges: they can run on multiple machines in different physical locations and the introduction of network traffic and multithreading leads to non-determinism. Software testers have tried to identify failed executions with resource-intensive solutions such as application-level logging code, run-time instrumentation of executables, controlled lab environments, and custom execution platforms. An automated method of identifying failed executions of a distributed system is needed.

1.1.1 Challenges

Since we are dealing with distributed systems, we must take into account two issues.

First, by definition, a distributed system can use multiple machines in different physical locations. Internet-wide applications such as SETI@home run on computers all over the world. Enterprise applications such as a billing system may need to scale across multiple database and application servers.

Second, the introduction of multiple threads and network traffic leads to non-determinism. Threads are introduced to handle multiple jobs, background processing, or communication. (The Java RMI framework presents several interesting thread issues; see 3.2.) While network traffic and threads are essential to a distributed system, they are impossible to predict: packets may take 10 millisecond or 1 second to move between computers (or never arrive at all), and a thread may be blocked for longer than was expected by the developer.

1.1.2 Previous solutions

Several attempts have been made at identifying failed executions in distributed systems, but all of them have drawbacks.

1.1.2.1 Application-level logging

Application-level logging is inefficient at runtime, difficult to introduce system-wide, and difficult to manually interpret.[1] Assuming access to the source code, adding logging code to the appropriate places in a large system is non-trivial and results in the mixing of application logic and debugging logic.[2] Analyzing the logs is very difficult, as well: synthesizing n logs from m machines with different clock times is very difficult. Furthermore, having a “Debug” build with logging and a “Release” build without logging may expose bugs: a race condition with a window of 10 milliseconds may never exposed if a logging call takes 100 milliseconds.

[1]It is easy to sort multiple log files by time, but creating a coherent mental construct based the contents is still difficult.

[2]Aspect-oriented programming may be helpful here, but the analysis problem remains.

1.1.2.2 Runtime instrumentation

Runtime instrumentation of application code in order to introduce logging or tracing is sub-optimal for several reasons. Although some instrumentation can be done at compile time, dynamically loaded/mobile code will have to be instrumented at run-time, with a very high cost. As above, the system is no longer executing the same code as usual, possibly changing the outcome of an execution. {**Todo #1.0:** more reasons}

1.1.2.3 Controlled environments

Using "test labs" with controlled network environments is a popular development option but it can cause problems in the long run. Assumptions such as the length of time it takes to deliver a packet, or the hardware in use may not hold during deployment on a different network. Network problems are difficult to record and reproduce.

1.1.2.4 Custom platforms

Using a custom VM to retrieve low-level information is sub-optimal: executions take longer, it is resource intensive to create/acquire the custom VM, and it may not port to other platforms. Furthermore, the testing platform may then be different than the deployment platform, exposing more bugs.

1.1.3 This work

As shown above, there are two problems: getting the data about the system, and analyzing it to draw some meaningful conclusion.

The best way to get the data is to introduce a minimal disturbance into the system: execute the same code on the same platform at the same speed on the same network. The best way to analyze data is automatically, without user intervention. This work, a system called `Ixor`, fulfills the goals above for Java distributed systems using RMI:

1. The `Ixor` client captures stack traces from participants in a system
2. The `Roxi` tool analyzes information from multiple executions
3. Clustering software will group executions to help identify failures

1.2 Problem definition

1.2.1 Distributed system

From a high-level, a distributed system looks like Figure 1-1. There are multiple processes, with multiple threads, communicating with other processes via messages.

1.2.2 Communication mechanisms

In order to communicate messages between processes, there are three common levels of abstraction.

1.2.2.1 Sockets

Sockets are a flexible, general-purpose network communication mechanism. However, the burden on the application programmer is great: the design and implementation of an application-level client/server protocol requires a great deal of thought and programming effort.

1.2.2.2 Remote Procedure Calls

Traditionally, an alternative to sockets is the Remote Procedure Call (RPC), which will hide the communication behind a local procedure call [35]. As

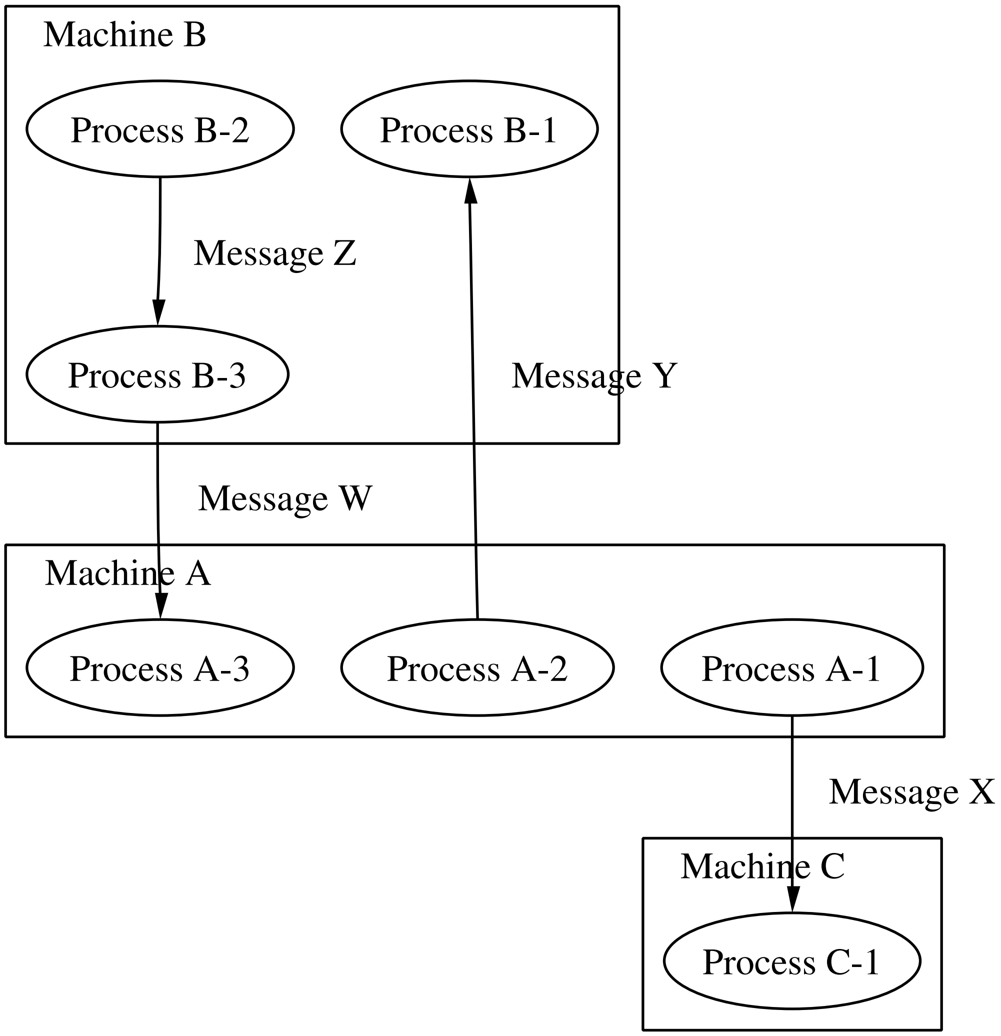


Figure 1-1: Distributed system.

far as the programmer is concerned, the function is local, but in reality, the parameters are being encoded and transferred to a remote target, which then sends back an encoded response.

1.2.2.3 Objects

However, the RPC abstraction does not work for object systems because communication between program-level objects is required [35]. In order to match the semantics of object invocation, distributed object systems require remote method invocation or RMI. In such systems, a local surrogate or stub object manages the invocation on a remote object.

1.2.3 Our problem

The point of this work is to find a way of detecting defective runs of programs in distributed systems by examining multiple executions.

We restrict the problem to Java RMI because it captures the semantics of a distributed system very cleanly: requests are sent to remote machines via a well-defined interface. The application logic is not hidden behind sockets: a stub class is used, but it has specific methods.

1.2.4 Generalizability

All work presented is generalizable to other platforms and programming languages; Java was chosen for four reasons. First, the profiling interface [56] is well-known. Second, it is easy to create sample applications and generate executions for it. Third, Java RMI captures the semantics of a distributed system very cleanly compared to sockets: requests are sent to remote machines via a well-defined interface. Fourth, the Java RMI runtime is quite easy to modify in order to insert both tracing code and fault injection code.

1.3 Observation-based testing

{**Todo #1.1:** Re-write this about observation-based testing.}

[13] describes how, with appropriate profiling, failures often have unusual profiles that are revealed by cluster analysis; failures often form small clusters or chains in sparsely-populated areas of the profile space. The estimation of software reliability using stratified sampling is presented in [41]. A capture/replay tool for observation-based testing of Java programs is presented in [49].

From [25]:

> The traditional paradigm for testing software is to construct test cases that cause runtime events that are likely to reveal certain kinds of defects if they are present. Examples of such events include: the use of program features; execution of statements, branches, loops, functions, or other program elements; flow of data between statements or procedures; program variables taking on boundary values or other special values; message passing between objects or processes; GUI events; and synchronization events.
>
> It is generally feasible to construct test cases to induce events of interest if the events involve a program's external interfaces, as in functional testing (black-box testing, specification-based testing). However, it often extremely difficult to create tests that induce specific events internal to a program, as required in structural testing (glass-box testing, code-based testing). For this reason functional testing is the primary form of testing used in practice. Structural testing, if it is employed at all, usually takes the form of assessing the degree of structural coverage achieved by functional tests, that is, the extent to which the tests induce certain internal events.

Structural coverage is assessed by profiling the executions induced by functional tests, that is, by instrumenting or monitoring the program under test in order to collect data about the degree of coverage achieved. If necessary, the functional tests are augmented in an ad hoc manner to improve structural coverage.

The difficulty of constructing test data to induce internal program events suggests an alternative paradigm for testing software. This form of testing, which we call observation-based testing, emphasizes what is relatively easy to do and de-emphasizes what is difficult to do. It calls for first obtaining a large amount of potential test data as expeditiously as possible, e.g., by constructing functional tests, simulating usage scenarios, capturing operational inputs, or reusing existing test suites. The potential test data is then used to run a version of the software under test that has been instrumented to produce execution profiles characterizing the program's internal events. Next, the potential test data and/or the profiles it induces are analyzed, in order to filter the test data: select a smaller set of test data that induces events of interest or that has other desirable properties. To enable large volumes of potential test data to be analyzed inexpensively, the analysis techniques that are used must be fully or partially automated. Finally, the output resulting from the selected tests is checked for conformance to requirements. This last step typically requires manual effort – either in checking actual output or in determining expected output.

Many forms of execution profiling can be used in observation-based testing. For example, one may record the occurrences of any of the kinds of program events that have traditionally been of interest in testing. Typically, a profile takes the form of a vector of event counts, although other forms, such as a call graph, may be used in observation-based testing. Since execution profiles are often very large – ones with thousands of event counts are common – automated help is essential for analyzing them. In structural testing, profiles are usually summarized by computing simple coverage measures, such as the number of program statements that were executed at least once during testing. However more sophisticated multivariate data analysis techniques can extract additional information from profile data. For example, [41] and [42] report experiments in which automatic cluster analysis of branch traversal profiles, used together with stratified random sampling, increased the accuracy of software reliability estimates, because it tended to isolate failures in small clusters.

Among the most promising multivariate data analysis techniques for use in observation-based testing are multivariate visualization techniques like correspondence analysis and multidimensional scaling. In essence, these computer-intensive techniques project many-dimensional execution profiles onto a two-dimensional display, producing a scatter plot that preserves important relationships between the profiles. This permits a human user to visually observe these relationships and, with the aid of interactive tools, to explore their significance for software testing.

. . .

A serious problem with synthetic test data is that it does not reflect the way that the software under test will be used in the field. Even if it reveals defects, it may not reveal those having a significant impact on the softwares reliability as it is perceived by users. By contrast, operational testing (beta testing, field testing) does reflect the way software is used in the field, and it also may reduce the amount of inhouse testing (alpha testing) software developers must do. In operational testing, the software to be tested is provided to some of its intended users to employ as they see fit over an extended period. The advantages of operational testing are somewhat offset by the fact that beta users often fail to observe or report failures, because they are unfamiliar with the softwares specification and because testing is not their primary occupation. This problem can be addressed by using a capture/replay tool to capture executions in the field, so they can later be replayed and examined in detail by trained testing personnel. If many executions are captured, it may be practical to examine only a fraction of them in this way. Rather than examining a random sample of executions, it is desirable to filter the captured sample to identify executions with unusual characteristics that may be associated with failure. Multivariate visualizations can be used to filter operational executions in much the same way they can be used to filter regression test suites.

1.4 xdProf

Earlier work on an independent study project called xdProf, published in [21], was the basis for some parts of the implementation. xdProf was a basic framework for implementing the system described in this thesis.

1.4.1 Overview of xdProf

We describe the design and implementation of xdProf: a tool that captures and analyzes stack traces sent at a fixed interval from Java Virtual Machines

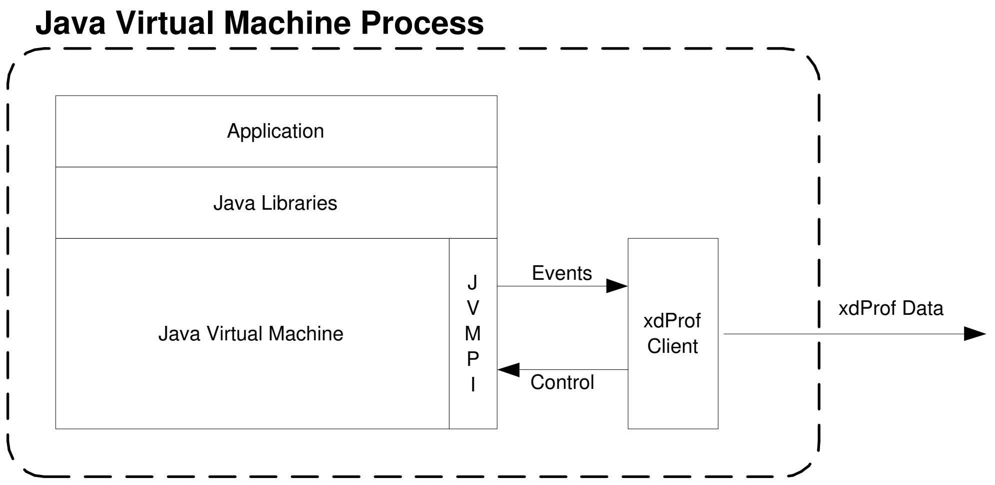


Figure 1-2: xdProf client architecture.

in a distributed system. The xdProf client uses the Java Virtual Machine Profiling Interface and works with any compliant implementation; no access to application source code is necessary, no library modifications are needed, and there is no run-time instrumentation of Java byte code. Configuration options given at virtual machine startup specify the interval for stack trace transmission and the remote xdProf server. The xdProf server collects information from multiple xdProf clients and provides an extensible interface for analysis. Current tools include a graphical user interface for viewing the most recent stack traces from multiple virtual machines and the generation of control flow graphs for each virtual machine. The performance impact of the xdProf client sending data over a local area network is minimal: less than a 8% increase in total elapsed time for a set of standard benchmarks.

1.4.2 xdProf client

The xdProf client runs on a Java Virtual Machine (JVM) and sends stack traces to a remote machine at fixed interval. It is a small dynamic link library (`xdProf.dll` or `libxdprof.so`) which can be used with the IBM Java Development Kit 1.3 and the Sun Java Development Kit 1.2 and 1.3; it is currently available on the Intel/Win32 platform, Intel/Linux, and Ultra/Solaris platforms. The xdProf client is invoked with the `-Xrun` command-line option: `java -XrunxdProf:server=`*machine_name*`,port=`*port_number*`,refresh=`*milliseconds* *ApplicationToRun*

The Java applet viewer (`appletviewer`) and Remote Method Invocation Registry (`rmiregistry`) can use the xdProf client when the `-J-XrunxdProf:...` command-line argument is used. Also, on some Java Virtual Machines, the environment variable `_JAVA_OPTIONS` can be set to the `-XrunxdProf:...` argument so all programs running on the Java virtual machine will automatically use the xdProf client.

