A Distributed System Testing Framework

by

Fiona Warman
B.Eng., University of Victoria, 2006

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Supervisory Committee

A Distributed System Testing Framework

by

Fiona Warman
BEng, University of Victoria, 2006

Supervisory Committee

Dr. Stephen Neville, Department of Electrical and Computer Engineering
Supervisor

Dr. Kin Li, Department of Electrical and Computer Engineering
Departmental Member

Dr. Daniel Hoffman, Department of Computer Science
Outside Member

Abstract

Supervisory Committee
Dr. Stephen Neville, Department of Electrical and Computer Engineering
Supervisor
Dr. Kin Li, Department of Electrical and Computer Engineering
Departmental Member
Dr. Daniel Hoffman, Department of Computer Science
Outside Member

Distributed systems are becoming increasingly common. However, the testing of these systems is difficult due to their non-linear, stochastic and dynamic behaviours, and limited application-level testing support. In this thesis, a prototype cluster computing-based test harness has been developed that can be used for performance testing on a variety of distributed systems. Its usefulness is demonstrated through tests conducted on an example distributed system, including using the test harness to perform a parameter search on the system in an iterative fashion.

Table of Contents

List of Tables

List of Figures

Dedication

To my parents

Chapter 1: Introduction and Prior Art

1.0 Introduction

Distributed computing solutions are becoming more common as software systems are developed which require a large number of resources. Distributed systems are often designed to provide greater fault tolerance or to process large amounts of data much faster than could be accomplished on a single machine. The machines in a distributed system run a variety of software, unlike grid computing which generally uses the same software running on a large number of machines. Distributing a system across multiple low-cost commodity machines can also be much cheaper than buying the specialized hardware required to run the entire system on one computer.

However, the testing of distributed systems is much more difficult than testing a monolithic or single-computer application. Distributed systems are dynamic and their behaviours change over time. This complicates testing, as the response to various inputs cannot be accurately predicted. Distributed systems have discontinuities; these occur, for example, when queues overflow. These systems also exhibit stochastic behaviours. If the same test is run repeatedly on a distributed system, always starting from the same initial environment, the results may be different every time. Distributed systems are non-linear and concurrent, so the timing of events in the system can affect the results. All of these issues make effective testing of distributed systems very difficult to accomplish.

1.1 Prior art

There are four standard approaches used for testing distributed systems:

- Analytical models
- Simulation-based analysis
- Ad hoc testing in real environments
- Emulation-based testing

1.1.1 Analytical models

An analytical model of a distributed system attempts to describe its operation in terms of mathematical equations with, ideally, closed form solutions. As long as the real system follows the model, the equations apply and the details of the system are known. The central problem with analytical models is that, for tractability, they often require assumptions to be made about the system itself.

The equations often used for distributed system analysis [1] have restrictive assumptions that may not apply to all distributed systems. For example, Little's Law, used for calculating the average number of items in a distributed system at a given time, is derived from Markov modelling. Markov models generally require that the random processes involved be ergodic and memoryless, and these restrictions are both a part of the Little's Law derivation [2]. It has been shown that packet traffic on the Internet is complex, and does not follow a Poisson distribution or exhibit the stationary behaviours required for standard Markov models to apply. Instead, this traffic exhibits characteristics of long-range dependency, self-similarity and heavy-tailed distributions [3].

Another example of restrictive assumptions comes from Khoumsi [4], who derived analytical constraints around emulation-based testing. He produced the constraints that must be followed by a distributed system under test in order to achieve observability and controllability in testing. However, the requirements for this method to be accurate are that outputs are not received without an input being sent, and that an input produces at most one output. These requirements provide tractability, but they tend not to apply to real-world distributed systems.

1.1.2 Simulation-based analysis

In simulation-based analysis, a model of a system is created and its performance in various scenarios is examined. In order for simulation testing to be possible, an accurate analytical model must be created to represent the system under test. Simulation is often used to explore the behaviour of models which are too complex to be solved analytically. However, since simulation-based analysis requires the creation of a model, systems for which models do not exist cannot be simulated. A core issue with simulation testing is whether or not the simulation accurately predicts the behaviour of the real world system.

The conclusions reached from the simulation are invalid if abstractions assumed by the simulation tool do not adequately represent the system's true behaviour. In such cases, the performance predicted for the system will prove inaccurate.

Simulation-based analysis is often used for testing network configurations, since it allows the user to test large-scale networks which would be expensive to set up on real hardware. Such simulations are not affected by “real” traffic or other processes and can be exactly repeated using the same model with the same configurations. OPNET Modeler [5] and ns-2 [6] are two of the common simulation software packages, and both are discrete event simulators designed to test models at the network level. The behaviour of networking components can be modelled and predicted using simulation. The testing of distributed systems, on the other hand, is more complex. For example, it is difficult to *a priori* construct a rich enough model that encompasses all fault conditions in a distributed system.

One example of simulation being used to test in a distributed environment is the testing of the Chord protocol [7]. This algorithm is designed for use in peer-to-peer applications. The authors note that the main concern in peer-to-peer networks is how to locate a piece of data requested by a user when there is no controlling authority to ask for the information. This is especially difficult when nodes are joining and leaving the network. Chord is used for mapping keys to nodes in a balanced way so that the load is evenly distributed across the network, and so that the number of keys that have to be moved when a new node joins is as small as possible. The performance of Chord was examined using packet-level simulation to determine how well-balanced the key distribution is across the network and what the behaviour of the system would be if several nodes failed at once. However, the simulation included only packets with exponentially distributed delays, which may not be an appropriate assumption for the distribution of packets on an actual peer-to-peer network. If peer-to-peer traffic does not actually follow an exponential distribution, this simplifying assumption could be ignoring the additional load caused by “bursts” of requests for information about a particular node. A test on a live network, assuming the system is ergodic, would help to determine whether the packet distribution is really exponential or whether Chord is able to handle the data flow that actually exists.

1.1.3 Ad hoc testing in real environments

Ad hoc testing is testing performed via real deployments in real networks. It is useful for testing what will happen when an application or protocol is exposed to real traffic behaviours, as it would be in a production environment. This could uncover failures which the developers may have failed to account for or test in a controlled environment, such as bursts of traffic that could overwhelm the system. Unfortunately, since the traffic is partially determined by other users on the network, each test run will have a different set of test traffic, so behaviour caused by a specific traffic sequence may be impossible to repeat. Repeatability is a key component of the scientific method, and it is not supported under ad hoc testing. If the test needs to be repeatable, an incoming traffic trace can be collected as the system is running in a live network. This traffic trace could then be used to test the distributed system in a controlled network environment, but this would move the testing from ad hoc testing to emulation-based testing, as discussed in the next section.

Princeton's PlanetLab project focuses entirely on tests involving Internet-based nodes, with hundreds of nodes available around the world [8]. PlanetLab's software runs as an overlay network across all of the nodes. It allows users to request use of the machines and creates virtual machines to isolate experiments from other tests running on the same node. The tests are not isolated at the network level, where each experiment is exposed to the traffic from the other experiments and the Internet, introducing the potential for network-level resource conflicts. The nodes are owned by different organizations around the world and are assigned to experimenters as required. PlanetLab's development was driven by feedback from the user community, and was designed with decentralized control and to evolve over time as new requirements or concerns came to light. One of the main concerns was the trust required between node owners and experimenters. Experimenters were worried that their experiments could be terminated early by the hardware owners or disrupted by other users, while institutions providing nodes wanted assurances that their network connection would not be saturated by the users' experiments. These concerns meant that some limits had to be imposed on the users. If an experiment sends its daily bandwidth limit of 16 GB/day, its upload speed is capped for the rest of the day. This restriction could affect the performance of an experiment,

since the upload speed could change several times over the course of a long experiment run.