1.4.2.1 Java Virtual Machine Profiling Interface

The xdProf client uses the Java Virtual Machine Profiling Interface (JVMPI), which was proposed as a "general-purpose and portable mechanism for obtaining comprehensive profiling data from the Java virtual machine...it is extensible, non-intrusive, and powerful enough to suit the needs of different profilers and virtual machine implementations."[56] Currently, both IBM and Sun support the JVMPI specification on various platforms: Windows, Linux, Solaris, Macintosh OS X, etc. The JVMPI eliminates the need for an instrumented Java Virtual Machine, and allows one profiler to work with many different virtual machines. In the current version of the JVMPI, only one profiler agent per virtual machine can be supported. The JVMPI sends events to a profiler that has registered its interest in specific events via JVM callbacks. xdProf uses seven of these events, listed in Table 1.1. The details for each event are available in [53].

xdProf will store or remove information about the thread or class upon receiving the appropriate JVMPI event. Once the JVM initialization done event is received, xdProf will create a background thread, described in 1.4.2.2, to communicate with the xdProf server and notify this thread when the JVM shutdown event is received.

xdProf also enables the object allocation event for a short period of time. The JVM initialization thread will allocate objects before a thread start event has been sent for it. xdProf uses the object allocation event to discover this thread, and request a thread start event for it via the JVMPI. After xdProf has retrieved all applicable information about the thread, the object allocation event will be disabled, increasing performance.

1.4.2.2 Communication Thread

Once xdProf has been notified that the JVM is initialized, xdProf starts a background communication thread (Figure 1-3). This thread initializes communications by attempting to connect to the xdProf server; if it cannot connect to the server specified in the command-line arguments, it will disable all future event notification, effectively unloading xdProf. Every *delay* milliseconds, xdProf will disable garbage collection and suspend all threads in the system; both these steps are necessary: garbage collection must be disabled so it does not start while call stacks are being accessed and running threads must be suspended so they do not change their call stacks while information is being collected. Information about the threads, methods, and classes is stored in the data format presented in 1.4.2.3. Suspended threads are resumed and the information is transmitted. Once xdProf has been notified that there is a virtual machine shutdown pending, the communication thread will close its socket to the server.

This "sampling" algorithm is based on the statistical CPU sampling algorithm used by the HPROF profiler [56], except that xdProf does not attempt to calculate or assign costs to the contents of the call stack. Like HPROF, this algorithm is independent of the number of processors and should work equally well on single processor and multiprocessor machines.

1.4.2.3 xdProf Data Format

The xdProf client sends plain-text ASCII data to the machine specified via command-line arguments. The data format, shown in Figure 1-4, is stateless: any information necessary to determine what is running in the VM is sent with the data; no history is assumed. Thread, class, and method identifiers are acquired from the JVMPI events indicating their creation, and are transmitted as eight-digit hexadecimal numbers.

The thread name, group name, and parent name are the values returned by the JVMPI when the thread event was received; changes at runtime (via `java.lang.Thread.setName`) are not visible. The logical start specified for each thread is a monotonically increasing integer corresponding to the order in which threads were started: the first thread started is assigned 1, the second thread is assigned 2, etc. Threads are sent in no particular order; however, the logical start value provides a way to order them by starting time and to determine a possible parent relationship (the child's parent name is the same as the possible parent's name, and the parent's logical start value is less than the child's logical start value). A gap in the sequence of logical start values indicates that the missing thread has terminated. Thread status is an integer indicating if the thread is runnable, waiting on a monitor, or waiting on a condition variable; if a thread is interrupted or suspended in any of these three states, a flag bit will be set.

After sending information about a thread, xdProf sends the number of frames in the call stack, and then the content of the call stack as a list of stack frames. Each stack frame consists of a method identifier and a line number. Line numbers will reference a line in the class source file or indicate a compiled method, a native method, or an unknown line number. The top of the call stack, the method currently executing, is sent first; the thread entry point is sent last.

After the number of methods is sent, xdProf will send the class identifier, method identifier, method name, and a method descriptor[3] for each method, in no particular order. To reduce network traffic, xdProf sends method information only for those methods that currently appear in the call stack. Class information is sent last and, to reduce network traffic, only classes with one or more methods in the call stack are sent. Inner and anonymous classes are sent with names as they are internally represented: `package.name.Outer$Inner`, `SomeClass$1`, etc. The xdProf client does not use all information accessible for classes: for example, names and data types of static fields and instance fields are omitted because transmitting this information would require significantly more network traffic.

1.4.2.4 Performance

The xdProf client introduces two distinct kinds of overhead with respect to application execution time. First, xdProf must process certain events which happen mainly near the beginning of a program's execution: loading of classes (such as the Java library classes), the starting of threads, and the completion of VM initialization. The second source of overhead is that xdProf must stop

[3]The method descriptor describes the data types of the parameters and return data type in a concise format: "`Object mymethod(int i, double d, Thread t)`" has the method descriptor "`(IDLjava/lang/Thread;)Ljava/lang/Object;`"[27].

every running thread every *delay* milliseconds, generate a call stack, look up applicable information, and send the data to the server.

SPECjvm98 We used the SPECjvm98 benchmark[47] to evaluate the effect of the xdProf client. The SPECjvm98 is a standard measure of the performance of a Java virtual machine and the underlying hardware. Several different tests are run and the elapsed time for each test is used to calculate a SPEC ratio with respect to a reference system. The SPECjvm98 and SPECjvm98_base metrics are the geometric means of the best and worst ratios for each test, weighted equally. Benchmarking is performed inside a web browser or Java applet viewer: the system under test will download the test classes from a remote web server and run them.

Table A.13 contains the SPECjvm98 and SPECjvm98_base results for the Sun Java 2 Runtime Environment 1.3 with HotSpot Client VM; we used an Intel Pentium II 350 MHz computer with 512 MB of RAM running Windows 2000 Professional for our testing.[4] xdProf was either not used at all, used to send data to the local machine every 100 milliseconds, or used to send data to a remote machine (different from the web server) every 100 milliseconds. Since we were interested in the performance effect of the client, we used a native code program [46] to receive data from the client, but did not process or analyze the data.

The SPECjvm98 value without xdProf is between 2.8% and 5.8% higher than with xdProf; the SPECjvm98_base value is 1.6% higher if xdProf sends data locally instead of not sending data, and not sending data at all is 5.2% faster than sending data remotely. However, since the SPECjvm98 benchmark runs as an applet and measures the elapsed time for test execution, it does not factor the time for Java VM initialization and shutdown.

[4]The test ratios used to determine these values and the complete system environment are listed in A.

Total elapsed time To measure the overall effect of xdProf, we measured the total elapsed time from JVM start to JVM shutdown of an application that ran each SPECjvm98 test exactly twice; this includes the "up-front" overhead such as loading library classes, etc. We tested against Sun's HotSpot Client and Classic (no just-in-time compilation; all byte code is interpreted) virtual machines and no changes were made to default garbage collection behavior.

Table A.14 shows that the overhead of xdProf on the HotSpot VM is between 1.93% and 7.76%, and that, as the delay between messages increases, the overhead generally decreases. Table A.15 is interesting because the overhead of xdProf sending information locally is significantly higher than the overhead to send data remotely. We believe that this is because more context switches are required for the Classic VM to send and receive data locally but more investigation is necessary. Preliminary performance measurements on the IBM 1.3 Classic VM are consistent with the Sun Classic VM performance results.

Network traffic As shown in A, xdProf sends approximately 4300 bytes per stack trace with the Classic VM and 2800 bytes per stack trace with the HotSpot VM, excluding TCP/IP transmission overhead. Network traffic is highly-application dependent: Sun's Forte for Java, Community Edition version 1.0[52] sent approximately 7100 bytes per trace when the Classic VM was used, and around 4300 bytes per trace when the HotSpot VM was used. The difference is due to the fact that the HotSpot VM will only report call stacks for threads that are not blocked so fewer stack frames, classes, and methods are sent. Overall, the Classic VM generated 50% more network traffic per stack trace. However, since execution of the elapsed time benchmarks took longer under the Classic VM, more stack traces were sent, and the Classic VM generated a total of fourteen to sixteen times the network traffic of the HotSpot VM.

1.4.3 xdProf server

The xdProf server receives and analyzes information from multiple xdProf clients. It supports the use of multiple analysis tools, which can be added or removed at runtime. The xdProf server and analysis tools are written in Java; it would be possible to write equivalent server programs in other languages or to use Java Remote Method Invocation to off-load analysis to another, more powerful machine.

1.4.3.1 Interfaces

Each analysis tool in the xdProf server is notified when an event happens: an xdProf client connects to the xdProf Server, a trace is received for a connection, or a client disconnects from the server. There are three functions in the `ServerListener` interface, listed in Table 1.2. A `Connection` object stores information about the client; a `Trace` object contains information about the classes, methods, and threads loaded, as well as a time stamp. The same analysis object is used for all connections; this allows the tool to examine global patterns.

1.4.3.2 Analysis Tools

We will describe two analysis tools that currently exist; they are written in Java and can be run simultaneously or separately.

GUI tool The GUI tool in Figure 1-6 displays the xdProf clients currently connected to the server, the threads running in a selected client, and the stack trace of a selected thread. There is a `Pause` button than will "lock" the GUI and allow the user to examine the stack traces of all the virtual machines at a specific point in time. (Since traces are received in any order and after different delays, this is not a global state.) After the user is done examining the system, clicking the `Pause` button again will update the display with the most recent information.

Call graph generator We also wrote a call graph generator for xdProf, the output of which is shown in Figure 1-7. This tool generates files for the `dotty` graph visualization program[36]; it is highly configurable and has a GUI front-end.

1.5 Related work

1.5.1 Debugging

[7] presents a framework for distributed debugging. [55] presents a new mechanism based on execution tracing and cryptography that allows a mobile agent owner to determine if some site in the route followed by the agent tried to tamper with the agent state or code. Issues in debugging optimized code are discussed in [9].[5] Mapping between source code and optimized code (the "code location problem") is discussed in [54]. [8] discusses how to produce an accurate call stack trace if frame pointers are occasionally absent. [62] shows how to integrate event visualization with sequential debugging.

1.5.2 Visualization

Some related work is from the visualization community. Walker in [58] describes "Visualizing Dynamic Software System Information through High-level Models." Earlier work by De Pauw, in [61], focused solely on object-oriented systems. [60] describes the visualization of concurrent and object-oriented systems, as well. A methodology for building application-specific visualizations of

[5] Sun's Java does not use an optimizing compiler, so this is not an issue for this work.

parallel programs is presented in [48]. A popular tool for visualization, and one used by [21], is presented in [36]. [18] talks about visualizing message patterns in object-oriented program executions. An overview of using visualization for debugging, including the use of sound, is in [2].

1.5.3 Java-related

Specifically related to Java client/server applications, [20] described how a custom, instrumented Java VM could be used to profile and trace events throughout a system, with an emphasis on performance monitoring. In it, the authors mentioned how JVMPI [56, 53], used in `Ixor`, might be used in the future to avoid the complex instrumentation required. Just-in-time compilation is discussed in [12].

1.5.4 Software engineering

From a program-understanding viewpoint, [28] describes how to use automatic clustering to produce high-level system organizations of source code. Understanding distributed software via component module classification is discussed in [33]. In [59], Walker describes the efficient mapping of software system traces to architectural views. A debugging and testing tool for supporting software evolution is presented in [1].

1.5.5 Sequences, patterns, and trees

[38] describes a numerical similarity/dissimilarity measurement for trees. [39] studies the problem of estimating a word by processing a noisy version which contains substitution, insertion, deletion and generalized transposition errors; this occurrs when transposed characters are themselves subsequently substituted, as is typical in cursive and typewritten script, in molecular biology and

in noisy chain-coded boundaries.

[6] describes two popular algorithms for sequence comparison: edit distance and gap distance, both used in `Roxi`. An influential paper on sequence matching and the basis for one of the algorithms used in `Roxi` was [26].

1.5.6 Distributed systems

An general overview of distributed systems is in [10]. An overview of virtual time and global state algorithms is given in [31]. The "Holy Grail" of detecting causal relationships in distributed computations is discussed in [44]. The use of Markov Nets and the benefits of using probabilistic models for distributed and concurrent systems is discussed in [3]. [32] gives a method of profiling paths across processes.

Finding consistent global checkpoints of a distributed computation is important for analyzing, testing, or verifying properties of these computations; [29] gives a theoretical foundation for finding consistent global checkpoints.

Distributed systems depend on consistent global snapshots for process recovery and garbage collection activity; [51] provides exact conditions for an arbitrary checkpoint based on independent dependency tracking within clusters of nodes.

A general functional model of monitoring in terms of generation, processing, distribution and presentation of information is shown in [30].

1.6 Trademarks

Java and HotSpot are trademarks of Sun Microsystems, Inc. Microsoft Visual C++ 7.0 is a trademark of Microsoft. All trademarks are property of their respective owners.

1.7 Organization

Chapter 2 gives the design of the components in `Ixor`. Chapter 3 discusses the algorithms used to analyze the data. Chapter 4 presents two case study applications and the results of executing `Ixor` on this system and its executions. Chapter 5 presents conclusions and ideas for future work. Appendix A contains information on the performance overhead of xdProf.

Table 1.1: JVMPI events used by xdProf.

Event Name	**Description**	**Information used by xdProf**
`THREAD_START`	A thread is started in the VM	Thread name, group and parent name, thread identifier
`THREAD_END`	A thread ends in the VM	Thread identifier
`CLASS_LOAD`	A class is loaded in the VM	Class name and identifier, source file, methods in the class
`CLASS_UNLOAD`	A class is unloaded in the VM	Class identifier
`JVM_INIT_DONE`	VM initialization is done	None
`JVM_SHUT_DOWN`	VM is shutting down	None
`OBJECT_ALLOC`	An object is allocated	Object identifier, class identifier, thread identifier

```
Connect to server; if not, disable all events and end this thread
While there is not a shutdown pending
      Wait the user-specified $delay$ using a JVMPI monitor
      Disable garbage collection
      Suspend all running threads in the virtual machine
      Gather information about the thread call stacks, methods, and classes
      Resume the threads that were suspended by xdProf
      Enable garbage collection
      Send information to remote server
Disconnect from the xdProf server
```

Figure 1-3: Algorithm for communication thread.

```
<VM process id> <command-line description>
<virtual machine, runtime, and operating system information>
<N: number of threads>
<thread 1 identifier>
<thread 1 name>
<thread 1 group name>
<thread 1 parent name>
<thread 1 logical start>
<thread 1 status>
<F: number of frames for thread 1>
<frame F method identifier> <frame F line number>
<frame F - 1 method identifier> <frame F - 1 line number>
...
<frame 1 method identifier> <frame 1 line number>
...other thread blocks...
<M: number of methods>
<method 1 class identifier> <method 1 identifier> <method 1 name> <method 1
descriptor>
...
<method M class identifier> <method M identifier> <method M name> <method M
descriptor>
<C: number of classes>
<class 1 identifier> <class 1 name> <class 1 source file>
...
<class C identifier> <class C name> <class C source file>
```

Figure 1-4: xdProf data format.

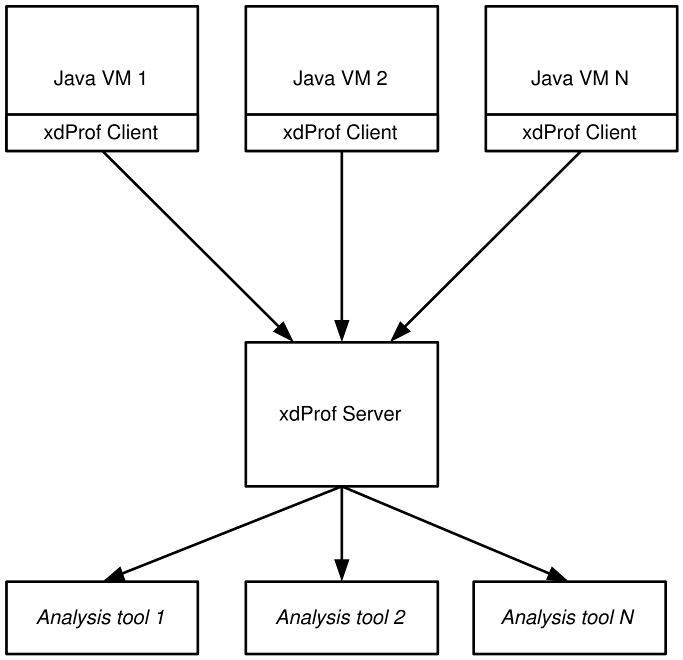
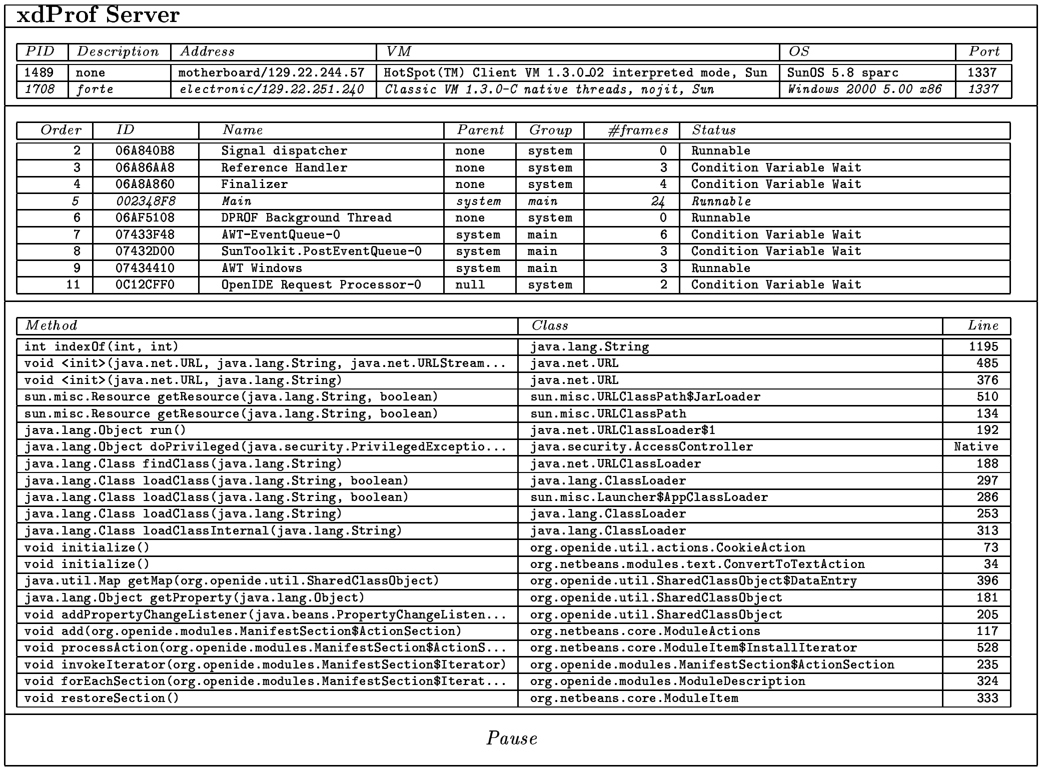


Figure 1-5: xdProf server architecture.

Table 1.2: xdProf analysis tool interface

`public void addConnection(Connection c);`	Called once when a new Connection arrives.
`public void addTrace(Connection c, Trace t);`	Called when a trace is received for a Connection.
`public void removeConnection(Connection c);`	Called when a Connection disconnects.



xdProf Server

PID	*Description*	*Address*	*VM*	*OS*	*Port*
1489	none	motherboard/129.22.244.57	HotSpot(TM) Client VM 1.3.0_02 interpreted mode, Sun	SunOS 5.8 sparc	1337
1708	*forte*	*electronic/129.22.251.240*	*Classic VM 1.3.0-C native threads, nojit, Sun*	*Windows 2000 5.00 x86*	*1337*

Order	*ID*	*Name*	*Parent*	*Group*	*#frames*	*Status*
2	06A840B8	Signal dispatcher	none	system	0	Runnable
3	06A86AA8	Reference Handler	none	system	3	Condition Variable Wait
4	06A8A860	Finalizer	none	system	4	Condition Variable Wait
5	*002348F8*	*Main*	*system*	*main*	*24*	*Runnable*
6	06AF5108	DPROF Background Thread	none	system	0	Runnable
7	07433F48	AWT-EventQueue-0	system	main	6	Condition Variable Wait
8	07432D00	SunToolkit.PostEventQueue-0	system	main	3	Condition Variable Wait
9	07434410	AWT Windows	system	main	3	Runnable
11	0C12CFF0	OpenIDE Request Processor-0	null	system	2	Condition Variable Wait

Method	*Class*	*Line*
int indexOf(int, int)	java.lang.String	1195
void <init>(java.net.URL, java.lang.String, java.net.URLStream...	java.net.URL	485
void <init>(java.net.URL, java.lang.String)	java.net.URL	376
sun.misc.Resource getResource(java.lang.String, boolean)	sun.misc.URLClassPath$JarLoader	510
sun.misc.Resource getResource(java.lang.String, boolean)	sun.misc.URLClassPath	134
java.lang.Object run()	java.net.URLClassLoader$1	192
java.lang.Object doPrivileged(java.security.PrivilegedExceptio...	java.security.AccessController	Native
java.lang.Class findClass(java.lang.String)	java.net.URLClassLoader	188
java.lang.Class loadClass(java.lang.String, boolean)	java.lang.ClassLoader	297
java.lang.Class loadClass(java.lang.String, boolean)	sun.misc.Launcher$AppClassLoader	286
java.lang.Class loadClass(java.lang.String)	java.lang.ClassLoader	253
java.lang.Class loadClassInternal(java.lang.String)	java.lang.ClassLoader	313
void initialize()	org.openide.util.actions.CookieAction	73
void initialize()	org.netbeans.modules.text.ConvertToTextAction	34
java.util.Map getMap(org.openide.util.SharedClassObject)	org.openide.util.SharedClassObject$DataEntry	396
java.lang.Object getProperty(java.lang.Object)	org.openide.util.SharedClassObject	181
void addPropertyChangeListener(java.beans.PropertyChangeListen...	org.openide.util.SharedClassObject	205
void add(org.openide.modules.ManifestSection$ActionSection)	org.netbeans.core.ModuleActions	117
void processAction(org.openide.modules.ManifestSection$ActionS...	org.netbeans.core.ModuleItem$InstallIterator	528
void invokeIterator(org.openide.modules.ManifestSection$Iterator)	org.openide.modules.ManifestSection$ActionSection	235
void forEachSection(org.openide.modules.ManifestSection$Iterat...	org.openide.modules.ModuleDescription	324
void restoreSection()	org.netbeans.core.ModuleItem	333

Figure 1-6: GUI tool.

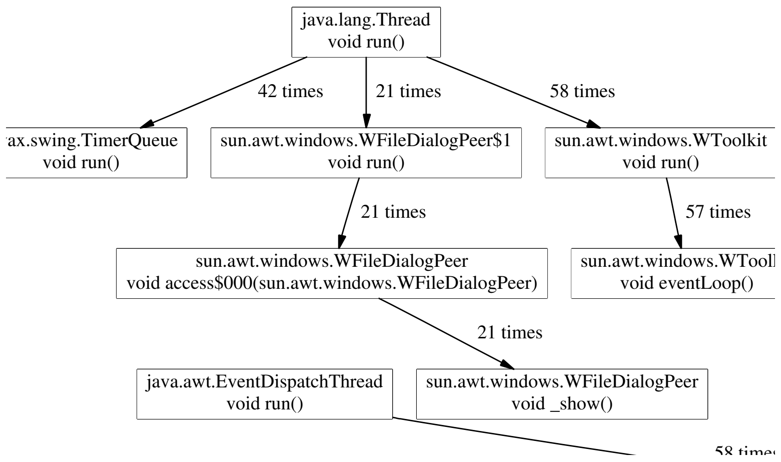


Figure 1-7: Call graph generator output (detail).

Chapter 2

SOFTWARE DESIGN

Overview

This chapter describes the design and implementation of `Ixor`. Figure 2-8 shows an overview of the `Ixor` system. There are three main components in the system:

- `Ixor` client collects stack traces (2.1)
- `Ixor` server requests stack traces (2.2)
- `Roxi` analyzes the data from the server (2.3)

2.1 `Ixor` Client

The `Ixor` client is a dynamic link library (`ixor.dll`) that uses the Java Virtual Machine Profiling Interface. It is a native binary for the Windows 32-bit platform written using Visual C++ 7.0.

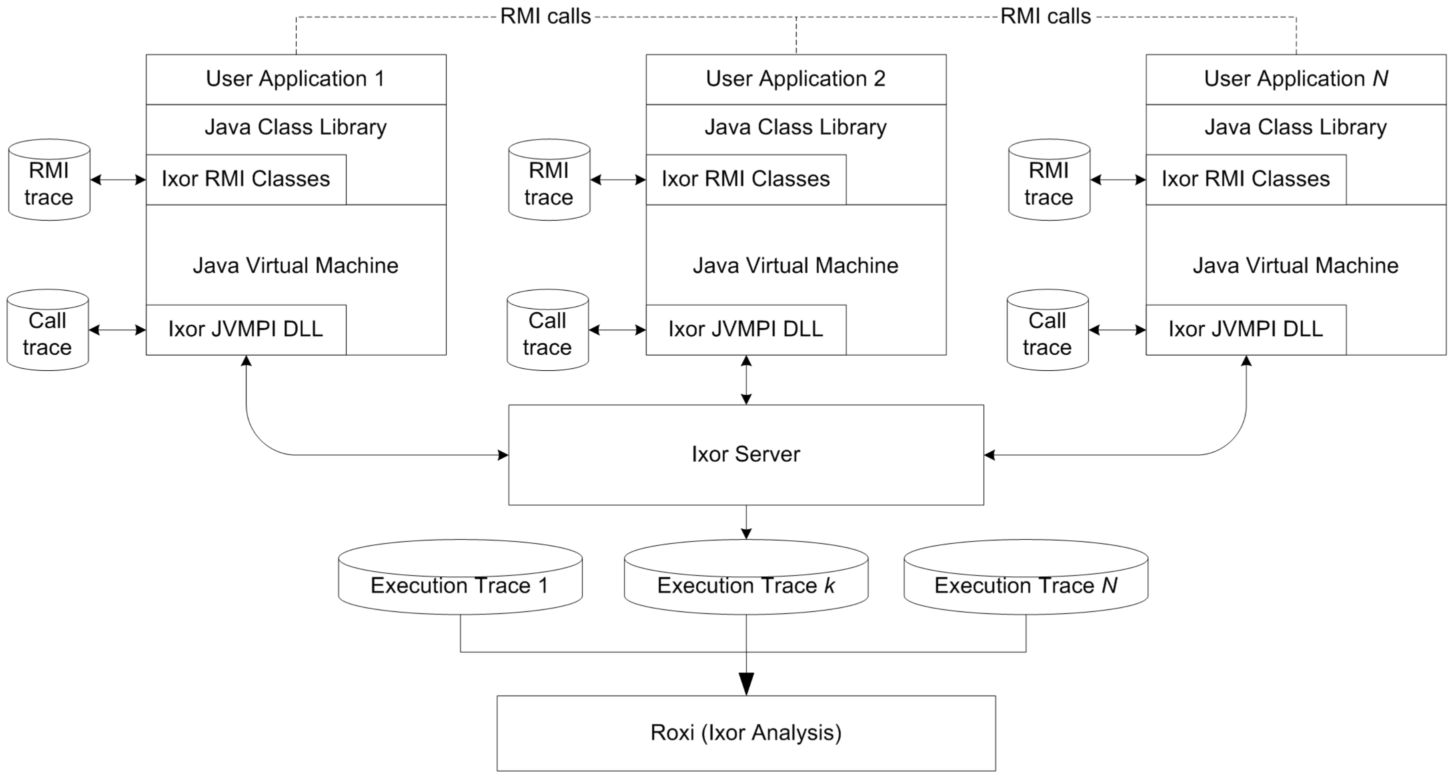


Figure 2-8: Overview of `Ixor` architecture.

2.1.1 Invocation

`Ixor` is invoked at runtime by loading the `Ixor` DLL with

`-Xrunixor:hostname=`*`H`*`,description=`*`Desc`*`,logfile=`*`path/Filename.log`*

The `hostname` H is the hostname of the remote `Ixor` server. If it is not specified, then it will default to `localhost`. The `description` is a required description of the VM in use: `Client1`, `Server`, etc. The `logfile` parameter is a file where the call stack traces will be stored; it can be a network location or the same directory.

Since this is inconvenient for the user, there is a script file which contains a predefined hostname and will set the parameters based off the VM description and a random number:

```
    T:\example>ixor MyDescription com.example.ClassToRun
-Dcom.example.option=Sample Argument1
```

will expand to

```
java -Xrunixor:hostname=electronic,description=MyDescription,
         logfile=T:/example/MyDescription31337.log
      com.example.ClassToRun
           -Dcom.example.option=Sample Argument1
```

It is possible to use `Ixor` with “native” Java applications like the Java applet viewer and Remote Method Invocation Registry if all arguments are prefixed with a `-J`.

2.1.2 Java Virtual Machine Profiling Interface

The `Ixor` client uses the Java Virtual Machine Profiling Interface (JVMPI), which was proposed in [56] as a “general-purpose and portable mechanism for obtaining comprehensive profiling data from the Java virtual machine… it is

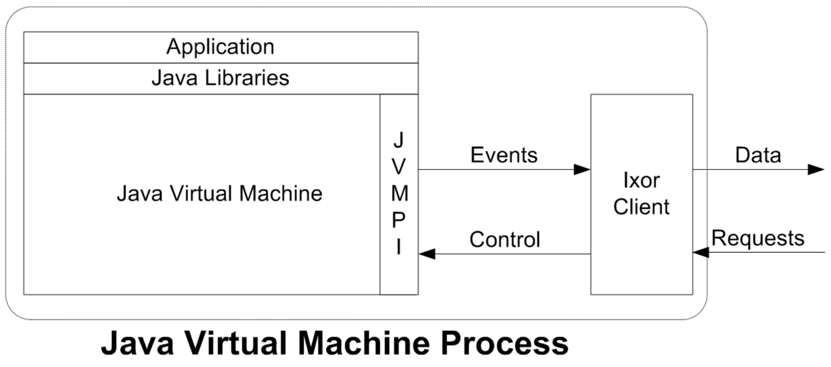


Figure 2-9: JVMPI Architecture.

extensible, non-intrusive, and powerful enough to suit the needs of different profilers and virtual machine implementations."

Currently, both IBM and Sun support the JVMPI specification on various platforms: Windows, Linux, Solaris, Macintosh OS X, etc. The JVMPI eliminates the need for an instrumented JVM, and allows one profiler to work with many different virtual machines. In the current version of the JVMPI, only one profiler agent per virtual machine can be supported. JVMPI does *not* define a wire format of any kind: the profiler is allowed to use any kind of communication.

As shown in Figure 2-9, JVMPI sends events to a profiler that has registered its interest in specific events via JVM callbacks. The profiler can send control messages to JVMPI at any time. `Ixor` receives requests and sends data from the `Ixor` server.

`Ixor` uses six of the JVMPI notification events, shown in Table 2.3. The details for each event are available in [53]. `Ixor` will store or remove information about the thread or class upon receiving the appropriate JVMPI event. Once the `VM initialization done event` is received, `Ixor` will create a background thread, described in 2.1.3, to communicate with the `Ixor`

Table 2.3: JVMPI events used by `Ixor`

Event Name	**Description**	**Information used by `Ixor`**
`THREAD_START`	A thread is started	Thread name, group and parent name, thread identifier
`THREAD_END`	A thread ends	Thread identifier
`CLASS_LOAD`	A class is loaded	Class name and identifier, source file, methods in the class
`CLASS_UNLOAD`	A class is unloaded	Class identifier
`JVM_INIT_DONE`	VM initialization is done	None
`JVM_SHUT_DOWN`	VM is shutting down	None

server and notify this thread when the JVM shutdown event is received.

2.1.3 Communication thread

Once `Ixor` has been notified that the JVM is initialized, `Ixor` starts a background communication thread (Algorithm 2.1. The general execution of this thread proceeds as follows, with line numbers in parentheses referring to Algorithm 2.1:

[1]

- Create a TCP connection T to the server. (line 1)
- Send the UDP port number u that the client will listen on. (lines 2 – 3)
- Start listening on u for a requested packet. (line 4)
- While there isn't a shutdown requested: (line 5)
    - Wait for a packet on u (line 6)

[1]Podgurski comment: Why use both protocols?

  - Extract the packet number p from the UDP packet (line 6)
  - Wait until all currently running events are complete (line 7) [2]
  - Disable garbage collection (line 8)
  - Suspend all other running threads in the virtual machine (lines 9 – 11)
  - Record the high-resolution counter value v (line 12) [3]
  - Gather information about the call stacks, methods, and classes (lines 13 – 17)
  - Resume all threads (lines 18 – 19)
  - Enable garbage collection (line 20)
  - Save system information, counter value v, packet number p, and information to log file (lines 21 – 24)

- Send the log file name to the server via TCP if it has not been sent (line 25)

2.1.3.1 Details of algorithm

Three things bear discussion.

TCP and UDP First, we use both TCP and UDP for communication. A TCP connection is used for initial setup and transferring the location of the log file name; obviously, this must be reliable, so TCP is the appropriate choice. However, transferring the packet numbers (requesting a stack trace) is done via UDP. The reason is simple: it is better to lose a packet (using the method described in 2.1.3.2) than to receive/process every packet inaccurately.

[2]Podgurski comment: How is this implemented?
[3]Podgurski comment: Where is this defined?

Synchronization Second, JVMPI events can arrive on any thread, at any time. To prevent two threads from simultaneously modifying global data, the JVMPI event processing function will enter a critical section *CS*, as discussed in 2.1.4.1, when processing begins, and will exit the critical section when processing is complete. The server must enter the same critical section *CS* in order to make sure the data is in a consistent state.