Tests conducted on PlanetLab's nodes have the same inherent advantages and flaws as any other ad hoc environment. They allow developers to test how their system will operate against live traffic, but lack the repeatability required by the scientific method. PlanetLab is very well structured for testing new Internet protocols or content distribution schemes, and could certainly be used in combination with emulation- or simulation-based testing, *e.g.* Flexlab [9].

1.1.4 Emulation-based testing

Emulation-based approaches allow the actual implementation of a system to be tested in a controlled environment. The workload used to test the distributed system is generated, either from a previously recorded live traffic trace or using artificially generated workloads. The network is isolated and the machines can be restored to their original state before every test, so the experiment can be repeated by rerunning the same sequence of data against the system under test. As mentioned before, repeatability is a critical piece of the scientific method that is possible under emulation-based approaches but not with ad hoc testing in real environments. Since the distributed system can be tested as it would be implemented in a production environment, the testing results generally have higher fidelity compared to the analytical model and simulation-based methods. Emulation does not have the abstractions that analytical models and simulation approaches do, since the software being tested has been instantiated. The system under test can be run on the actual hardware that would be used in production or a close approximation of it. The limits of the physical hardware are only an upper bound on the hardware speeds that can be used in testing and slower speeds can be achieved by scaling down the performance of the testing hardware, *e.g.* decreasing CPU speed. However, not all testbeds can accurately reproduce the results in all production environments. If the two environments were using different processor architectures, the software would need to be recompiled to run the tests, which could affect the performance characteristics. Emulation on actual hardware can also be very expensive, as machines need to be purchased and reserved specifically for testing purposes, so that measurements can be taken without other running software affecting the results. Networking hardware is

required to connect the machines, since an experiment cannot be affected by live traffic, or it loses the advantage of repeatability. Despite the expenses, emulation approaches are worthwhile as they can be used to test systems which cannot be explored using analytical or simulation-based approaches. They also allow the simulation results to be validated using the actual implementation of the system, and they provide the repeatability missing in ad hoc testing.

Virtual machines (VMs) are one way of extending emulation-based testing. VMs allow multiple independent operating systems to be run on one physical computer. This is a much cheaper solution than purchasing all the computing hardware required to have each experiment running on its own set of machines. Some virtualization software is freely available, further reducing testing costs. Software such as Xen [10] allows each of the installations to run mostly independently, as each VM can be assigned its own portion of the RAM, hard drive, *etc*. However, multiple virtualized experiments running on the same processor can compete for CPU time, potentially affecting the performance of all distributed systems using that machine. Also, since communications to the experiment are passed through the VM software the timing characteristics are changed, which may affect the measured performance. For example, running an experiment inside a Xen VM adds additional overheads [11], which could change the behaviour of the system.

Emulation has been supported by the creation of general-purpose testbeds at academic institutions and in industry. The Emulab project at the University of Utah [12] includes both a network of local machines as well as access to the PlanetLab nodes distributed around the world, and can be customised for different machine configurations depending on the needs of the user. This structure allows them to support simulation, emulation, and ad hoc testing experiments. The local machines can be used for repeatable, controlled experiments under emulation, where each machine starts from a known state and the network is isolated. The nodes are fully configurable by the users, who have limited root permissions on the machines they use. Emulab uses the FreeBSD Jail [13] mechanism to isolate the experiments from each other, allowing multiple users to have separate root access. If an experiment goes awry, the broken experiment can be reimaged without affecting the other experiments running on those boxes. The experiment setup and data collection is left entirely to the experimenters themselves. This creates the core

of a general testbed, but does not provide a distributed system test harness for the experimenters to use to test their systems.

Emulab's software has been used for other testbeds such as DETER [14], which focuses on experiments in computer security. Other security-focused testbeds include the National Cyber Range being planned by DARPA [15], which will allow users to control the configuration of nodes via "recipes". These recipes define the complete configuration of each node in an experiment, including the operating system, applications, security protocols, *etc.* They also define the experiment itself, in the form of a specific test configuration and the parameters for each run. Recipes can be used to restore a machine to the same default configuration between test runs and to run an exact test again at a later date if it needs to be revisited.

All of these general testbeds have similar concerns: isolation of experiments, security of users' data, providing the capability to test using the scientific method and appropriate distribution of machines between experiments. The developers of the systems under test write and configure their own test harness within the minimal constraints imposed by the testbed.

1.1.5 Distributed system testing software

There is some software available for testing distributed systems that could be used on these general testbeds, but it often comes with restrictions. Existing technologies for testing are often only applicable within a narrow domain or to a specific distributed system. Developers writing new distributed systems have to create their own test harnesses in order to test their systems, rather than being able to re-use a general test harness. Development of these test harnesses is generally expensive, time-consuming and potentially error-prone.

There are several required components for testing of a distributed system, as shown in Figure 1. A workload generator is required to create test data that can be used to stress the system. The results of the test runs are collected through hooks placed in the distributed system. At the end of a run these results are analysed and used to determine the configuration for the next run, which is set up either manually or through an automated process. Some of this functionality can be supplied by existing software, depending on the technologies used in the distributed system. However, the goal of a

general purpose distributed system testbed would be able to swap one distributed system for another and still collect the results reported through the standardized hooks into the distributed system under test.

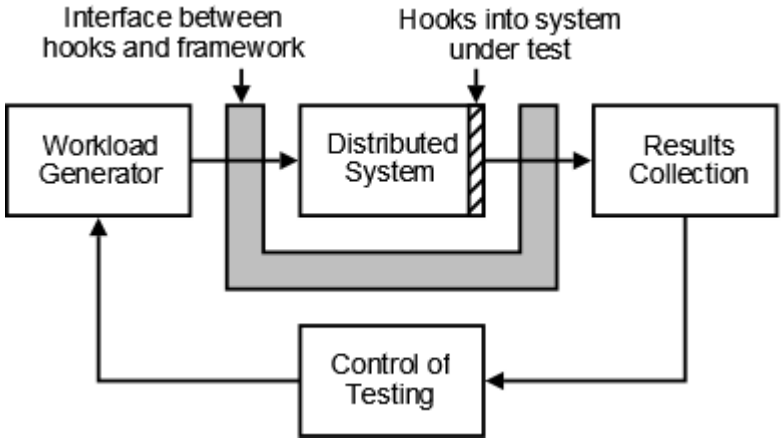

Figure 1: General distributed system testing

1.1.6 Overview of existing partial solutions

There is testing software available which could be used to support some of the functionality outlined in Figure 1. This existing software is not a complete solution and falls into several categories: packet-level workload generation, testing specific components of a system, testing systems with particular architectures, and testing distributed systems which use certain implementations.

1.1.7 Packet-level workload generation

Workload generation software exists, and mostly focuses on workloads at the packet level and, to a limited degree, the application level. Within the network engineering community, several examples of packet-level traffic generation software exist. For example, Harpoon [16] is a tool designed to create background network traffic. It can characterize a live traffic trace and generate a sequence of packets with the same timing characteristics but with random data used for the payload. The tool also provides support for creating traffic traces according to predefined statistical distributions. However, Harpoon always uses random data for the contents of the test packets rather than allowing the tester to control the contents, and this capability is critical for testing at the

application level. The payload can have much more of an impact on the operation of a distributed system than it does with packet-level testing, where the focus is on the path the packet takes through the network and the fragmentation of packets.

1.1.8 Specific component testing

The application-level workload generation software that does exist is often limited to only testing specific components of the distributed system. For example, web servers can be benchmarked independently using several tools currently available, such as httperf for general web servers [17] or Apachebench for testing the Apache web server [18]. These tools generate HTTP requests to the specified web server, then measure and display the server's characteristics including requests handled per second, size of replies, and number of errors produced. These tools are not useful for testing general distributed systems which do not use a web server or accept HTTP requests.

1.1.9 Specific distributed system architectures

Other tools are suitable for testing distributed systems with a specific structure. One example of this is [19], which focuses specifically on collecting results from a 3-tier distributed system and using those results to automate the next set of tests. On these 3-tier systems, the authors use testing agents that run on the clients, the middleware and the server, as shown in Figure 2. The agents test each section of the system using unit, stress, integration and regression testing. They then communicate the results to the other agents and determine how reliable the distributed system is. This solution is specific to 3-tier systems, excluding distributed systems which do not follow this model.

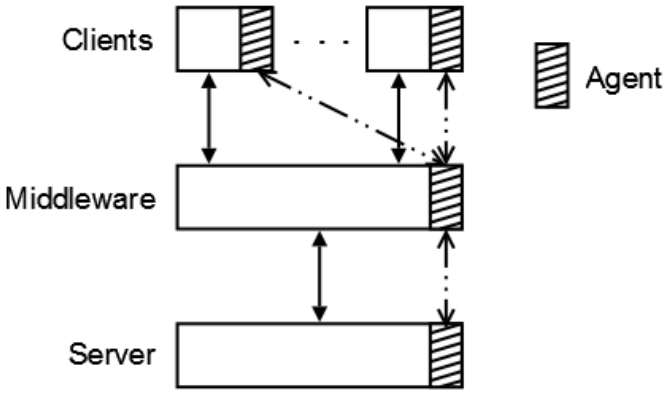

Figure 2: Testing of 3-tier distributed systems using agents

1.1.10 Specific distributed system implementations

Distributed system test harnesses that can test a specific distributed application or a system with certain characteristics do exist. A group at Lancaster University developed a prototype testbed to test a specific Java-based peer-to-peer network [20]. Since their test harness uses Java's versions of reflection and aspect-oriented programming, their current solution can only support Java applications. In their paper, the authors stated an intent to expand support to other programming languages in the future. A group in Sweden has developed a test harness to monitor systems at the remote procedure call level, and currently has support for systems written in Java and C++ [21]. Their solution was designed to help organizations to improve their software test cases. Neither of these solutions apply to the wide range of systems in production use, which are most commonly heterogeneous mixes of technologies and languages, requiring a more general test harness.

1.2 Distributed system test harness

The focus of this thesis is to develop a prototype distributed system testbed to address some of the testing issues discussed above. It is intended to provide a test harness into which many different distributed systems can be inserted. This provides more flexibility than some of the existing solutions that apply only to distributed systems which follow a specific structure or are programmed in a certain language. It can be viewed as an extension of existing emulation testing solutions such as Emulab, which provide a general-purpose testbed but rely on the users to set up and run all their own tests. This test harness can generate workloads at the application layer and run a series of automated tests over time, collecting the data from each test run. This capability is useful for examining the behaviour of a system as a particular configuration option is varied across runs, or to change the inputs for the next test run based on the results of the previous run. The workload being sent to the system under test can be customized, so that the response to workloads with different statistical characteristics can be examined. This allows for testing with the same type of load the production system faces or exploring "what if" scenarios.

The goal for the testbed discussed here was to provide as many configuration options for a test run as was feasible in the time available, while keeping the test harness sufficiently general and limiting the amount of knowledge of the system under test required by the testbed operators. The developers of the distributed system being tested are responsible for installation and configuration of their distributed system, as well as adding the hooks required to allow the test framework to collect information about the operation of their software. The harness provides enough support for developers to allow them to collect test results without having to program from scratch the software required to generate test data, collect results, and run a series of tests over time. Since distributed systems themselves are so varied, the results which need to be collected from each of them is expected to be very different as well. Support has been provided to allow the developers to determine which information they are interested in observing, and to only send those values to the test harness to be logged.

1.3 Thesis outline

The rest of this thesis is laid out as follows:

- Chapter 2 covers the instrumentation of the distributed system and development of the test harness.
- Chapter 3 uses the harness for testing of a prototype distributed system and discusses the effects of the test harness on the system.
- Chapter 4 provides examples of using the test harness to optimize the parameters of a test and improve a distributed system's performance.
- Chapter 5 discusses the conclusions reached and future work required.

Chapter 2: Instrumentation of a Distributed System

2.0 Introduction

The main components of the distributed system testbed are the input workload generator, the results collection system, and an assortment of scripts used to control the test parameters and automate testing. The detailed breakdown of these components in this testbed is shown in Figure 3, which can be compared to the general distributed system testing requirements outlined in Figure 1. These components are all designed to be flexible and scalable to allow a variety of distributed systems to be tested. Smaller tests could be run with one control box setting up the test run, generating the test workload, and collecting the results, or multiple machines can be used for the workload generation and data collection if required for more intensive tests. In order to exercise the distributed system testbed, a prototype distributed system was also developed. Chapter 3 provides an overview of this system. This will be used as the exemplar distributed system under test in the discussions which follow.

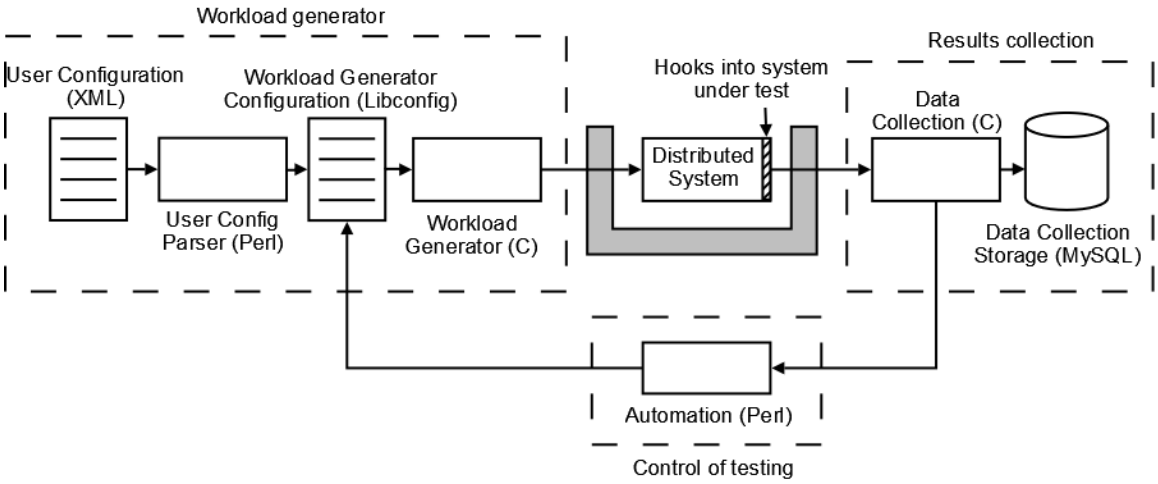

Figure 3: Overall architecture

2.1 Physical testbed overview

A physical testbed consisting of 42 servers connected with 4 separate networks was used for all development of the test harness (Figure 4 and Figure 5). The physical testbed

and its configuration software are not part of the test harness, and the details of their implementation are outside the scope of this thesis. However, a short overview of the testbed's capabilities will be provided here.