High-resolution counter Third, the high-resolution counter is a 64-bit integer, obtained from the `QueryPerformanceCounter` Win32 API. The resolution of the timer is found by calling `QueryPerformanceFrequency`, which returns the number of times the high-resolution timer increments itself every second. On an Athlon XP 1800+, this value is 3,579,545, giving a resolution of 2.79e-7 seconds. However, the actual value returned is irrelevant: the thing that matters is that it always increases between calls on a specific virtual machine.

[4]

The communication thread initializes communications by attempting to connect to the `Ixor` server; if it cannot connect to the server specified in the command-line arguments, it will disable all future event notification, effectively unloading `Ixor`. `Ixor` creates a TCP connection to the server and sends a UDP port number to the server to indicate where the client will be listening for requests.

When a UDP packet is received from the server, `Ixor` will wait until all currently executing events are complete: threads may be modifying the class lookup table or the list of threads. `Ixor` will disable garbage collection and suspend all threads in the system. Both these steps are necessary: garbage collection must be disabled so it does not start while call stacks are being

[4]Podgurski comment: ?? – 15

Algorithm 2.1 `Ixor` communication thread

```
1: $T \leftarrow$ TCP-Connection($ServerName$, $ServerPort$)
2: $u \leftarrow$ port for UDP communication
3: Send($T$, $u$)      // Send the value $u$ to the server
4: Listen($u$)
5: while not Shutdown-Requested?() do
6:    $p \leftarrow$ Extract-Value(Read-Integer(u))
7:    Wait-For-Running-Jvmpi-Events-To-Complete()
8:    Disable-Garbage-Collection()
9:    $Suspended \leftarrow \{\}$
10:   for all Thread $t \in$ Running-Threads() do
11:      Suspend($t$), $Suspended \leftarrow Suspended \cup \{t\}$
12:   $v \leftarrow$ Get-High-Resolution-Counter()
13:   $StackFrames \leftarrow$ Get-Call-Stacks(All-Threads())
14:         // Now that we have the method identifiers, we find the method
      details for methods we haven't already stored
15:   $MethodsToRecord \leftarrow$ Get-Methods($StackFrames$) $-$ $RecordedMethods$
16:         // Locate the class details for the methods we're going to store
17:   $ClassesToRecord \leftarrow$ Get-Classes($MethodsToRecord$) $-$ $RecordedClasses$
18:   for all Thread $t \in Suspended$ do
19:      Resume($t$)
20:   Enable-Garbage-Collection()
21:   Write($LogFile$, $v$, $p$, $StackFrames$, $MethodsToRecord$, $ClassesToRecord$)
22:         // Save the fact that we've used the methods and classes, so we
      don't save them again
23:   $RecordedMethods \leftarrow RecordedMethods \cup MethodsToRecord$
24:   $RecordedClasses \leftarrow RecordedClasses \cup ClassesToRecord$
25: Send($T$, Name($LogFile$))
```

accessed and running threads must be suspended so they do not change their call stacks while information is being collected.

`Ixor` stores more information than the user-readable stack trace in Figure 2-10; information about the threads, methods, and classes is stored in the data format presented in 2.1.5.

After all information has been stored, suspended threads are resumed, garbage collection is enabled, the information is transmitted, and the client then waits for another UDP packet. Once `Ixor` has been notified that there is a VM shutdown pending, the communication thread will close its socket to the server.

2.1.3.2 Dropping packets

To achieve a reasonable degree of accuracy, `Ixor` must not "get behind" on requests from the server. If we queue requests, the time when they are processed will diverge further and further from the time they should have been processed. By setting the `SO_RCVBUF` option on the UDP socket U to 1, we avoid building a queue by dropping packets.

`Ixor` will ignore packets with a lower number than the most recently received packet. This is necessary because we are using UDP: packets could arrive as 9, 11, 10.

2.1.3.3 UDP Port selection

The client must choose a UDP port number to use at run time and send it to the server. Hardcoding would not work because two VMs on the same machine would attempt to the use same UDP port. To get around this issue, a port number is generated by adding a fixed offset to the current process ID. Now, two VMs on the machine will have different port numbers.

2.1.4 Implications of approach

There are several implications of using JVMPI in `Ixor`.

2.1.4.1 Synchronization

Since JVMPI is cross-platform, it provides a method of synchronization in the form of "raw monitors" [53]. A raw monitor is similar to a Java monitor, except that it is not associated with a Java object. It is represented internally with an opaque pointer and has an associated string name. These monitors support the standard operations: Create, Enter, Exit, Wait (with specific timeout), NotifyAll (all waiting threads), and Destroy.

However, JVMPI monitors are unsuitable in `Ixor` for two reasons. First, they provide unnecessary semantics: `Ixor` does not need NotifyAll or a Wait with timeout for the most time-critical sections of code. Second, JVMPI must go through the Java Virtual Machine in order to perform the synchronization operation. Therefore, `Ixor` uses Win32-specific Critical Sections. These Critical Sections are fast: they execute mostly in user-space and are just above the operating system level. They provide enough functionality and avoid the indirection of going through the Java Virtual Machine.

2.1.4.2 Virtual machine choice

While JVMPI is widely supported, the amount of information available to the profiling agent is dependent on the virtual machine implementation.

The default JVM for the Sun Java 2 Runtime Environment, Standard Edition 1.3.1 is the HotSpot Client VM. The HotSpot VM is an "adaptive optimizer" which will perform various optimizations at run-time: method inlining, loop unrolling, and just-in-time compilation to native code [16]. The HotSpot VM supports the events given in Table 2.3 but, as a side-effect of method inlining and JIT compilation, does not return full stack traces. In addition, it can only return stack traces for threads which are currently running.

2.1.4.3 Class file information

Class files can contain extended information such as the line numbers for methods and the source code file they were compiled from. However, use of the `-g:none` flag during compilation prevents this information from being included. In some situations, most notably low-bandwidth connections, omitting this information is preferable, but use of code such as this with `Ixor` will result in decreased information for analysis. Regardless, `Ixor` will provide as much information as possible.

A related issue is the use of bytecode optimizers/obfuscators such as [43]. These tools will rename and reorder classes, methods, and method bodies so they cannot be reverse-engineered as easily. For example, this code

```
class Sample
{
    class Inner { /*...*/ }
    public Inner calcInt(int i) { /* ... */ }
    public Inner displayString(String s) { /* ... */ }
}
```

when compiled could be transformed via a bytecode optimizer to a class file similar to the results of compiling

```
class A
{
    class a { /* ... */ }
    public a b(String s)  { /* ... */ }
    public a b(int i)  { /* ... */ }
}
```

While increased speed and decreased size are usually beneficial, this arrangement provides less information to `Ixor` and is not recommended. (See 3.4.3.1 for more on how this could be misleading to `Ixor`.)

2.1.4.4 Class loading considerations

In Java, users can create custom classloaders which load class files from locations other than disk. For example, an applet in a web browser uses a URL classloader to download code from a website. `Ixor` works with custom classloaders; no modifications are necessary to either `Ixor` or the custom classloader.

2.1.5 `Ixor` Data Format

The `Ixor` client saves plain-text ASCII data to the logfile specified in the command-line arguments. The data format is shown in Figure 2-11. Thread, class, and method identifiers are acquired from the JVMPI events indicating their creation, and are transmitted as eight-digit hexadecimal numbers (`04EFBFB0`). Theoretically, a class identifier could occur as a method identifier, although this has not been seen.

The data format minimizes redundant information: once a class or method is recorded, it will not be saved again.[5] `Ixor` tracks this by maintaining a `hash_map` from method/class identifiers to a boolean value indicating whether or not it has been saved already; the default is that it has not been recorded. When classes are unloaded, these values are set to false in case the method or class identifier is reused.

The thread name, group name, and parent name are the values returned by the JVMPI when the thread event was received; changes at runtime (via `java.lang.Thread.setName`) are not visible because the JVM does not propagate them to JVMPI.

The logical start given for each thread is a monotonically increasing integer

[5]xdProf was significantly different: it would send traffic over the network and also used a stateless model in which all class and method information was sent with every trace. This was required because xdProf had multiple analysis tools entering at different points in the execution; since we only have one analysis tool, we can optimize for this.

corresponding to the order in which threads were started on the VM: the first thread started is assigned 1, the second thread is assigned 2, etc.[6] A gap in the sequence of logical start values indicates that the missing thread has terminated. If a logical start value has never occurred in a stack trace, but logical start values greater than it have, then the thread was "missed" by `Ixor`: if a stack trace has threads {10, 11} and the next stack trace has threads {10, 11, 13}, thread 12 was missed.[7]

Threads are recorded in no particular order; however, the logical start value provides a way to order them by starting time and to determine a possible parent relationship. A thread P is the possible parent of thread C (that is, it is possible that P started C) if $P.name = C.parent\_name$ and $P.logical\_start < C.logical\_start$.[8] Thread status is an integer indicating if the thread is runnable, waiting on a monitor, or waiting on a condition variable; if a thread is interrupted or suspended in any of these three states, a flag bit will be set.

After recording information about a thread, `Ixor` records the number of frames in the call stack, and then the content of the call stack as a list of stack frames. Each stack frame consists of a method identifier and a line number. Line numbers will reference a line in the class source file or indicate a compiled method, a native method, or an unknown line number. The top of the call stack, the method currently executing, is saved first; the thread entry point is recorded last.

After the number of methods is recorded, `Ixor` will save the class identifier, method identifier, method name, and a method descriptor for each method, in no particular order. The method descriptor describes the data types of the

[6]More specifically, the logical start value corresponds to the order in which the `THREAD_START` event was processed.

[7]This occurs with very short running "worker" threads.

[8]Changes at run-time would defeat this, as would multiple threads with identical names.

Table 2.4: Descriptor formatting

Signature	Programming Language Type
`Z`	boolean
`B`	byte
`C`	char
`S`	short
`I`	int
`J`	long
`F`	float
`D`	double
`V`	void (only valid for return types)
`L`*`fully-qualified-class`*`;`	fully qualified class name:Ljava/lang/String;
`[`*`type`*	array of type: `type[ ]`
`(`*`arg-types`*`)`*`ret-type`*	method type: X,Y,Z is converted to XYZ.

parameters and return data type in a concise format:
`Object mymethod(int i, double d, Thread t)` has the method descriptor `(IDLjava/lang/Thread;)Ljava/lang/Object;` [27]. Table 2.4 describes the format for descriptors.

To reduce storage requirements, `Ixor` saves method information only for those methods that currently appear in the call stack, and that have not been seen before. Class information is saved last and, to reduce storage, only for classes with one or more methods in the call stack and that have not been seen before. Inner and anonymous classes are recorded with names as they are internally represented: `package.name.Outer$Inner`, `SomeClass$1`, etc.

The `Ixor` client does not use all information accessible for classes: for example, names and data types of static fields and instance fields are omitted because saving this information would require significantly more space.

2.2 Ixor Server

The `Ixor` server is a Win32 binary written in Visual C++ 7.0. It is a stand-alone executable which requests traces at a fixed interval from `Ixor` clients and serializes the results to text files. For performance reasons, no analysis is performed at this stage.

2.2.1 Algorithm

The `Ixor` server is started with a directory for output. It proceeds as in Algorithm 2.2.

Algorithm 2.2 Server main program

```
Require: $args$ is an array where $args[1]$ is an output directory with a trailing
    slash and $args[2]$ is a delay in milliseconds
1: $OutputDirectory \leftarrow args[1]$
2: CREATE-DIRECTORY($OutputDirectory$)    // For output
3: $FilesList \leftarrow$ OPEN-FILE(STRINGIFY($OutputDirectory$, “files.list”))
4: CREATE-THREAD(BACKGROUND-UDP-THREAD($args[2]$))    // Algorithm 2.3
5: while true do
6:    $s \leftarrow$ ACCEPT-CONNECTION(SERVER_PORT)
7:    CREATE-THREAD(CONNECTION-THREAD($s$))    // Algorithm 2.4
```

Algorithm 2.3 Server BACKGROUND-UDP-THREAD(Millisecond *delay*)

```
1: $PacketNumber \leftarrow 0$
2: while true do
3:    SLEEP($delay$)
4:    if $|ConnectedVMs| = 0$ then
5:       continue
6:    $PacketNumber \leftarrow PacketNumber + 1$
7:    for all VM $v \in ConnectedVMs$ do
8:       SEND($v.socket$, $v.udp\_port$, “$PacketNumber$”)
```

Algorithm 2.4 Server CONNECTION-THREAD(Socket s)

```
1: $udp\_port \leftarrow$ READ-INTEGER($s$)
2: $process\_id \leftarrow udp\_port -$ BASE_UDP_PORT
3: $f \leftarrow$ OPEN-FILE(STRINGIFY($OutputDirectory$, IP-ADDRESS($s$), "-", $process\_id$, ".ixor"))
4: WRITE($FilesList$, NAME($f$))
5: $ConnectedVMs \leftarrow ConnectedVMs \cup \{$VM($s$, $udp\_port$, $process\_id$)$\}$
6: while IS-OPEN($s$) do
7:     WRITE($f$, READ-STRING-BLOCKING($s$))
```

2.2.2 Output

All output is sent to a directory specified on the command-line. This directory will contain a list of the files containing stack trace information (`files.list`) and files for participating VM's (`<address>-<process-id>.txt`). (Note that the process ID can be calculated as specified in 2.1.3.3.)

2.3 `Roxi` Analysis Tool

The `Ixor` Analysis Tool, `Roxi`, loads all information into memory and analyzes it. It takes a directory containing `files.list` as its argument.

Each file name in `files.list` is named according to `<address>-<process-id>.txt`. This file then contains the log file specified by the `Ixor` client during startup. `Roxi` will open the log file and read in all traces. Each stack trace contains the location of the RMI log file (2-11); `Roxi` will load this RMI log file as well.

The analysis is described in Chapter 3.

2.4 Differences with xdProf

Although based on xdProf ([21] and 1.4), `Ixor` is substantially different:

Stack traces `Ixor` clients wait until the centralized server sends a request before recording a stack trace. This gives a “snapshot” of the entire distributed system at any given time. xdProf clients send data every k milliseconds after starting up; there is no way to determine what was executing at the same time across clients.

Log files `Ixor` clients do not send each stack trace to the server; they only send the location of the log file, minimizing network usage. xdProf clients do not use local storage: they send the complete stack trace contents at each time.

Analysis and data format In `Ixor`, analysis is done off-line: data files are analyzed at a later time by `Roxi`, the analysis tool. `Ixor`’s data format takes advantage of this fact and is more compact: class names and method information are only stored once.

Since xdProf was intended for interactive use, multiple analysis tools could be loaded at any time. Each stack request received by the xdProf server needs to contain class names and method information instead of identifiers referring to previously received information. As a result, xdProf data files are significantly larger.

Extensibility The primary purpose of `Ixor`’s analysis tool, `Roxi`, is to do cluster analysis. It is easy to add additional distance measurements, but more difficult to perform visualization. Since all analysis is performed is off-line, however, different users can analyze the data in different ways. In contrast, xdProf supports any kind of analysis tool, but only by one user at the same time the system is running.

Implementation `Ixor` is completely written in Win32-specific C++, although it can be ported to other platforms. The xdProf client was written in cross-platform C++, but the server and analysis tools are completely written in Java.

```
"RMI TCP Connection(2)-129.22.251.240" daemon
 prio=5 tid=0x887d80 nid=0x4e4 runnable [0x92af000..0x92afdbc]
    at java.net.SocketInputStream.socketRead(Native Method)
    at java.net.SocketInputStream.read(SocketInputStream.java:91)
    at java.io.BufferedInputStream.fill(BufferedInputStream.java:186)
    at java.io.BufferedInputStream.read(BufferedInputStream.java:204)
    at java.io.FilterInputStream.read(FilterInputStream.java:69)
    at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:446)
    at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:706)
    at java.lang.Thread.run(Thread.java:484)
```

Figure 2-10: A user-readable stack trace for a single thread.

```
<VM process id>
<Ixor description and RMI log file location>
<virtual machine, runtime, and operating system information>
<P: packet number from the server this request corresponds to>
<V: high-resolution counter value for this packet>
<N: number of threads>
<thread 1 identifier>
<thread 1 name>
<thread 1 group name>
<thread 1 parent name>
<thread 1 logical start>
<thread 1 status>
<F: number of frames for thread 1>
<frame F method identifier> <frame F line number>
<frame F - 1 method identifier> <frame F - 1 line number>
...
<frame 1 method identifier> <frame 1 line number>
...other thread blocks...
<M: number of methods not saved previously>
<method 1 class identifier> <method 1 identifier>
     <method 1 name> <method 1 descriptor>
...
<method M class identifier> <method M identifier>
     <method M name> <method M descriptor>
<C: number of classes not saved previously>
<class 1 identifier> <class 1 name> <class 1 source file>
...
<class C identifier> <class C name> <class C source file>
```

Figure 2-11: Ixor data format.

Chapter 3

ALGORITHMS FOR ANALYSIS

Overview

This chapter describes how we analyze the data collected by the `Ixor` software. There are two steps. First, we compare the executions using different strategies. Second, we cluster the executions using CLUTO, a clustering toolkit.

3.1 Definitions

- Call stack: the methods being executed by a given thread.
- Thread: a single thread of execution in a given process.
- Process: a Java Virtual Machine[27]
- Stack trace: all the call stacks for all threads in one system at a given point of time.[1]

[1]Packet number in this system represents the time.

- Execution: all the traces for all the VMs in the system over a single run.

3.2 Comparing executions

Each execution consists of multiple virtual machines. Each virtual machine has a role: a descriptive label ("Client A", "Message Server", etc.) which is constant across executions. We would like to compare each execution with every other execution.

Before measuring the similarity between two executions, we must first normalize the stack traces across all executions by resolving opaque identifiers to their symbolic equivalent, and then mapping that symbolic equivalent to a value that will be shared across all executions. For example, execution 1 has a method identifier `0313AFVD` which corresponds to `sleepWithInterrupt`; execution 2 has an method identifier `30414A409` which also corresponds to `sleepWithInterrupt`. We "replace" both of these identifiers with a globally shared value of, for example, `49`.[2]

RMI Threads

RMI will use multiple worker threads to handle requests: `RMI TCP Connection(`***`thread_number`***`)` etc. RMI makes no guarantee about object/call/thread mapping [35], and so a call stack which occurs in `RMI TCP Connection(1)` may have the same purpose as one which occurs in `RMI TCP Connection(2)`. In order to get around this, we truncate the thread name after the first "(", [, or "-" character: all RMI TCP Connections are considered as one "thread."

Since we are interested in the "global state" of the system, we only consider stack traces which share a packet number with at least one other stack trace. In other words, if VM 1 sends a call stack trace c for packet number x, but no

[2]This has the added benefit of avoid string comparisons during the analysis stage.

other VM does, we will not consider c. However, if VM 1 sends d and VM 2 sends e for packet number y, we will consider both d and e. This number can be adjusted so only stack traces which occur when all VM's are participating.

3.2.1 Comparing all executions cleverly

Algorithm 3.1 shows how to compare all executions. It builds a matrix of differences such that $Result\,[i, j]$ is the difference between Execution[i] and Execution[j], generated by finding the differences between their member VM's.

Algorithm 3.1 Comparing all executions

```
 1: $Executions \leftarrow$ LOAD-AND-NORMALIZE-EXECUTIONS()
 2: for $i = 1$ to $|Executions|$ do
 3:    for $j = 1$ to $|Executions|$ do
 4:       if $i \geq j$ then // we already calculated it
 5:          continue
 6:       $ExecutionDifference \leftarrow 0$
 7:       for all VM $v \in Executions[i]$ do
 8:          // Let $m$ be the VM in $Executions\,[j]$ which has the same role
 9:          $m \leftarrow$ VM $e \in Executions\,[j]$ where $m.role = v.role$
10:          $ExecutionDifference$ += COMPARE($v$, $m$) + COMPARE($m$, $v$)
11:    $Results[i,\ j] = Results[j,\ i] = ExecutionDifference$
12: Save results matrix
```

It is easy to see that this has time complexity $\frac{n(n-1)}{2}$ where $n = |Executions|$ if we only consider "symmetric" measurements: $ExecutionDifference(e_1, e_2) = ExecutionDifference(e_2, e_1)$. If the metric is not symmetric, the cost is proportional to n^2. (Note that the Compare function for VMs does not have to be symmetric., although it is in `Ixor`.)

In order to compare VMs with the COMPARE(v, m) function, we use Algorithm 3.2.

Algorithm 3.2 COMPARE(v, m): comparing VM v to m

```
Require: DISTANCE($c$, $d$) calculates distance between call stacks $c$ and $d$
1: $diff \leftarrow 0$
2: for all Thread $t \in$ v do
3:    if $m$ has a thread with the same name then
4:       $MCallStacks \leftarrow$ GET-CALL-STACKS($m$, $t.name$)
               // Get all the call stacks which occur in a thread of the same
          name in VM m
5:       for all CallStack $c$ which occurs in $t$ on $v$ do
6:          // Find the most similar call stack on m
7:          $bestMatch \leftarrow (\min \text{DISTANCE}(c, d)$ for $d \in MCallStacks)$
8:          $diff$ += $bestMatch \times$ (# of times $c$ occurs in $v$)
9: return $diff$
```

3.2.2 Intuition for multiplying call stacks by occurrences

Our intuition tells us that:

- If the call stacks are similar for a long period of time, then the VMs are very similar.
- If the call stacks are similar for a short period of time, then the VMs are somewhat similar.
- If the call stacks are different for a short period of time, then the VMs are somewhat different.
- If the call stacks are different for a long period of time, then the VMs are very different.

We approximate execution time as "the number of times a call stack occurred during execution." This is a very rough approximation: code might not even have been executing: the thread may be blocked, for example. To make the VM difference follow our intuition, we multiply the number of times the call stack occurred (large if over a long period of time, small if over a short period of time) by the distance between the call stacks (large if very different, small if very similar). By adding over all call stacks, we get a distance metric which is intuitive.

3.2.3 Intuition for Compare(a, b) + Compare(b, a)

We define the difference between executions to be the sum of $Compare(a, b)$ and $Compare(b, a)$ for all appropriate VMs. This is because we want execution similarity to be symmetric: if execution i is similiar to execution j, then execution j should be similar to execution i.

Therefore, if VM A is distance x from B, then B should be distance x from A. However, since we multiply the call stack distance by the number of occurrences, this is not necessarily true. Let there be three executions, E_A, E_B, E_C each containing one VM (respectively, A, B, C). Assume that VMs A, B, and C have one thread and one call stack each $(a, b = a', c = a'')$. Call stacks b and c will always have minimum distance from a, so we represent them as a' and a''. Say that A contains 10 occurrences of a, B contains 10 occurences of a', and C contains 100 occurences of a''. Let $d = Dist(a, a')$ and $e = Dist(a, a'')$. Then $Compare(A, C) = 10e$ and $Compare(C, A) = 100e$. Since we want the difference between executions to be symmetric, we add these two values, so Difference$(E_A, E_C) = 110e$.

If we did not add the difference, then Difference$(E_A, E_B) \approx$ Difference(E_A, E_C), which does not agree with our intuition that A is more similar to B because it has the same number of call stacks.

3.3 Comparing call stacks

Now that we have a framework for comparing executions and virtual machines, we want to find algorithms which will compare two call stacks. A call stack consists of multiple stack frames, where each stack frame consists of class

```
CS1: f -> g -> h -> i
CS2: f -> g -> h -> j
CS3: k -> j -> h -> g -> f
CS4: f -> g -> h -> f -> g -> h
```

Figure 3-12: Call stacks

name, method name, method descriptor, and line number. We want "similar" call stacks to have a low distance while very different call stacks should have a higher distance.

For example, in Figure 3-12, CS 1 and CS 2 are fairly similar: they differ only by the last (top) stack frame. However, CS 3 is very different from CS 1 and CS 2: it's closest to CS 2 when reversed.

3.4 Edit distance

One classical distance metric presented in [6] is the edit distance between two sequences: the number of insertion, deletion, or substitution operations it takes in order to transform one sequence into another. A lower-valued distance indicates that call stacks are more closely related than a higher-valued distance. In Figure 3-12, we would have to make one substitution to change CS 1 to CS 2 (change i to j).

The classical algorithm for measuring edit distance uses dynamic programming and runs in time $O(mn)$, where m and n are the length of the sequences [6]. The algorithm is shown in Algorithm 3.3; it depends on a user-specified strategy consisting of three functions:

- Ins(y, j) will return the cost of inserting $y[j]$
- Del(x, i) will return the cost of deleting $x[i]$

- SUB(x, i, y, j) will return the cost of substituting $x[i]$ with $y[j]$

Algorithm 3.3 EDIT-DISTANCE(x, y, *strategy*) (from [6])

```
1:  $m \leftarrow |x|$
2:  $n \leftarrow |y|$
3:  $T[-1,-1] \leftarrow 0$
4:  for $j \leftarrow 0$ to $n-1$ do
5:      $T[-1,j] \leftarrow T[-1,j-1]$ + strategy.INS($y$, $j$)
6:  for $i \leftarrow 0$ to $m-1$ do
7:      $T[i,-1] \leftarrow T[i-1,-1]$ + strategy.DEL($x$, $i$)
8:      for $j \leftarrow 0$ to $n-1$ do
9:
            $T[i,j] \leftarrow \min\{T[i-1,j-1] + \text{strategy.SUB}(x, i, y, j),$
                             $T[i-1,j] + \text{strategy.DEL}(x, i),$
                             $T[i,j-1] + \text{strategy.INS}(y, j)\}$
10: return $T[m-1,n-1]$
```

By changing these functions, we can get different cost metrics with different properties. I used several different strategies for analysis.

The Levenshtein distance strategy (3.4.1), created by Vladimir Levenshtein in 1965, is a very basic strategy: it has constant costs for the insertion, deletion, and substitution operations [26].

I created two classes of strategies: **location-sensitive** and **call-stack sensitive**. The four location-sensitive strategies (Favor end (3.5), Squared favor end (3.6), Favor beginning (3.7), and Squared favor beginning (3.8)) assign different costs depending only on the location of the change. For example, making changes at the beginning of the call stack might be more preferable (lower cost) than making changes at the end of the call stack. The details of the changes do not matter: editing `java.io.File.delete():711` to `javax.swing.JScrollpane.getHorizontalScrollBarPolicy():445` has the same cost as editing `java.io.File.delete():711` to `java.io.File.delete():714` (if the locations are the same).

The three call-stack senstive strategies (Call stack strategy 1 (3.10), Call stack strategy 2 (3.11), and Call stack strategy 3 (3.12)) assign different costs depending on both the location of the change *and* the contents of the call stack. Small changes (changing line numbers when the class and methods are the same) cost less than large changes (changing the class name). For example, making small changes (changing line numbers) at the beginning of the call stack might be more preferable (lower cost) than making large changes (changing classes, methods, and line numbers) at the end of call stack.

3.4.1 Levenshtein

The simplest of all edit distance strategies is the Levenshtein distance, created in 1965 by Vladimir Levenshtein [26]. Substituting a letter with itself costs nothing, and all other operations have a cost of one. Algorithm 3.4 shows the formal definition.

Algorithm 3.4 Levenshtein distance

$$
\begin{aligned}
\textsc{Ins}(x,i) &= 1\\
\textsc{Del}(x,i) &= 1\\
\textsc{Sub}(x,i,y,j) &= \begin{cases} 0 & \text{if } x[i]=y[j],\\ 1 & \text{otherwise}\end{cases}
\end{aligned}
$$

3.4.2 Location sensitive-strategies

3.4.2.1 Favor end

The "favor end" strategy decreases the cost per operation as the position gets higher: we prefer insertions and deletions at the end of the call stack instead

```
cost = 1
                              1   ~  6
                              2   |  2
                              3   |  3
                              4   |  4
                              5   |  5

cost = 1
                              1   |  1
                              2   |  2
                              3   |  3
                              4   |  4
                              5   ~  6

cost = 1
                              +      7
                              1   |  1
                              2   |  2
                              3   |  3
                              4   |  4
                              5   |  5
```

Figure 3-13: Levenshtein example

Compared with `1 2 3 4 5`	Levenshtein	Favor end	Favor end squared	Favor beginning	Favor beginning squared
`6 2 3 4 5`	1	6	26	1	1
`1 2 3 4 6`	1	2	2	5	17
`7 1 2 3 4 5`	1	7	37	1	1

Table 3.5: Comparison of Levenshtein with location-sensitive strategies

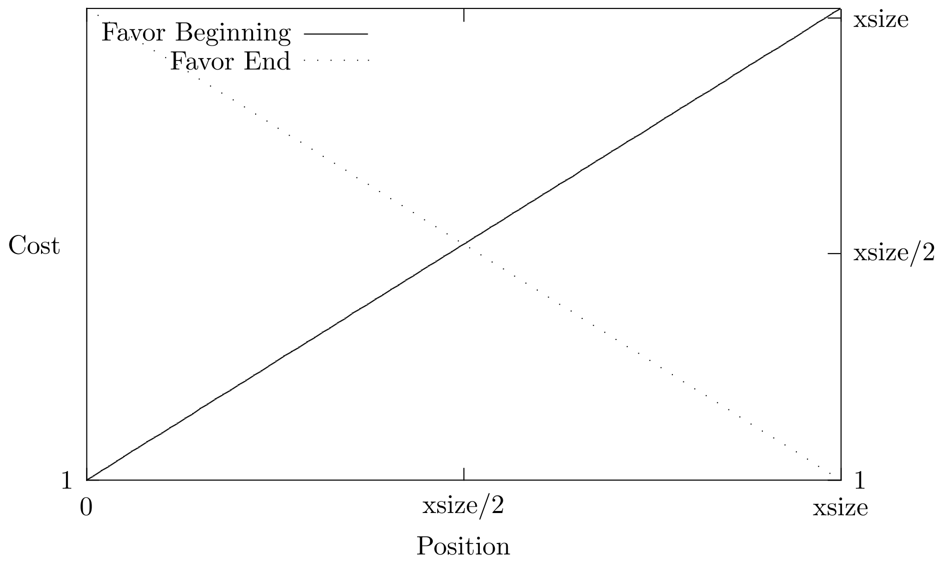


Figure 3-14: Favor insertion/deletion costs.

of at the beginning.[3] This makes some sense: as call stacks "diverge", their distance increases at a faster rate than it would with Levenshtein. The cost of a substitution is based on the average position.

Figure 3-14 shows the cost of insertions and deletions, while Figure 3-15 shows the cost of substitutions for the normal favor end strategy. Figure 3-16 shows the cost of insertions and deletions, while Figure 3-17 shows the cost of

[3]This strategy could be called "penalize beginning", as well.

{**Todo #3.2:** Generate this.}

Figure 3-15: Favor substitution costs.

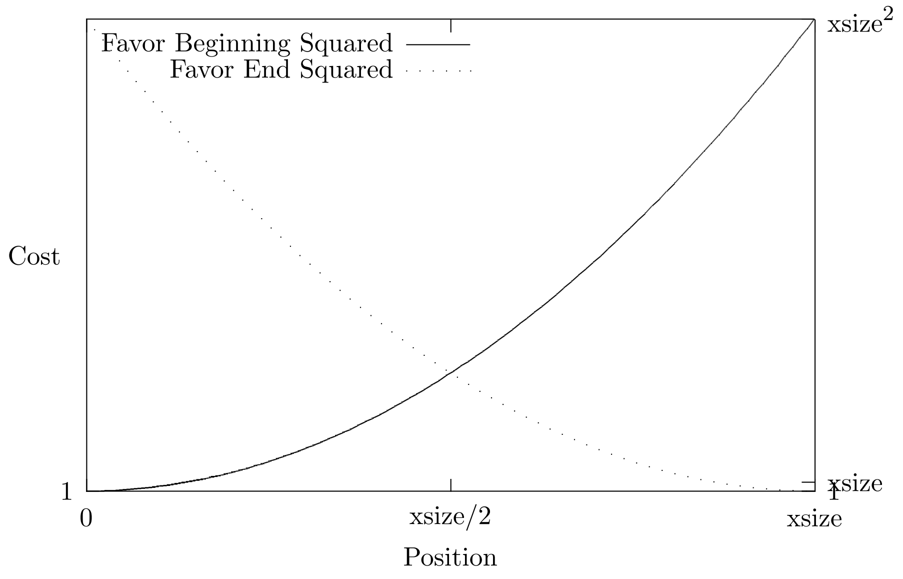


Figure 3-16: Favor squared insertion/deletion costs.

{**Todo #3.3:** Generate this.}

Figure 3-17: Favor squared edit costs.

substitutions for the favor end squared strategy. An example is in Figure 3-18.