The testbed provides scripts that can install a default operating system image on the blades and then configure each machine for the specific experiment being run. Using these scripts, the testbed could be configured with an image similar to the production environment in which each system under test would be run. One of the testbed's four networks is a control network used to connect to the blades and set up experiments; the other three networks carry data generated by test runs. The blades are connected to the Internet through a Cisco switch. In order to avoid tainting the results of a test with other simultaneous test data or Internet traffic, only the machines involved in a particular experiment are connected to that experiment's network and the connection between the blades and the Internet can be severed for the duration of the run. Multiple concurrent tests can be run, with each experiment running on a separate network. The quiet, controlled network and the reimaging scripts allow any experiment to be accurately reproduced from the same initial configuration using the same generated workload.

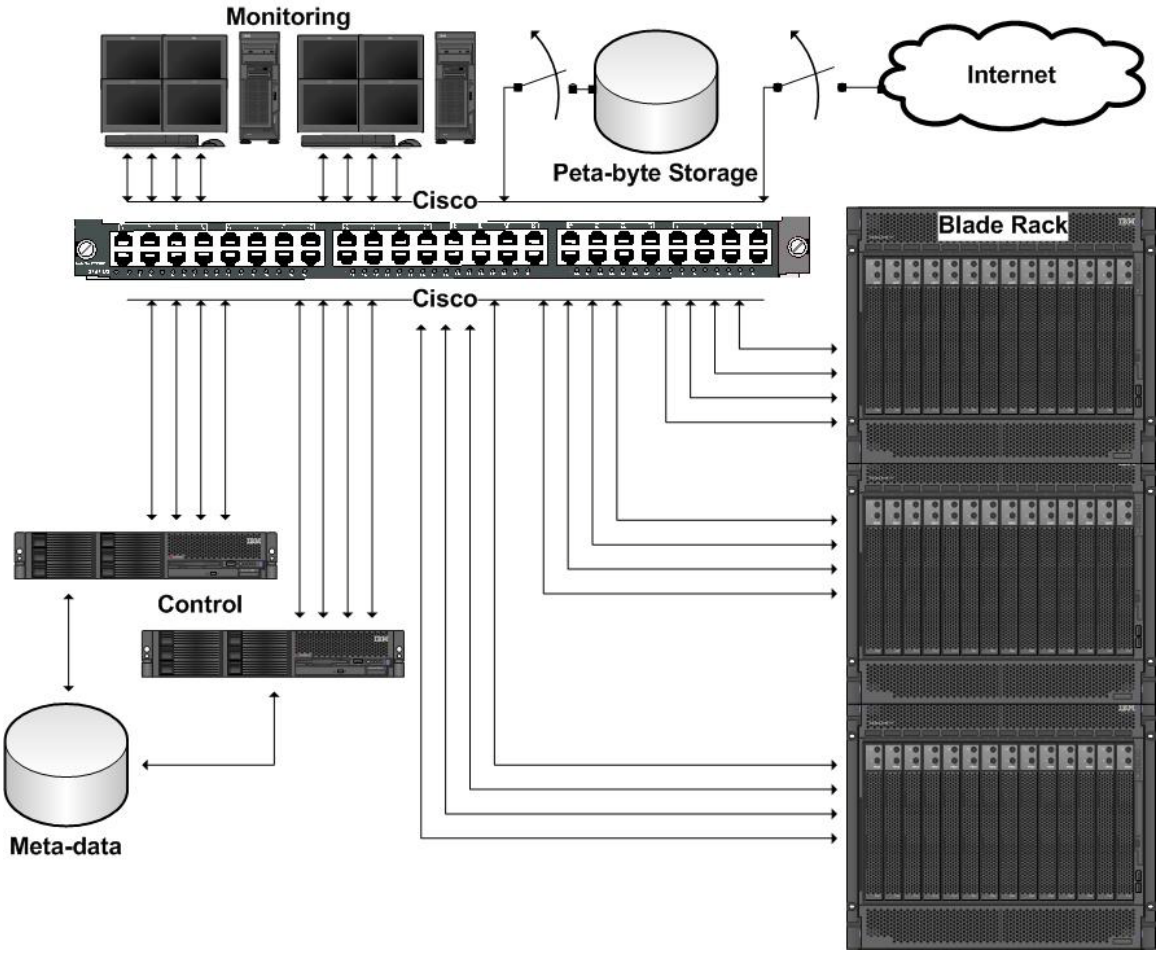

Figure 4: Overview of the physical testbed

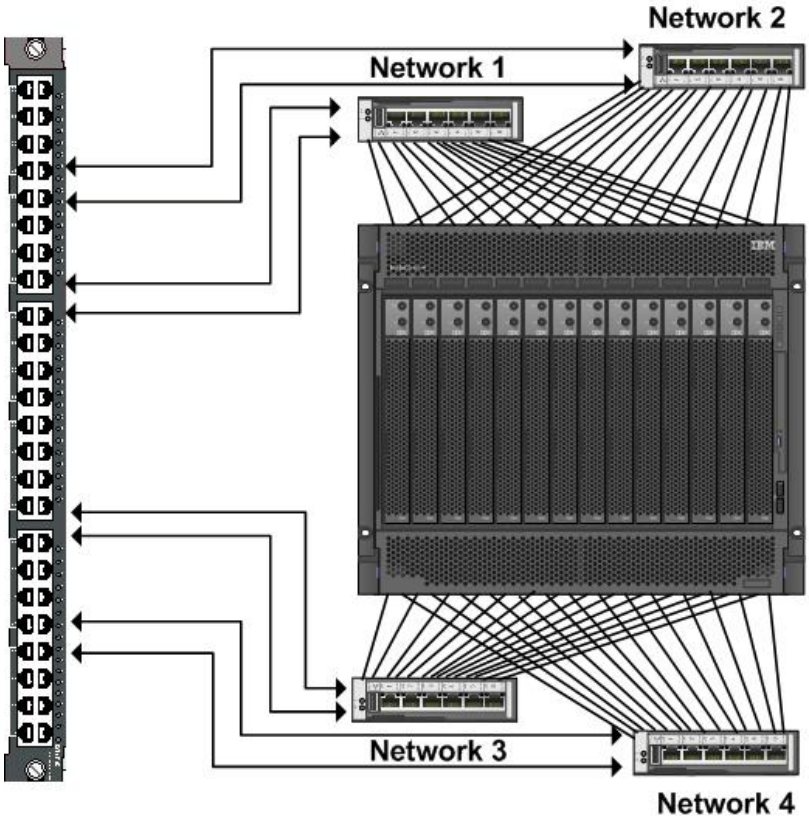

Figure 5: Network connections for the physical testbed

2.2 Workload generator

A workload generator exists which can produce custom workloads to test distributed systems. It generates "posts" which are files containing strings of text, and these post files are sent according to the statistical distribution selected by the tester. The workload generator's parameters are controlled with a libconfig [22] file which is read at runtime and can be changed between test runs. A more detailed discussion of the workload generator is provided in Appendix A. The development of the generator itself is outside the scope of this thesis.