The "favor end" and "favor end squared" algorithms are presented in Algorithm 3.5 and Algorithm 3.6

Algorithm 3.5 Favor end strategy

$$
\begin{aligned}
\textsc{Ins}(x,i) &= 1 + |x| - i \\
\textsc{Del}(x,i) &= 1 + |x| - i \\
\textsc{Sub}(x,i,y,j) &= \begin{cases} 0 & \text{if } x[i] = y[j], \\ 1 + \frac{|x|-i+|y|-j}{2} & \text{otherwise} \end{cases}
\end{aligned}
$$

3.4.2.2 Favor beginning strategy

The favor beginning strategy does the opposite of favoring the end (Algorithm 3.7). We can also square the values to drastically increase the difference

```
cost = 6
cost = 1 + 5^2= 26
                              1   ~  6
                              2   |  2
                              3   |  3
                              4   |  4
                              5   |  5

cost = 2
cost ^ 2 = 1 + 1^2 = 2
                              1   |  1
                              2   |  2
                              3   |  3
                              4   |  4
                              5   ~  6

cost = 7
cost^2 = 1 + 6^2 = 37
                              +      7
                              1   |  1
                              2   |  2
                              3   |  3
                              4   |  4
                              5   |  5
```

4

Figure 3-18: Favor end

Algorithm 3.6 Favor end squared strategy

$$\textsc{Ins}(x,i) = 1 + (|x| - i)^2$$
$$\textsc{Del}(x,i) = 1 + (|x| - i)^2$$
$$\textsc{Sub}(x,i,y,j) = \begin{cases} 0 & \text{if } x[i] = y[j], \\ 1 + \left(\frac{|x|-i+|y|-j}{2}\right)^2 & \text{otherwise} \end{cases}$$

(Algorithm 3.8). An example is in Figure 3.4.2.2.

Favoring changes at the beginning is appropriate if we want to assign more weight to frames which occur later in the call stack.

```
cost = 1
cost ^ 2 = 1
                             1   ~  6
                             2   |  2
                             3   |  3
                             4   |  4
                             5   |  5

cost = 5
cost ^ 2 = 1 + 4 * 4 = 17
                             1   |  1
                             2   |  2
                             3   |  3
                             4   |  4
                             5   ~  6

cost = 1
cost ^ 2 = 1
                             +      7
                             1   |  1
                             2   |  2
                             3   |  3
                             4   |  4
                             5   |  5
```

{**Todo #3.4:** Explanation}

3.4.3 Call stack-sensitive strategies

All of the call stack strategies multiply a factor based off the location of the operation (beginning, end) by the distance between the call frames.

The distance between call frames is determined by comparing the contents

Algorithm 3.7 Favor begin strategy

$$
\begin{aligned}
\textsc{Ins}(x, i) &= 1 + i \\
\textsc{Del}(x, i) &= 1 + i \\
\textsc{Sub}(x, i, y, j) &= \begin{cases} 0 & \text{if } x[i] = y[j], \\ 1 + \frac{i+j}{2} & \text{otherwise} \end{cases}
\end{aligned}
$$

Algorithm 3.8 Favor begin squared strategy

$$
\begin{aligned}
\textsc{Ins}(x, i) &= 1 + i^2 \\
\textsc{Del}(x, i) &= 1 + i^2 \\
\textsc{Sub}(x, i, y, j) &= \begin{cases} 0 & \text{if } x[i] = y[j], \\ 1 + \left(\frac{i+j}{2}\right)^2 & \text{otherwise} \end{cases}
\end{aligned}
$$

(class name, method name, method descriptor, and line number). The distance between two call frames with different class names is the biggest possible difference and is assigned a cost of 1000. If they have the same class name, but different method names, the cost is assigned 100. When the method names are the same, but the parameters or return type is different, the cost is 10. Finally, if the line numbers differ, the cost is 1. Comparison stops after the first mismatch: two same-named methods (`toString()`) in different classes are assigned 1000.

If the call frames are identical, the cost is a negative number: $-1000 \times factor$. This has the effect of preferring exact matches over changes.

Assigning costs in increasing order is intuitive: different classes are further apart than different methods in the same class; two methods in the same class, with the same name, but different parameters are further apart than two lines within the same method with the same parameters.

Using 1000/100/10/1 for the costs, however, is not as straightforward. Fundamentally, this is a heuristic.

There are approximately 2000 source files in the Java source code tree. The average length per source file is 300 lines. The Java source tree is heavily commented, so we assume a code to comment ratio of 1.0 and say that there are approximately 150 lines of code per source file.[5] The average number of methods per source file is 15, and so there is approximately 10 lines of code per method.

If we are on different lines of the same method (C.a(int):15 and C.a(int):20), then we assign a cost of 1 because we are barely different.

If we are in two overloaded versions of the same method (C.a(int) and C.a(double)), then we know that the lines will be different (by definition). Since there are approximately 10 lines per method, we assign a cost of 10.

If we are in the same class, but in two different methods (C.a and C.b), we assign a cost of 100. There are ten other methods in the class and each method has ten lines of code; we multiply and get a cost of 100. (Alternatively, there is approximately 100 lines of code in the class.)

If we are in different classes (C and D), then we assign a cost of 1000. One of the 10 methods in class C is going to be replaced with one of the 10 methods of class D, and one of those methods is going to have 10 lines of code; we multiply and get a cost of 1000.

3.4.3.1 Call stack strategy 1

The call stack strategy scales linearly and favors changes to the beginning: call stacks which start out the same are closer than those which do not (Algorithm 3.10).

[5]Only lines with code will show up in the call stack.

Cost	**Difference**	**Example**
1000	Class	`java.io.File` →`java.lang.String`
100	Method Name	`listFiles` →`createNewFile`
10	Method Parameters, Return Type	`File[] listFiles()` →`File[] listFiles(FilenameFilter filter)`
1	Line number	`File[] listFiles():852` →`File[] listFiles():854`

Table 3.6: Call stack operations and costs.

Algorithm 3.9 Compare-Frames(f, g) Comparing two stack frames

```
1: if f.class_name != g.class_name then
2:     return 1000
3: else if f.method_name != g.method_name then
4:     return 100
5: else if f.method_signature != g.method_signature then
6:     return 10
7: else
8:     return 1      // f.line != g.line
```

Algorithm 3.10 Call stack strategy 1

$$\begin{aligned}
\textsc{Ins}(x,i) &= 1000 \times (1+i) \\
\textsc{Del}(x,i) &= 1000 \times (1+i) \\
\textsc{Sub}(x,i,y,j) &= \left(1 + \frac{i+j}{2}\right) \times \begin{cases} -1 & \text{if } x[i] = y[j], \\ \textsc{Compare-Frames}(x[i], y[j]) & \text{otherwise} \end{cases}
\end{aligned}$$

3.4.3.2 Call stack strategy 2

Instead of favoring the beginning *or* the end, this favors both the beginning and the end. In other words, we penalize changes to the middle but are okay with changes at the beginning or end of the stack. For example, ten methods may call one important sequence of operations, which branches out into ten different functions. In other words, we care more about the items in the middle of the stack.

In some ways, this is appropriate for RMI calls: the RMI engine at the beginning of the stack dispatches operations to the application code and the application code uses the Java library. Which RMI function called the application code is not important, nor is the implementation within the Java library.

The relation between the position and factor is shown in Figure 3-19. The factor ranges from 1 to xsize/2; the maximum factor occurs when the position is xsize/2. Algorithm 3.11 describes the process.

Algorithm 3.11 Call stack strategy 2

$$
\begin{aligned}
\textsc{Ins}(x,i) =& 1000 \times \left(1 + \frac{|x|}{2} - \text{abs}\left(i - \frac{|x|}{2}\right)\right) \\
\textsc{Del}(x,i) =& 1000 \times \left(1 + \frac{|x|}{2} - \text{abs}\left(i - \frac{|x|}{2}\right)\right) \\
\textsc{Sub}(x,i,y,j) =& \frac{1}{2}\left(\frac{|x|}{2} - \text{abs}\left(i - \frac{|x|}{2}\right) + \frac{|y|}{2} - \text{abs}\left(j - \frac{|y|}{2}\right)\right) \times \\
& \begin{cases} -1 & \text{if } x[i] = y[j], \\ \textsc{Compare-Frames}(x[i], y[j]) & \text{otherwise} \end{cases}
\end{aligned}
$$

3.4.3.3 Call stack strategy 3

Call stack strategy 3 is similar to Call stack strategy 2, except that it will reach a peak of xsize at position xsize/2, instead of xsize/2 at xsize/2. Algorithm 3.12 defines the strategy.

Algorithm 3.12 Call stack strategy 3

$$
\begin{aligned}
\textsc{Ins}(x,i) =& 1000 \times 2 \times \left(1 + \frac{|x|}{2} - \text{abs}\left(i - \frac{|x|}{2}\right)\right) \\
\textsc{Del}(x,i) =& 1000 \times 2 \times \left(1 + \frac{|x|}{2} - \text{abs}\left(i - \frac{|x|}{2}\right)\right) \\
\textsc{Sub}(x,i,y,j) =& \left(\frac{|x|}{2} - \text{abs}\left(i - \frac{|x|}{2}\right) + \frac{|y|}{2} - \text{abs}\left(j - \frac{|y|}{2}\right)\right) \times \\
& \begin{cases} -1 & \text{if } x[i] = y[j], \\ \textsc{Compare-Frames}(x[i], y[j]) & \text{otherwise} \end{cases}
\end{aligned}
$$

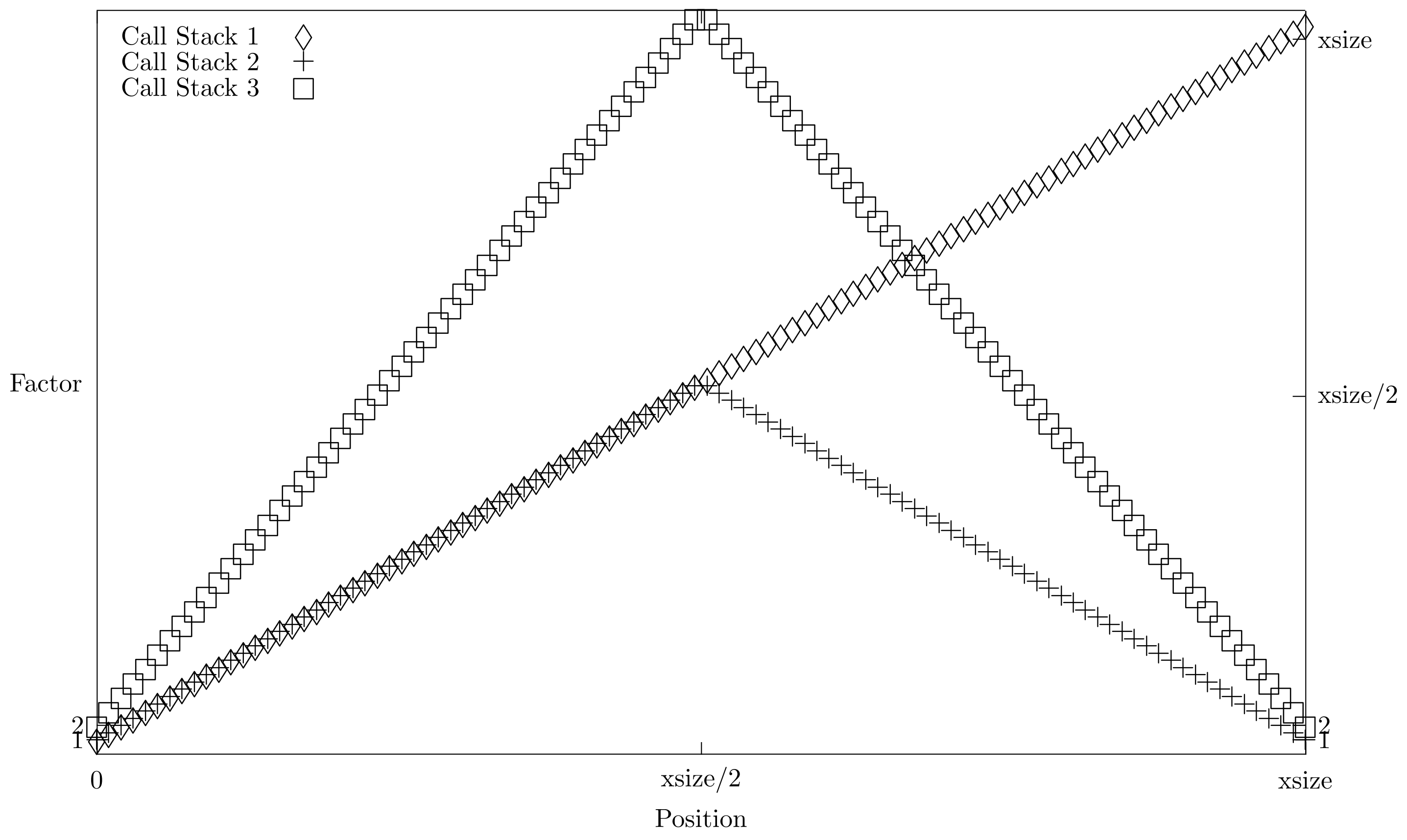


Figure 3-19: Call stack factors.

3.5 Gap distance

Instead of penalizing the deletion or insertion of letters, we can penalize the length of the gaps with an algorithm due to Gotoh [15] and presented in [6].

Let $D(i,j)$ be the score of an optimal alignment between $x_0x_1...x_i$ and $y_0y_1...y_j$ ending with deletions of letters of x. $I(i,j)$ indicates the score of an optimal alignment between $x_0x_1...x_i$ and $y_0y_1...y_j$ ending with insertions of letters of y. $T\left[i,j\right]$ is the score of an optimal alignment between $x_0x_1...x_i$ and $y_0y_1...y_j$. Let $\lambda(k)$ indicate the cost of a gap of length k. Then the computation of an optimal alignment (lowest distance) is done with the following recurrence formula [6]:

$$\begin{aligned} D(i,j) &= \min\left\{T[k,j] + \lambda(i-k) \mid k \in [0, i-1]\right\} \\ I(i,j) &= \min\left\{T[i,k] + \lambda(j-k) \mid k \in [0, j-1]\right\} \\ T\left[i,j\right] &= \min\left\{T[i-1,j-1] + \textsc{Sub}(x,\ i,\ y,\ j), D(i,j), I(i,j)\right\} \end{aligned}$$

If we do not restrict λ, then the cost of the optimal alignment can be found in $O(mn(m+n))$ time. However, if we let $\lambda(k) = g + h(k-1)$, where g is the cost of opening a gap, and h is the cost of widening the gap, we can solve the problem using Algorithm 3.13 in $O(mn)$ time with the recurrence [6, 15]:

$$\begin{aligned} D(i,j) &= \min\left\{D(i-1,j) + h, T\left[i-1,j\right] + g\right\} \\ I(i,j) &= \min\left\{I(i,j-1) + h, T\left[i,j-1\right] + g\right\} \\ T\left[i,j\right] &= \min\left\{T\left[i-1,j-1\right] + Sub(x_i, y_j), D(i,j), I(i,j)\right\} \end{aligned}$$

The intuition for using this for call stacks is that we want to favor long sequences over multiple insertions and deletions.

Algorithm 3.13 GAP-DISTANCE(x, y, *strategy*) (from [15, 6])

```
$m \leftarrow |x|$
$n \leftarrow |y|$
$g \leftarrow strategy.gap\_open$
$h \leftarrow strategy.gap\_widen$
for $i \leftarrow -1$ to $m - 1$ do
    $D[i, -1] \leftarrow \infty$
    $I[i, -1] \leftarrow \infty$
for $i \leftarrow -1$ to $n - 1$ do
    $D[-1, i] \leftarrow \infty$
    $I[-1, i] \leftarrow \infty$
$T[-1, -1] \leftarrow 0$
$T[-1, 0] \leftarrow g$
$T[0, -1] \leftarrow g$
for $i \leftarrow 1$ to $m - 1$ do
    $T[i, -1] \leftarrow T[i - 1, -1] + h$
for $i \leftarrow 1$ to $n - 1$ do
    $T[-1, i] \leftarrow T[-1, i - 1] + h$
for $i \leftarrow 0$ to $m - 1$ do
    for $j \leftarrow 0$ to $n - 1$ do
        $D[i, j] \leftarrow \min\{D[i - 1, j] + h, T[i - 1, j] + g\}$
        $I[i, j] \leftarrow \min\{I[i, j - 1] + h, T[i, j - 1] + g\}$
        $T[i, j] \leftarrow \min\{T[i - 1, j - 1] + \text{strategy.SUB}(x,i,y,j), D[i, j], I[i, j]\}$
```

3.5.1 Distance strategies

For our distance strategy, we use the Levenshtein strategy (see 3.4.1) with two additional fields: `gap_open` = 3 and `gap_widen` = 1.

3.6 Call stack and stack frame counting

We can also use a much simpler method of comparing executions: count the number of times unique call stacks (identical sequences of stack frames) and unique stack frames (classes, methods, and line numbers) occurred.

While intuitive and fast[6], this method does not consider the semantics of call stacks. It is useful as a basis for comparison, though: comparing counts is the most traditional way of doing execution profiling.

3.7 Cluster analysis

After running our analysis on the data, we have several comparison matrices: one for each strategy. For our edit and gap strategies, each row represents an execution and each column represents the difference/distance between the row and an execution. In the case of our simple count profiling, each row is an execution and each column is the number of times a stack frame or stack trace occurred.[7]) Our goal is now to cluster these executions using various settings in CLUTO [19].

[6]`Roxi` analyzes this approximately 10 times faster than the edit or gap approaches. The resultant files for clustering, however, have an average of 1000 columns, instead of the n columns (one for each execution) as in edit or gap strategies.