2.3 User configuration

The test harness needs to provide the ability to run multiple consecutive tests with changing parameters, and this can be accomplished by changing the workload generator's runtime configuration between runs. This ability is important as it allows the tester to sweep through a set of input parameters and observe the resulting change in the distributed system's output. To support this functionality, the tester needs a way of

defining the values for the parameters to sweep over. A format was defined for a "user configuration" file, from which parameters could be parsed. The language chosen for this file was XML, since a large number of programming languages have parsing support for it. Many programmers have used XML, which would make the general format familiar to them, instead of requiring that they learn a completely new format. XML processing can often be slow, but since the file is read between test runs, it does not need to completed quickly since it will not change the performance characteristics of a run. A sample user configuration file is shown in Figure 6.

```
<distributedTest>
        <testCollection name="TestScenario1" numRuns="10">
                <string name="content">
                        <value>a</value>
                        <value>b</value>
                </string>
                <integer name="iterations">
                        <value>15</value>
                </integer>
        </testCollection>
        <testCollection name="TestScenario2" numRuns="1">
                <string name="distribution">
                        <value>poisson</value>
                </string>
                <stepper name="mu">
                        <max>10</max>
                        <min>1</min>
                        <step>2</step>
                </stepper>
        </testCollection>
</distributedTest>
```

Figure 6: Sample user configuration file

A "testCollection" is a set of related tests to be run, where the name attribute is used to identify a set of tests later. The "numRuns" attribute is used to run the same test multiple times – a test using each combination of parameters is executed numRuns times, so that the variation between runs can be examined.

Each subsection within a testCollection is a parameter to the workload generator. If only one value is provided for a parameter, such as a single integer or string value, the default value in the workload.cfg file is replaced by that value for every test run in that testCollection. If some of the parameters are given more than one value, a test run is created for each possible combination of parameters. Multiple strings can be defined by providing multiple "value" entries, while an array of integers is formed using the

"stepper" tags. Steppers have a min, max, and step size, and the value starts at the minimum and increases by the step value until reaching the maximum value. All parameters that are not defined in the XML file are left as their default values. So in the TestScenario1 example above, there would be a test with the content=a and iterations=15, and another test with content=b and iterations=15. All other values would be the defaults, and each of the two tests would be run 10 times. The default values for each parameter are shown in Table 1.

Parameter name	**Default value**
hostname	localhost
port	8080
iterations	1000
tracepath	.
numThreads	1
persistentConnections	true
distribution	Poisson
alpha	0
beta	0
lambda	0
mu	0
header	(blank)
content	Inspire
headerBodyDelim	\n\n

Table 1: Default user configuration values

A Perl script was developed to parse the contents of the XML file and use them to produce libconfig files that can be read in by the workload generator. The Perl script produces a libconfig file, starts the test run, then waits until the run is finished to write out the libconfig file with the next parameters and start a new test run. If a test with a certain set of parameters needs to be run multiple times, the tests are executed one after another without having to write out the libconfig file again until a test with new parameters is to be run.

2.4 Data collection

2.4.1 Overview

The data collection components are designed to track data moving through the user's distributed system and collect information about throughput, breakpoints, and whether any data is being lost. Data is sent from "hooks" placed in the user's code by their developers, tracking data every time it moves into or out of one of the user's distributed system components. This data is then stored for later analysis. An overview of the data collection system is shown in Figure 7.

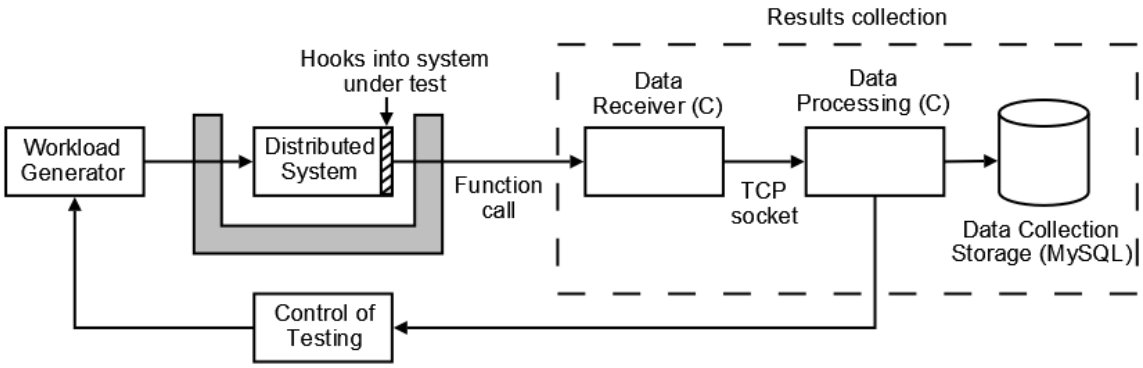

Figure 7: Architecture of data collection piece

The data collection needed to be flexible, as different users might have information they wanted to track which is specific to their distributed system and the needs of their organization. To accommodate this, the data collection system receives and processes strings, which can contain any text the distributed system testers want to record about the current state of the system or the progress of the workload data passing through it. The system also needed to be fast, as each component of the system under test would be sending streams of data for the duration of each test, and each piece of information needs to be stored as quickly as possible so the testbed can keep up with the rest of the data coming in. Ideally, the data collection system should not affect the performance of the distributed system under test, since this could change the characteristics that data collection is intended to monitor. In reality, some measurements cannot be made without having an impact on the performance of the system, and the goal is then to minimize this impact. The effect of measuring a system's behaviour on the behaviour itself is

sometimes called the “observer effect”. The number of components sending data in the system under test increases the load on the data collection components, so the ability to scale the capacity of this system to handle the increased load was also important.

Since processing and analysis of the test results can generally be done when all the runs are complete, the retrieval of data from the database for these purposes does not need to be extremely fast. However, speed is a concern with large data sets, since full table scans of unsorted data are potentially very slow. If a series of test runs are automated and run consecutively, the results of each run also need to be stored somewhere for later retrieval and analysis. This is accomplished by dumping the contents of the database, which can then be loaded back into MySQL later for analysis. With all of these considerations, the specific design for the data collection component is shown in Figure 8.