[7]In the sample applications, there were approximately 1500 stack frames which occurred and 4000 call stacks.

3.7.1 Parameters

There are five parameters to consider when we evaluate clustering algorithms.

3.7.1.1 Number of clusters

Obviously, the number of clusters is a large factor in determining the accuracy of the clustering approach. If there are too few clusters, normal and failed executions will be grouped together; if there are too many, then it is difficult to spot trends.

3.7.1.2 Clustering method

CLUTO supports five different clustering methods. Agglomerative hierarchial clustering was selected as the clustering method due to the advantage in execution speed.

The desired k-way clustering solution is computed using the agglomerative paradigm: locally optimize (minimize or maximize) a particular clustering criterion function. The solution is obtained by stopping the agglomeration process when k clusters are left. [19]

3.7.1.3 Clustering criterion function

Three clustering criterion functions were used.

Single-link In single-link clustering (also called the connectedness or minimum method), we consider the distance between one cluster and another cluster to be equal to the shortest distance from any member of one cluster to any member of the other cluster [5].

Complete-link In complete-link clustering (also called the diameter or maximum method), we consider the distance between one cluster and another cluster to be equal to the longest distance from any member of one cluster to any member of the other cluster [5].

UPGMA In this method, the distance between two clusters is calculated as the average distance between all pairs of objects in the two different clusters. This method is also very efficient when the objects form natural distinct "clumps," however, it performs equally well with elongated, "chain" type clusters.

{**Todo #3.5:** Why these?}

3.7.1.4 Scaling each row

We might also want to apply scaling to each row; there are two options for this.

none No scaling is performed.

sqrt The columns of each row are scaled to be equal to the square-root of their actual values [19]. Let:

$$\operatorname{sign}(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \end{cases}$$

$$r'_{i,j} = \operatorname{sign}(r_{i,j})\sqrt{r_{i,j}}$$

3.7.1.5 Similarity

The similarity between objects is computed using the cosine function: the dot product divided by their magnitudes:

$$CS_{xy} = \frac{\sum_i (x_i y_i)}{\sqrt{\sum_i {x_i}^2 \sum_j {y_j}^2}} \tag{3.1}$$

Chapter 4

CASE STUDY

Overview

The motivating factor for this work is to find a way of examining large sets of executions in a distributed system in order to find failures. I will outline two realistic sample applications which use classical distributed systems algorithms. Then I will introduce application-level defects and perform fault injection similar to faults in the real world. A large set of executions will be analyzed with the strategies and clustering methods discussed in Chapter 3 and discuss the effectiveness of each method.

4.1 Possibilities

I examined several Java distributed systems before deciding to write my own sample application.

4.1.1 ECPerf

ECPerf is the J2EE benchmark kit: it has an end-to-end scenario involving Enterprise JavaBeans, databases, servlets, etc. However, the license agreement prohibits modification of the source code, discussing the results, or the design of the system.

4.1.2 jBoss

jBoss is a J2EE container for Enterprise JavaBeans [17]. While it is open source, and comes with some unit tests, it is suboptimal for an example for several reasons.

- Very large system/difficult to conceptualize. Consisting of over 1900 Java source code files and 40,000 lines of code, it is very difficult to conceptualize the relationship between so many classes.
- Test set. Only 150 unit tests exist and these are mostly "toy" programs which are intended only to check the most basic functionality for a short period of time.
- Not a real world application. jBoss is a framework for writing other applications. Most people write applications, not frameworks.
- Lack of VM's. Out of the box, the jBoss unit tests run on two VM's: the jBoss application server (which runs the database as well) and the test runner. It is possible to run the database on another VM, but the tests are centered on a single-machine.
- Uses raw network sockets traffic in some situations. jBoss has a JDBC driver which uses raw sockets to communicate with the database; this is not RMI.

- Classloading. jBoss uses its own assembler[1] in order to generate proxies; this makes it difficult to trace code: `Proxy$0`, `Proxy$1`, etc.
- Fault injection. It was difficult to find valid places to insert faults into the jBoss source code. Comparing to old versions (2.2 and 2.4.1 versus 2.4.4) at the source code and CVS level did not give any guidance.

4.1.3 Jini Technology Core Platform Compatibility Kit

The Jini TCK was also examined. It contains 20 tests (divided among service, client, and lookup service categories) which are intended to test if a Jini service conforms to the Jini Technology Core Platform Specification. Namely, is this service a good citizen with respect to the Jini world?

However, it generated very little data. Most of the tests for Jini services are time-based: x must do y before time t. Therefore, the TCK spends a large portion its time waiting, with very actual code executing. I also compared different versions of Jini (1.1, 1.2) to the TCK (1.1B, 1.2A), with no clear success.

4.1.4 Applications selected

Due to these problems, I wrote two sample applications which use classical distributed systems algorithms: Bully and Byzantine.

4.2 Bully application

"Bully" is a distributed system which searches for large prime numbers, similar to `http://www.distributed.net`. Each machine has a static priority and the

[1]`org.jboss.proxy.ProxyAssembler`

machine with the highest priority is elected the leader/coordinator using the bully distributed election algorithm, from Garcia-Molina [14].

The clients connect to the leader and request tasks; clients work on the task and notify the coordinator with the results of the computation (whether or not the number is prime). The search space and results are replicated throughout the system so another machine can take over if necessary. All communication uses Java RMI and most requests go through a shared, reliable message server instead of between clients.[2]

- Multiple virtual machines. Bully requires a message server, a coordinator, and several client VMs.
- Well-defined set of behaviors makes the system easier to conceptualize. Clients join the message server, start an election, get tasks, publish results, etc. It easy to see that several transitions do not make sense: sending results to a non-coordinator, etc.
- Non-deterministic. While the jBoss unit tests may have different thread ordering, Bully has different outcomes: Client 1 may handle many tasks or very few of them.
- Heavy use of RMI. Clients use RMI to invoke methods on the message server; the message server uses RMI to invoke methods on other clients, etc.
- Non-trivial. Bully uses lots of threads: worker threads, RMI handling, timeouts, etc.

[2]This simplifies the logic from the client perspective, yet is a realistic "middleware" approach.

4.2.1 Architecture

Bully consists of approximately 4000 lines of Java code. There are four main packages: bully.algorithm, bully.client, bully.coordinator, and bully.messaging.

bully.algorithm Various shared items.

bully.client The actual client which connects to everything.

bully.coordinator All coordinator-related classes: Task, Results, etc.

bully.messaging The reliable message server is in here.

4.2.2 Execution flow

4.2.2.1 Starting up

1. A client comes up and looks up the Messaging Server.
2. The client registers itself with the Messaging Server.
3. The client starts an election.
4. At some point, the leader/coordinator has been decided.
5. The client requests a number from the coordinator and tests it for primality.
6. The client notifies the coordinator with the result.
7. The coordinator will tell the client how many work items are left.

{**Todo #4.6:** Architecture picture}

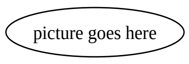


Figure 4-20: Bully architecture

4.2.2.2 Election details

Although very similar to Garcia-Molina's paper, this uses a centralized server, which simplifies the client code.

{**Todo #4.7:** Election algorithm}

{**Todo #4.8:** Sequence diagram}

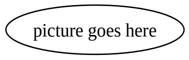


Figure 4-21: Sequence diagram for election

{**Todo #4.9:** Sequence diagram}

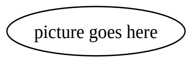


Figure 4-22: Sequence diagram for prime number processing

4.2.3 Specification

Fundamentally, there are three requirements:

1. Communication is reliable and robust to network failures.
2. The system shall eventually reach a consistent state and have one leader (coordinator).
3. All numbers will be processed within a reasonable amount of time.[3]
4. Clients will not attempt to take over.

Violations of these conditions result in catastrophic failure: a VM may crash/exit, no progress is made in the election algorithm, multiple coordinators exist, etc. The common (90 – 99%) case is that nothing goes wrong.

4.3 Byzantine application

The Byzantine application uses the Byzantine Generals algorithm, described by Lamport et al. in [24], to decide whether or not the participants should attack (attempt to factor a prime number) or retreat (exit without doing anything). The complication is that the commanding general and/or any of the other generals may be unloyal and relay the wrong message.

4.3.1 Description of problem

4.3.2 Algorithm

4.3.3 Implementation

Consists of approximately 3500 lines of Java code.

Packages: ...

[3]The time limit is three times the normal execution time.

4.4 Fault injection

4.4.1 Failure-inducing behaviors

We are interested in failure-inducing behavior specific to distributed systems. For this, we consider two main classes:

Detectable errors in communication A "detectable" error is one which is detected via a Java `IOException`. For example, a socket may close early, be reset by a peer, timeout, be unable to connect, have issues receiving the data, have issues sending the data, etc.

Undetectable errors in communication An "undetectable" error is an error which does not fail loudly: a large network delay, or the random swapping of bytes within a packet by an attacker are two examples. Sometimes these will cause exceptions in the future (for example, a method hash value has been modified) and other times they will not (the value of a parameter is changed).

4.4.2 Injection

In order to make this as generalizable as possible, it is desirable to minimize the number of changes necessary to the environment. There are two places where fault-inducing code will be injected:

1. `java.net.SocketInputStream` which handles input from a socket
2. `java.net.SocketOutputStream` which handles output from a socket

The fault injection code will:

1. Decide whether or not a fault should be injected

2. If so, then write a message to a log indicating that a fault will be injected, and
    (a) Throw an exception, or
    (b) Randomly swap bytes in the input/output
3. Otherwise, proceed as normal

4.5 Application-level issues

Application-level violations of the program specification are considered to be worthy of further inspection. For example, a program could change its priority and take over as the coordinator, or a general could send conflicting messages to other generals.

4.5.1 Bully – priority elevation

After a period of time, one of the clients will perform the following events:

- Unregister from the Message Server
- Change its priority to 200
- Re-register with the Message Server
- Perform the "recovery" step as specified in the algorithm

This code was inserted without modifying existing line numbers. (Changing line numbers would be a confounding effect.)

4.5.2 Byzantine – unloyal generals

As described in the Byzantine Generals paper, generals (including the commander) can be unloyal or traitorous: they will either not answer or send the wrong answer.

In order to test `Ixor`'s ability to detect failed executions, I generated a set of 277 "normal" executions via `Ixor` where the fault injection framework is present, but no faults are injected. Then, I generated 15 fault-injected executions (5%) which exhibited failures as described above.

I ran `Roxi` against this large (over 1.4 gigabyte) data set and generated dissimilarity matrices (as described in Chapter 3) for each strategy. This was then analyzed with CLUTO using the clustering algorithms discussed in 3.7.

4.5.3 Faults injected

Table 4.7 gives the number of faults injected into the executions. Execution 7008 was manually terminated early in order to simulate a massive simultaneous failure: a power failure or denial of service attack, for example. The probability of an IO exception or random byte swap was set to $P = 0.001$.

{**Todo #4.10:** Explain results} {**Todo #4.11:** Byzantine failures}

4.6 Evaluating the approach

4.6.1 Sampling methods

After generating all the clustering combinations, I created a program to perform the following types of cluster sampling on the clusters.

Fault count	**Execution identifier**
3	6000
1	6001
2	6002
6	6003
2	6004
5	6005
3	7000
3	7001
9	7002
5	7003
4	7004
8	7005
9	7006
4	7007
0	7008

Table 4.7: Fault count

4.6.1.1 Random sampling

The easiest sampling method is to randomly sample m executions from the entire population.

4.6.1.2 1-per cluster sampling

The 1-per-cluster sampling method selects one execution at random from each cluster. Hence, the number of executions to be checked is equal to the total number of clusters. Since small clusters were typically more common than large ones in our experiments and since executions with unusual profiles are found in small clusters, this method favored the selection of such executions.

4.6.1.3 n-per cluster sampling

The n-per-cluster sampling method is a generalization of one-per cluster sampling. It selects a fixed number n of executions from each cluster. If there are fewer than n executions in a cluster, then all of them are selected. The number of executions selected by this method depends on the distribution of cluster sizes; the maximum is n times the total number of clusters.

n-per-cluster sampling has a greater chance of finding a failure in a cluster that also contains successful executions than does 1-per-cluster sampling.

4.6.1.4 Small-cluster sampling

The small-cluster sampling method selects executions exclusively from small clusters. The sample size m is chosen first. The clusters are then formed into groups composed of all clusters of the same size. Starting with the group of smallest clusters, executions are selected at random and without replacement from the clusters in the current group. If the executions in a group of clusters are exhausted without reaching a total of m executions, the group with the next larger cluster size is sampled. This process continues until m executions are selected.

4.6.1.5 Adaptive sampling

The adaptive sampling method augments the sample when a failure is found, to seek additional failures in its vicinity. Adaptive sampling proceeds in two phases. First, one execution is selected at random from each cluster, as in 1-per-cluster sampling, and the selected executions are checked for conformance to requirements. Then, for each failure found, all other executions in its cluster are selected and checked. The number of additional executions selected depends on the distribution of failures among clusters. Adaptive sampling should be beneficial if failures tend to cluster together.

Adaptive sampling outperformed the other methods: it had a higher failure detection rate at lower sample sizes than the other methods. (This confirms the finding in [13].)

4.7 Results

4.7.1 Distribution of failures

We want to verify that failures are not uniformly distributed. We do this by comparing the percentage of failures p_f found in the smallest x percent of clusters to the expected percentage of failures x in a uniform distribution (x).

Table 4.8 shows that the percentage of failures is significantly larger than would be expected for a uniform distribution.

```
bully02.avg        8.90056
bully05.avg       16.5997
bully10.avg       29.4211
bully25.avg       53.2867
bully50.avg       75.7643

byz02.avg        5.99647
byz05.avg       14.1174
byz10.avg       26.6595
byz25.avg       55.5556
byz50.avg       76.5833
```

Table 4.8: Percentage of failures found in the subpopulations contained in the smallest clusters.

4.7.2 Singletons

Figure 4-23 shows the relationship between failures in singleton clusters and all executions in singleton clusters. With the Bully program, on average, over all clustering methods, 14% of failures were found in singleton clusters but only 5.7% of the total executions were in singleton clusters. The Byzantine program had 15% of failures in singleton clusters on average, with only 6% of total executions in singleton clusters. This confirms our intuition that since

```
Bully:
 percentage of failures in singleton clusters
      (averaged over all clustering methods)
        Median:         9.09091
        Average:        14.2399

 percentage of all executions in singleton clusters
      (averaged over all clustering methods)
        Median:         2.67559
        Average:        5.73334

Byzantine:
 percentage of failures in singleton clusters
      (averaged over all clustering methods)
        Median:         0
        Average:        15.1996

 percentage of all executions in singleton clusters
      (averaged over all clustering methods)
        Median:         3.125
        Average:        6.13313
```

Figure 4-23: Singleton failures compared to singleton normal executions.

failures have unique profiles, they will tend to be in singleton clusters.

4.7.3 Purity

The measure of "purity" is presented in [63]; it measures the "extent to which each cluster contained documents from primarily one class." (Instead of documents, we are looking at executions. Also, there are only two classes: normal executions and failed executions.) Let S_r be a cluster of size n_r. n_r^i is the number of documents of the ith class which were assigned to the rth cluster. From [63]:

Purity of a cluster S_r is defined to be

$$P(S_r) = \frac{1}{n_r} \max_i \left(n_r^i\right) \tag{4.2}$$

which is the fraction of the overall cluster size that the largest class of documents[4] assigned to that cluster represents. The overall purity of the clustering solution is obtained as a weighted sum of the individual cluster purities and is given by

$$Purity = \sum_{r=1}^{k} \frac{n_r}{n} P(S_r) \tag{4.3}$$

In general, the larger the values of purity, the better the clustering solution is.

Ideally, we would have a purity value of 1.0, indicating that each cluster consisted of only one class of executions. Purity is useful for verifying how effective it our clustering is, but it is not useful from a practictioner standpoint because the classes are unknown. However, if we know that the purity for a clustering technique is very high (close to 1.0), then, in the future, we could reasonably apply 1-per-cluster sampling (instead of adaptive sampling) to infer the class of the every execution in the cluster.

Table 4.9 shows the results for the Bully executions and Table 4.10 shows the results for the Byzantine executions.

4.7.4 Average percentage of failures

Using the adaptive sampling method and averaging across the six clustering settings, we can find the average percentage of failures found.

Table 4.11 shows the results for the Bully executions and Table 4.12 shows the results for the Byzantine executions.

[4]In our case, executions: whether or not an execution is a normal or a failure.