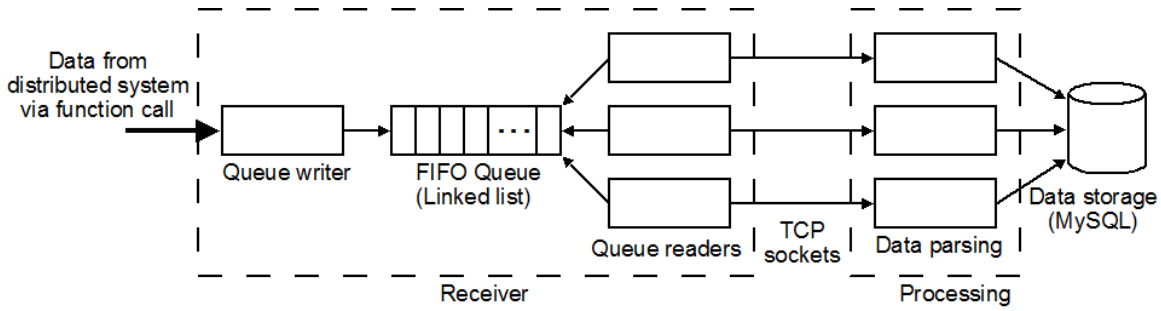

Figure 8: Detailed view of data collection

The data collection’s impact on the distributed system needed to be minimal, and the processing is both time and CPU-intensive, so a design was chosen which did not require the distributed system to wait for all the data processing to complete. There are two data handling steps, receiving the data and processing it for insertion into the database, and separating these two meant that the distributed system could send strings off for data collection, and continue with its own operation as quickly as possible. Strings received by the data collection component are inserted into a FIFO queue via a queue writer thread in the data receiver. No processing happens before the queue insertion so the queue writer can continue accepting data as quickly as possible. A set number of queue reader threads are used to retrieve strings from the queue, and each thread sends the strings to a corresponding thread in the processing software on a remote machine over a TCP socket.

2.4.2 Transmitting data from the distributed system

Three approaches were considered for passing data from the distributed system to the data receiver:

- TCP sockets
- UDP sockets
- Function call to software on the local machine

The method selected needed to have minimal impact on the performance and operation of the distributed system, to avoid changing the test results. Running the data receiver on a remote machine would prevent its CPU and memory usage from interfering with the operation of the distributed system itself, but this would require a fast and reliable way of passing data over the network.

TCP sockets were originally considered for passing data, but the TCP handshaking procedure slowed down the data transfer enough that the distributed system was often left waiting for the transmission to complete. The measured impact of TCP-based data logging on the prototype distributed system was a 48% reduction in the system under test's performance, *i.e.* an average of 558 requests per second compared to an average of 1059 requests per second. The TCP socket connections were deemed to have too great of an impact on the distributed system itself.

UDP sockets were also tested, which do not guarantee that the data will arrive. It was expected that data would not be lost on the quiet, controlled network used for testing in the physical testbed. However, with the speed at which the distributed system runs and the quantities of data being sent to the collection component, data was being lost due to buffers between two machines overflowing. Since UDP does not retransmit lost data, some of the strings were never received, and the performance of the distributed system dropped to 852 requests per second, a difference of 20% from the value of 1059 requests per second without data collection.

The final solution was to run the data receiver on the same machine as the distributed system code. Strings to be logged are passed to the receiver through a function call, which eliminates the network delays associated with sockets, and lessens the effect of data collection on the system's performance. With the data receiver on the same machine of the distributed system, the system's speed was only 3.2% slower than its speed without

any data collection running. However, the CPU and RAM usage of the data receiver could affect the performance of the distributed system, since both run on the same physical hardware. Testing on the prototype distributed system demonstrated that RAM usage was the larger concern, and that this problem could be avoided by adding additional RAM to the machine or by modifying the test being run, the details of which are covered more fully in Chapter 3.

2.4.3 Details of the data collection implementation

All data collected is specified by the developers of the distributed system under test, through the use of a simple XML file that defines which parameters are important to the developers. These are simply the labels assigned to each component of the logged string sent to the data collection system. An example of these labels is shown in Figure 9.

```
<?xml version="1.0"?>
<logger>
        <parameter>timestamp</parameter>
        <parameter>id</parameter>
        <parameter>module</parameter>
        <parameter>postdata</parameter>
</logger>
```

Figure 9: XML file describing sample distributed system results to log

Since the users decide which parameters they are interested in, they can tailor the testing to their specific concerns. Some of the parameters in the example, such as timestamp, id, and module, are likely relevant to all distributed systems. Timestamping a piece of data as it passes through each component allows the user to determine the actual time spent in each module. The module parameter provides the name of each component, so that data moving through the system can be tracked, and an ID number on each piece of data can be used to discover if data is being lost. Any other parameters would be specific to a distributed system domain and implementation, but the user is free to log any string data of interest, as long as it does not contain the character being used as a data separator. The default value for this character is a colon, but it is defined in a libconfig configuration file and can be easily changed.

Each module of the user's distributed system would need to be configured by the developer to report the requested information to the data collection system. This is

accomplished by adding “hooks” to the distributed system in each location where results need to be collected. All results are passed as strings through a function call to the data receiver. The hooks can be written in any programming language, as all they need to do is construct a string with the correct separators and line-terminating character, then send it to be logged. As an example, the procedure for creating this colon-separated string and passing it to the data receiver in C is shown in Figure 10. This example passes the same information outlined in the XML file above (Figure 9).

```
sprintf(dcString, "%ld.%ld:no_id:%s:%s\n", (long)tv.tv_sec, (long)tv.tv_usec,
"queue writer", refBuffer);
dcwriter(dcString);
```

Figure 10: Sample data collection hooks in C

The data processing is all written in C. The XML file is first checked for consistency with the .xsd schema file which defines the format, by using the “xmllint” command. The only parameter requirements which are currently enforced by the XSD schema are that all parameter names must be alphanumeric and between 1 and 50 characters long. To avoid parameter names conflicting with reserved words in MySQL, the column names are of the form (parameter)_logger. Since the maximum length of a column name in MySQL is 64 characters, the upper length limit of 50 characters should prevent column names that would cause a MySQL create table statement to fail. If the XML file passes schema validation, the parameter names are parsed out of the file and columns in a MySQL database are created based on those parameters.

Once the database is fully configured, a thread pool is created in the data processor. These threads sit and wait for any information to be sent from the data receiver, and store any received strings in the database. In order to make the database insert go as quickly as possible, each thread has its own static connection to the database and uses a prepared statement for the insert. Prepared statements allow the “insert” SQL command to be created once and associated with that thread's connection to the database, then used repeatedly with the data changed for each insert.

Data is sent through a TCP socket connection to the processing software. The original version of this software used pipes to receive data, but it was found that the pipe buffers

would overflow if a large amount of data was sent and the inserts to the database could not be performed quickly enough. This was causing a small amount of data to be lost – approximately 8 items out of the 50 million items sent for a particular test. Switching the data transmission to a socket connection has been much more reliable, with tests run up to 110 million entries and no data being lost.

Information is sent from the distributed system and passed through the data receiver for processing in a specific format, so the strings can be parsed out later. For example, using the column types defined in the XML file above (Figure 9):

1192389234334:4:database1:bob\n

The \n character denotes a new line, and is used to separate the lines of data moving through the distributed system. Each line of data is split based on the colon separator and inserted into the database in the same column order defined by the XML file. The data separator can be changed in a configuration file to any character which is certain to not appear in the data being transmitted, as determined by the developer of the distributed system under test. Since the relevant column names and data to be collected are defined by the user, there is no way to be sure of the type of data submitted while still abstracting the user from the underlying data collection system. If the user had to identify the type of each piece of data, they would need to know which database was being used for storage, since the database implementations (*e.g.* MySQL, Oracle, PostgreSQL) are not completely consistent on the names of their data types. This is an unreasonable requirement and would tie the system to a particular database, so all data is inserted as a "text" field in the current MySQL database, which is wasteful in terms of space but allows a text string of any length to be stored. Once each test is complete, the results in the database can be converted to implementation-specific column types and moved to long term storage using post-processing scripts if necessary.