Clusters	frame	stack	CS 1	CS 2	CS 3	beg^2	beg	end^2	end	edit lev	gap lev
3	0.932389	0.934463	0.944481	0.944889	0.942222	0.933611	0.950852	0.952667	0.961111	0.958056	0.957111
8	0.950093	0.948944	0.966611	0.960889	0.958889	0.962019	0.981519	0.961667	0.964556	0.984981	0.987315
15	0.961019	0.958278	0.976519	0.962333	0.960111	0.977352	0.99087	0.96287	0.965926	0.991315	0.994426
30	0.967463	0.969222	0.980704	0.964741	0.961759	0.988685	0.993741	0.963593	0.969333	0.995056	0.997204
45	0.970685	0.975074	0.983519	0.967704	0.964148	0.993889	0.99637	0.965037	0.973074	0.996981	0.99837
60	0.972963	0.979852	0.986722	0.970259	0.966352	0.995519	0.997148	0.966111	0.975352	0.998833	0.999111
75	0.975778	0.982907	0.989	0.973426	0.968722	0.997667	0.998333	0.967093	0.977833	0.999444	0.999833
90	0.977815	0.985407	0.990519	0.97637	0.972	0.998315	0.999333	0.968259	0.980593	0.999833	0.999833

Table 4.9: Average purities for Bully.

Clusters	frame	stack	CS 1	CS 2	CS 3	beg^2	beg	end^2	end	edit lev	gap lev
3	0.960704	0.964556	0.953	0.954315	0.953444	0.953	0.953148	0.962315	0.965593	0.965296	0.965296
6	0.96337	0.968815	0.954778	0.961574	0.958185	0.953	0.959222	0.98513	0.980426	0.972444	0.971463
13	0.965426	0.974759	0.962296	0.974889	0.971056	0.953148	0.963815	0.998667	0.992519	0.978981	0.977852
19	0.966741	0.97913	0.969704	0.978444	0.975815	0.953593	0.969148	0.998667	0.996481	0.982944	0.981204
26	0.9695	0.98087	0.974333	0.979315	0.977944	0.955667	0.971056	0.999556	0.998389	0.986333	0.984426
32	0.972648	0.983315	0.97587	0.982574	0.979519	0.957444	0.973037	1	0.998667	0.989907	0.987481
38	0.975444	0.984537	0.977296	0.983815	0.981185	0.960704	0.974426	1	0.999704	0.993056	0.990352

Table 4.10: Average purities for Byzantine.

4.8 Conclusion

{**Todo #4.12:** Why did these methods work best?}

`Ixor` was run on two non-trivial distributed applications running across four virtual machines.

Clusters	frame	stack	CS 1	CS 2	CS 3	beg^2	beg	end^2	end	edit lev	gap lev
3 (1%)	13.4574	14.8463	31.937	33.0778	29.2074	15.8426	40.7759	39	48.3852	46.9722	46.713
8 (2.5%)	36.2944	36.9833	58.1611	49.9704	47.2296	53.7519	76.4907	50.1389	54.4833	82.1444	86.6222
15 (5%)	51.8852	49.337	70.7407	53.1	49.3444	72.4407	88.8056	52.1889	57.1426	90.4019	93.1537
30 (10%)	59.9389	62.3944	77.5426	57.0778	52.1	86.4	92.8667	54.6352	63.4204	94.35	96.9278
45 (15%)	64.9111	70.9852	80.9259	62.3389	56.9352	92.4019	96.2648	57.8759	69.5315	96.8185	98.3537
60 (20%)	67.2907	77.3778	85.2685	67.6704	61.2296	94.6444	97.3519	60.1907	73.1556	98.8167	99.0648
75 (25%)	71.4574	80.9407	88.1222	72.3907	66.25	97.3463	98.4778	63.7778	77.1074	99.4537	99.8519
90 (30%)	74.9481	83.6556	89.7333	76.1796	70.713	98.2259	99.4167	66.5537	80.3926	99.7963	99.8519

Table 4.11: Bully average percentage of failures detected with adaptive sampling.

Clusters	frame	stack	CS 1	CS 2	CS 3	beg^2	beg	end^2	end	edit lev	gap lev
3 (2.5%)	21.337	27.2426	4.46852	8.46296	7.82037	8.54259	9.1537	30.5111	33.1722	32.9056	30.3037
6 (5%)	27.1889	36.1	9.08148	25.4907	17.2204	8.98704	22.9296	72.4148	61.5907	45.5111	46.6889
13 (10%)	34.3648	49.8685	27.8556	50.0426	43.387	14.4444	35.7852	97.7426	86.65	61.4074	58.0648
19 (15%)	38.75	59.6056	43.6944	59.6037	55.3889	17.187	43.9222	98.2685	94.4481	71.2148	65.8407
26 (20%)	44.0463	64.1111	53.1444	63.9463	59.387	25.5056	49.8889	99.5278	97.9556	78.1056	74.8685
32 (25%)	51.7648	68.6278	57.6167	69.7926	64.437	30.7667	54.8426	100	98.5204	85.9519	81.3389
38 (30%)	58.4222	71.2352	62.6963	73.1111	68.5407	36.7148	58.7444	100	99.6222	90.6574	87.2352

Table 4.12: Byzantine average percentage of failures detected with adaptive sampling.

Chapter 5

CONCLUSION

5.1 Conclusion

I have presented a method of collecting and analyzing stack traces from virtual machines in a distributed Java system. Novel ways of comparing executions were applied to a non-trivial sample application and the effectiveness of various strategies and clustering algorithms showed that this approach can distinguish between normal and failed executions.

5.2 Future directions

With some modifications, this approach may be applicable to single-machine systems, as well.

5.3 Issues with current work

Since `Ixor` operates under the assumption that the network is unreliable, the best practice for setting up `Ixor` is to put two network cards in each machine: one going to the `Ixor` server and another going to the world.

There is no security built into `Ixor` at all. Also, for reasons specified in 2, it is not cross-platform.

`Ixor` must compare identical versions of the software: no line numbers can change.

Unlike call profiling, we can't get use the descriptive/discriminating features which caused the program to be clustered like that. In other words, we have fewer leads.

Appendix A

XDPROF PERFORMANCE TESTING

We used an Intel Pentium II 350 MHz with 512 MB of RAM running Windows 2000 Professional for our testing. The web pages were served off of a machine running FreeBSD and the Apache web server. The low geometric mean in Table A.16 is the SPECjvm98_base score; the high geometric mean is the SPECjvm98 score. We used the `timethis` utility[34] against a special build of xdProf for Table A.17 and A.18.

Table A.13: SPECjvm98 results for Sun Java 2 Runtime Environment 1.3 with HotSpot Client VM.

	SPECjvm98	**SPECjvm98_base**
Without xdProf	22.0	18.3
With xdProf, local machine, 100 millisecond updates	20.8	18.6
With xdProf, remote, 100 millisecond updates	21.4	17.4

Table A.14: Elapsed time on Sun 1.3 HotSpot Client VM (without xdProf = 383.136 seconds).

Refresh (milliseconds)	**Local time (seconds)**	**Local overhead**	**Remote time (seconds)**	**Remote overhead**
100	412.863	7.76%	395.258	3.16%
200	402.338	5.01%	390.531	1.93%
1000	398.202	3.93%	392.143	2.35%

Table A.15: Elapsed time on Sun 1.3 Classic VM (without xdProf = 3600.477 seconds).

Refresh (milliseconds)	**Local time (seconds)**	**Local overhead**	**Remote time (seconds)**	**Remote overhead**
100	4352.799	20.90%	3680.101	2.21%
200	4263.951	18.43%	3639.373	1.08%
1000	4207.349	16.86%	3601.698	0.03%

Table A.16: SPEC ratios for Sun 1.3 HotSpot Client VM.

	without xdProf		**with xdProf local machine 100 milliseconds**		**with xdProf remote machine 100 milliseconds**	
Benchmark	*Low*	*High*	*Low*	*High*	*Low*	*High*
_227_mtrt	22.10	25.30	21.80	24.20	21.10	25.10
_202_jess	22.40	31.30	23.50	29.10	17.90	30.90
_201_compress	11.80	13.40	12.70	12.80	9.36	12.90
_209_db	12.90	13.80	12.60	13.10	13.20	13.20
_222_mpegaudio	32.80	35.40	32.00	33.70	32.70	34.50
_228_jack	30.40	38.40	30.50	36.60	30.60	36.60
_213_javac	9.15	12.40	9.62	11.60	10.10	12.30
Geometric Mean	18.30	22.00	18.60	20.80	17.40	21.40

Table A.17: Network traffic details for Sun 1.3 HotSpot Client VM (without xdProf = 383.136 seconds).

Server	**Refresh (ms)**	**Time (seconds)**	**Overhead**	**Traces**	**Total Bytes**	$\frac{\textbf{Total Bytes}}{\textbf{Traces}}$	$\frac{\textbf{Time}}{\textbf{Traces}}$
local	100	412.863	7.76%	3860	11133599	2884.352	106.959
local	200	402.338	5.01%	1927	5507828	2858.240	208.790
local	1000	398.202	3.93%	391	1124780	2876.675	1018.419
remote	100	395.258	3.16%	3731	10608305	2843.287	105.939
remote	200	390.531	1.93%	1874	5348439	2854.023	208.394
remote	1000	392.143	2.35%	386	1102655	2856.619	1015.915

Table A.18: Network traffic details for Sun 1.3 Classic VM (without xdProf = 3600.477 seconds).

Server	**Refresh (ms)**	**Time (seconds)**	**Overhead**	**Traces**	**Total Bytes**	$\frac{\textbf{Total Bytes}}{\textbf{Traces}}$	$\frac{\textbf{Time}}{\textbf{Traces}}$
local	100	4352.799	20.90%	40974	178144602	4347.747	106.233
local	200	4263.951	18.43%	20536	89289586	4347.954	207.633
local	1000	4207.349	16.86%	4174	18183230	4356.308	1007.990
remote	100	3680.101	2.21%	35230	152744536	4335.638	104.459
remote	200	3639.373	1.08%	17646	76579076	4339.741	206.244
remote	1000	3601.698	0.03%	3578	15630092	4368.387	1006.623

Bibliography

[1] D. Abramson and R. Sosic. A debugging and testing tool for supporting software evolution. *Automated Software Engineering: An International Journal*, 3(3/4):369–390, August 1996.

[2] Ron Baecker, Chris DiGiano, and Aaron Marcus. Software visualization for debugging. *Communications of the ACM*, 40(4):44–54, 1997.

[3] Albert Benveniste, Eric Fabre, and Stefan Haar. Markov nets: Probabilistic models for distributed and concurrent systems. Technical Report 1415, Institut de Recherche en Informatique et Systèmes Alétoires, September 2001.

[4] Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, and Beth Stearns. *The J2EE Tutorial.* Sun Microsystems, 2002.

[5] Stephen P. Borgatti. How to explain hierarchical clustering. Web, 1994. `http://www.analytictech.com/networks/hiclus.htm`.

[6] Christian Charras and Thierry Lecroq. Sequence comparison. `http://www-igm.univ-mlv.fr/~lecroq/seqcomp`, February 1998. LIR (Laboratoire d'Informatique de Rouen) et ABISS (Atelier Biologie Informatique Statistique Socio-linguistique).

[7] Wing Hong Cheung, James P. Black, and Eric Manning. A framework for distributed debugging. *IEEE Software*, 7(1):106–115, January 1990.

[8] Max Copperman. Producing an accurate call-stack trace in the occasional absence of frame pointers. Technical Report UCSC-CRL-92-25, UCSC, 1992.

[9] Max Copperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, May 1994.

[10] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems – Concepts and Design*. Addison-Wesley, 2001.

[11] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczk. Compiling Java just in time. *IEEE Micro*, 17:36–43, May – June 1997.

[12] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling java just in time. *IEEE Micro*, 17:36–43, May – June 1997.

[13] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering*, pages 246–255. ACM Press, 2001.

[14] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):49–59, January 1982.

[15] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[16] D. Griswold. The Java HotSpot Virtual Machine Architecture, 1998. http://java.sun.com/products/hotspot/whitepaper.html.

[17] jBoss. jBoss. http://www.jBoss.org/.

[18] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing message patterns in object-oriented program executions. Technical Report 96-15, Georgia Institute of Technology, May 1996.

[19] George Karypis. CLUTO: CLUstering TOolkit. http://www-users.cs.umn.edu/∼karypis/cluto/, April 2002. Version 2.0. University of Minnesota, Department of Computer Science.

[20] I. H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. A. Konstan, D. J. Lilja, and P.-C. Yew. JaViz: A client/server Java profiling tool. *IBM Systems Journal*, 39(1):96–117, 2000.

[21] John Lambert and Andy Podgurski. xdProf: A tool for the capture and analysis of stack traces in a distributed Java system. In *Proceedings of the 2001 SPIE Conference*, 2001.

[22] L. Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, Reading, 1994.

[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, July 1978.

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[25] David Leon, Andy Podgurski, and Lee J. White. Multivariate visualization in observation-based testing. In *Proceedings of the 22nd international conference on Software engineering*, pages 116–125. ACM Press, 2000.

[26] I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics-Doklady*, 6:707–710, 1966.

[27] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[28] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IEEE Proceedings of the 1998 Int. Workshop on Program Understanding (IWPC'98)*, 1998.

[29] D. Manivannan, Robert H. B. Netzer, and Mukesh Singhal. Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):623–627, 1997.

[30] Masoud Mansouri-Samani and Morris Sloman. Monitoring distributed systems (a survey). Technical Report DOC92/23, Imperial College, September 1992.

[31] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[32] David Melski and Thomas W. Reps. Interprocedural path profiling. In *Computational Complexity*, pages 47–62, 1999.

[33] Nabor C. Mendonca and Jeff Kramer. Component module classification for distributed software understanding.

[34] Microsoft Corporation. *Microsoft Windows 2000 Professional Resource Kit*. Microsoft Press, 2000.

[35] Sun Microsystems. Java remote method invocation. `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`, October 1998. Revision 1.50, JDK 1.2.

[36] Stephen C. North and Eleftherios Koutsofios. Application of graph visualization. In *Proceedings of Graphics Interface '94*, pages 235–245, Banff, Alberta, Canada, 1994.

[37] Numega. TrueTime and TrueCoverage.

[38] B. J. Oommen, K. Zhang, and W. Lee. Numerical similarity and dissimilarity measures between two trees. *IEEE Transactions on Computers*, 45(12):1426–1434, December 1996.

[39] B. John Oommen and R. K. S. Loke. Noisy subsequence recognition using constrained string editing involving substitutions, insertions, deletions and generalized transpositions. In *ICSC*, pages 116–123, 1995.

[40] Open Source Initiative. The BSD License.

[41] Andy Podgurski, Wassim Masri, Yolanda McCleese, Francis G. Wolff, and Charles Yang. Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodology*, 8(3):263–283, July 1999.

[42] Andy Podgurski and Charles Yang. Partition testing, stratified sampling, and cluster analysis. In *Proceedings of the first ACM symposium on Foundations of software engineering*, pages 169–181. ACM Press, 1993.

[43] PreEmptive Solutions. DashOPro.

[44] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Lisboa 92 - An Advanced Course on Distributed Systems*, 1992.

[45] Sitraka Software. JProbe. `http://www.klgroup.com`.

[46] @stake Research Labs. `netcat` 1.1 for Win 95/98/NT/2000.

[47] Standard Performance Evaluation Corporation. SPECjvm98, 1998. `http://www.spec.org/osg/jvm98/`.

[48] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[49] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 158–167. ACM Press, 2000.

[50] Scott D. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[51] Sreenivas Subhash. Independent global snapshots in large distributed systems, 1997. 4th International Conference on High Performance Computing December 18-21, 1997 - Bangalore, India.

[52] Sun Microsystems. Forte for Java `http://www.sun.com/forte/ffj`.

[53] Sun Microsystems. Java Virtual Machine Profiler Interface Documentation. `http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html`.

[54] C. Tice and S. Graham. Key instructions: Solving the code location problem for optimized code. Technical Report 164, Compaq Systems Research Center, September 2000.

[55] Giovanni Vigna. Protecting mobile agents through tracing. In *3rd ECOOP Workshop on Mobile Object Systems*, Jyväskylä, Finland, 1997.

[56] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Systems Journal*, 39(1):82–95, 2000.

[57] VMGEAR. Optimizeit.

[58] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 271–283, Vancouver, British Columbia, Canada, 18–22 October 1998. ACM Press. Published as ACM SIGPLAN Notices 33(10), October1998.

[59] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient mapping of software system traces to architectural views. Technical Report TR-00-09, Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, British Columbia, Canada V6T 1Z4, 7 July 2000.

[60] Hendra Widjaja and Michael Oudshoorn. Visualisation of concurrent and object-oriented programs.

[61] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 326–337, 1993.

[62] I. Yu. Integrating event visualization with sequential debugging. Master's thesis, University of Waterloo, 1996.

[63] Ying Zhao and George Karypis. Criterion functions for document clustering: Experiments and analysis. Technical Report 01-40, University of Minnesota, Department of Computer Science / Army HPC Research Center, February 2002.