The data collection component was designed as a distributed system, which allowed the requirements of speed and scalability to be satisfied. If one instance of the data collection piece is not sufficient for handling the traffic generated by all the components of the distributed system, multiple instances can be started. The data receiver is naturally

distributed, as a copy of it needs to run on every machine where the distributed system is running and needs to log information. Duplicates of the processing component would require editing the data receivers to make each one point to a specific instance of the processing software. All the processors could insert to the same database, or a separate database could be started for each of the new streams of data (*e.g.* Figure 11), and results would have to be retrieved from all of these databases for analysis at the end of a test run.

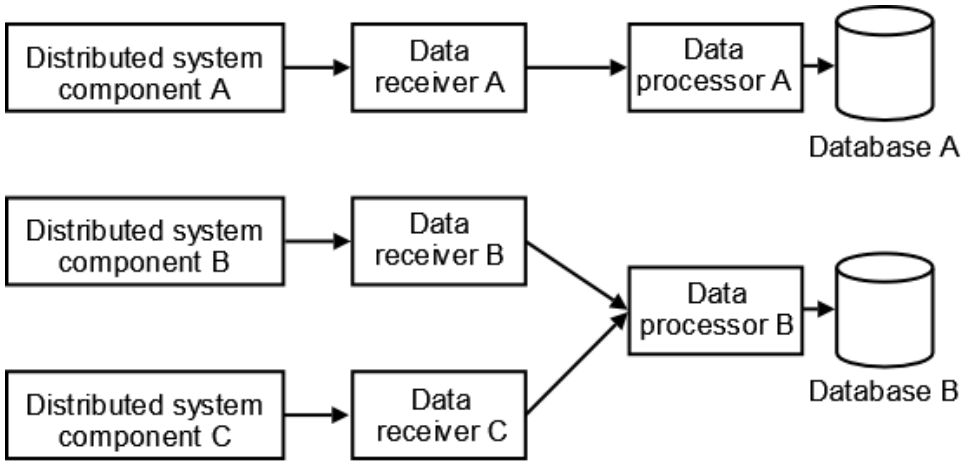

Figure 11: Example of distributing the data collection system

2.5 Automated testing

Automation is a key component of the testbed, as it allows tests to be run consecutively without human interaction. This is valuable, especially when many tests need to be run and it is not feasible to have someone checking constantly to see if a test run has finished and start the next one. There are two main uses for automated testing: running the same test multiple times to gather statistical information about the system under test and sweeping through a set of parameters to examine the changes to the system's performance as the parameters change. Repeated runs of the same test are important as distributed systems are stochastic, and their behaviour could change significantly between runs. If a test is only run once, limited information is collected about the underlying distribution that the measured parameters follow. Collecting data from multiple runs of each test provides information about these distributions. The other use of automation, sweeping through a set of input parameters, allows the testers to find the value which produces the optimal performance of the distributed system. These results can be used when selecting the configuration parameter values in the production version

of the system. These two types of tests can be run using the automation scripts in the test harness.

The Perl automation scripts developed for this test harness focus specifically on setting up experiments, monitoring them while they run, then killing them and cleaning out the results so the next set of tests can be run. They can be used in combination with the user configuration scripts or the search scripts (Chapter 4), both of which edit the workload generator's configuration file between automated runs. Once the test has been configured, the automation scripts simply run the same test as many times as specified.

The automation script is only responsible for controlling the operation of the test harness itself; it calls startup and teardown scripts provided by the developers of the distributed system under test to control its operation. The automation script starts all of the test harness software, then calls the startup script for the distributed system and waits for the test to complete. At the end of each test run, the teardown script for the current distributed system is called, followed by any cleanup required for the test harness components themselves (Figure 12). This keeps the automation scripts general so developers using the test harness do not need to know or edit the actual harness code to accommodate their distributed system.

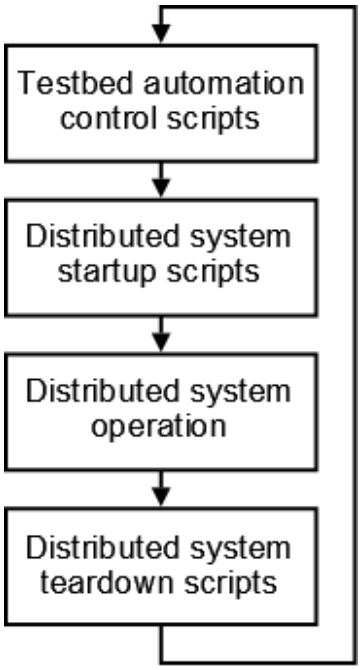

Figure 12: Workflow for the automation scripts

The automation script currently waits until all of the workload data has been received at a database before concluding that the test is complete. This method assumes that the test will be successful, and the success condition is specific to the prototype distributed system. One potential solution to this is to decide that a test has failed if the test does not complete within a predetermined amount of time; this would be trivial to implement. If the timeout is reached, any software that is still running would be killed and the test would be restarted. The length of time to wait would depend entirely on the particular test being run and the characteristics of the distributed system under test, and thus would need to be provided by the developers of each distributed system.

Before the test begins, the script cleans up the contents of the prototype distributed system and data collection databases to ensure that no data remains from previous tests. It is essential to start from the same initial machine configuration before each test is run, as any artifacts from a previous run could affect the results. This cleanup is run at the beginning of the test, rather than at the end, so that the last test will leave data in the database in case the database results are of interest to the testers or errors need to be tracked down.

After the cleanup is complete, the automation script starts the appropriate software on each testbed machine. The script includes a set of configuration options that define which machine should be running which test harness component. Once the software is started, the script checks that each element of the test harness is running correctly. If a test fails due to software failing to start, the test harness will terminate any processes which started successfully and automatically rerun the failed test. Currently if a test run fails repeatedly the automation script will try to rerun it until a human intervenes. This could be resolved if the script was changed to keep track of the number of times a specific test had failed, and to terminate the test if the number of failures reached a certain threshold. The output of each test run is printed to log files, so if a test does fail unexpectedly the user can determine what happened later by reading the error messages that each software component prints to these files. If the test setup is successful, the automation script then starts checking the prototype distributed system's database periodically to see whether the test has completed, and records the output of the "top" command to a file on a regular interval so the testers can monitor CPU and RAM usage.

When the test is complete, the distributed system teardown script is called and the commands to save the results of the test and kill the test harness components are run. If there are tests left to run, the distributed system and data collection databases are cleared and the next run is started.

2.6 Timestamp processing

Timestamps can be recorded for any test where timing is important, such as measuring the throughput of a distributed system. The initial timestamp is inserted into the test data by the workload generator, and the final timestamp is recorded when the data is inserted into the database at the end of the distributed system's operation. This is implemented in the example distributed system as a proof of concept, but could be altered to accommodate other distributed systems as required.

Between each automated run of the distributed system, the start and end timestamps are pulled from the database and printed out to a text file. This raw data includes timestamps for one line of every post sent through the distributed system. If a particular subset of the data is required, it can be parsed out using a Perl script that was written for this purpose. The script can be used to extract information such as the average or standard deviations of the time a piece of data takes to pass through the distributed system, or the average over each 10 lines. The raw data is still stored so it can be examined later if there are interesting anomalies in the results.

2.7 Summary

This chapter has covered the design of the prototype test harness components including the user configuration scripts, the data collection subsystem, the automation scripts and the timestamp collection capabilities. In combination with the existing workload generator, these allow tests to be run on and results collected from distributed systems. There are a large number of parameters to provide support for a wide variety of test configurations.

Chapter 3: Testing a Prototype Distributed System

3.0 Introduction

Testing was carried out on a prototype distributed system to determine whether the test harness components themselves were affecting its performance and as an example of the ways the harness could be used to explore the operation of a distributed system. All tests were run under Ubuntu Linux, version 7.04 (Feisty Fawn), 64-bit edition.

3.1 Prototype distributed system

In order to fully test the distributed system testbed, a sample distributed system needed to be created. The system under test needs to be instrumented to collect information about the data moving through it between each component, so it was important to have a system which was fully understood and configurable by the testbed designers. Since the code was familiar, the data collection components could easily be inserted into the software being tested, as would be required by any organization who was planning to use the testbed to measure the performance of their distributed system. The system was designed to be fast, so the processing capabilities of the testbed could be explored and any situations where it would not be able to keep up with the streams of data being sent to it would be exposed. Since speed was important, C was chosen as the language for this test system. The distributed system architecture is shown in Figure 13. Its capabilities are modelled on those of a proprietary, commercial distributed system that is used for web mining and which will be discussed further in Chapter 4.

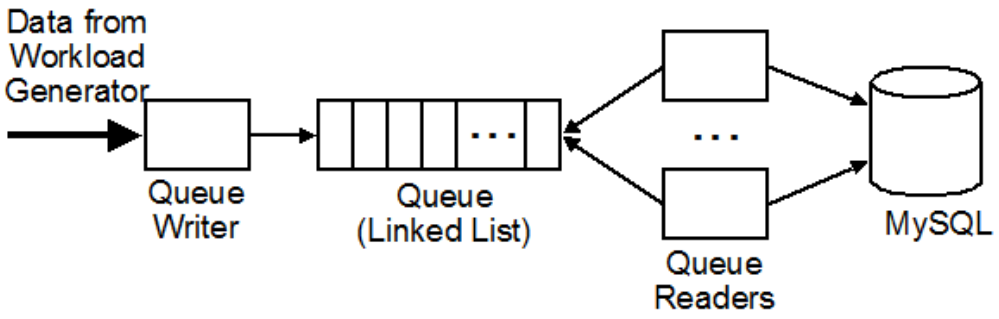

Figure 13: Prototype distributed system

The queue writer accepts TCP socket connections on port 8080, which receive strings separated by the newline (\n) character. This data is produced by the workload generator and sent according to the statistical characteristics defined for the particular test. The queue writer is multithreaded to allow multiple simultaneous connections from the workload generator, with each socket connection running in its own thread.

The queue writer thread receives characters one at a time from its incoming socket until a new line character is reached. It takes the string it has received between new lines and inserts it into the linked list, then posts a semaphore to indicate that there is data waiting to be read. The linked list is a FIFO queue protected by a mutex to avoid multiple readers trying to access the same piece of data or the writer thread adding data at the same time. If the incoming socket fails, the workload generator will create a new socket connection to the queue writer and continue sending data.

The queue readers are each a separate thread, since they have to both read from the linked list and insert into the database, and the latter has the potential to be slow. Each queue reader has its own connection to the database, and the threads sit waiting for the semaphore to be activated before trying to receive new data from the linked list. The first thread to access the data will receive it and do the database insert. The other threads cannot receive a duplicate copy of the same data, as they will be blocked waiting on the semaphore unless there is more data available in the queue, and the first thread will not unlock the mutex on the queue until it has finished removing its data from the linked list. The data insert is done on a MySQL database, where the whole text string is inserted directly into a MySQL “text” field.

There is no processing done on the data, but the system can be slowed down if required for testing through the use of delays. The ability to decrease the speed of certain components of the system is useful for testing purposes to shift the bottleneck of the system to that component and see how that affects the performance of the other modules.

For the testing done in this thesis, the data collection hooks were placed as shown in Figure 14. Each line of data is logged as it enters the queue writer, before it is inserted in the queue. The data is again logged from the queue readers, after they retrieve it from the queue but before it is inserted in the distributed system database. Currently, only the timestamp, the data itself, and the name of the module that sent it are recorded. Since

there are two data collection points for these tests, the data collection system records twice as many lines of data as the distributed system does. One instance of the data collection software was enough to keep up with the prototype distributed system, but for a larger distributed system with many collection points there would need to be multiple data collectors and associated databases.

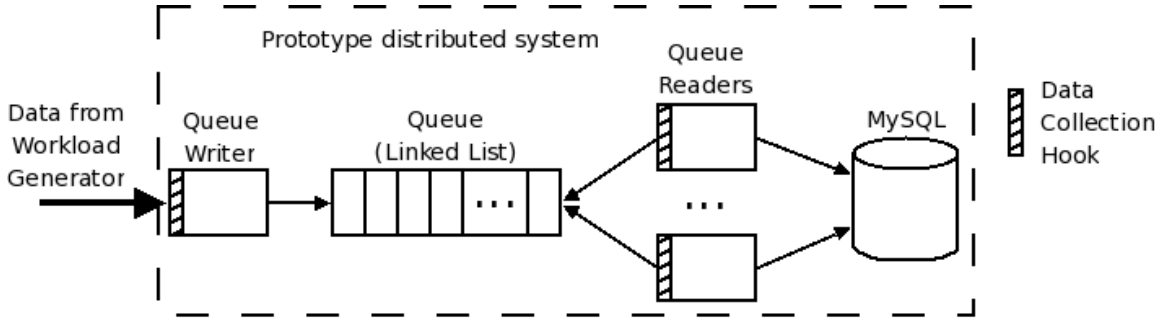

Figure 14: Location of data collection hooks in the prototype distributed system

3.2 Impact of measurement (test harness overheads)

Since the data collection component is designed to monitor the actual performance of a distributed system, it is important that the effect of the logging software on this performance be reduced as much as possible. As discussed in chapter 2, several approaches were tried for sending data to the data collector to be logged. These included TCP sockets, UDP sockets, and pipes, but these methods either heavily affected the distributed system's operation, or would lose data due to buffer overflows when the log data was sent too quickly. The eventual solution was to send the information via a function call to software running on the same machine as each distributed system component. The performance of the test distributed system with and without the data collection component turned on is shown in Figure 15, and the corresponding histograms are shown in Figure 16 and Figure 17. Since some additional processing is required to send log data off to the data collector, there is a small decrease in throughput when data collection is turned on. When the data to be logged was passed via a TCP socket, the distributed system had to wait for the data acknowledgements to be received from the socket before continuing with processing, which reduced the performance of the distributed system to 52% of the speed it achieved with data collection turned off. The

current method of passing data via a function call causes only a 3.2% decrease in throughput, from an average of 1114 requests per second without data collection to around 1079 requests per second. This figure also demonstrates that there is no correlation between the performance of multiple consecutive runs, at least in the case of this particular test.

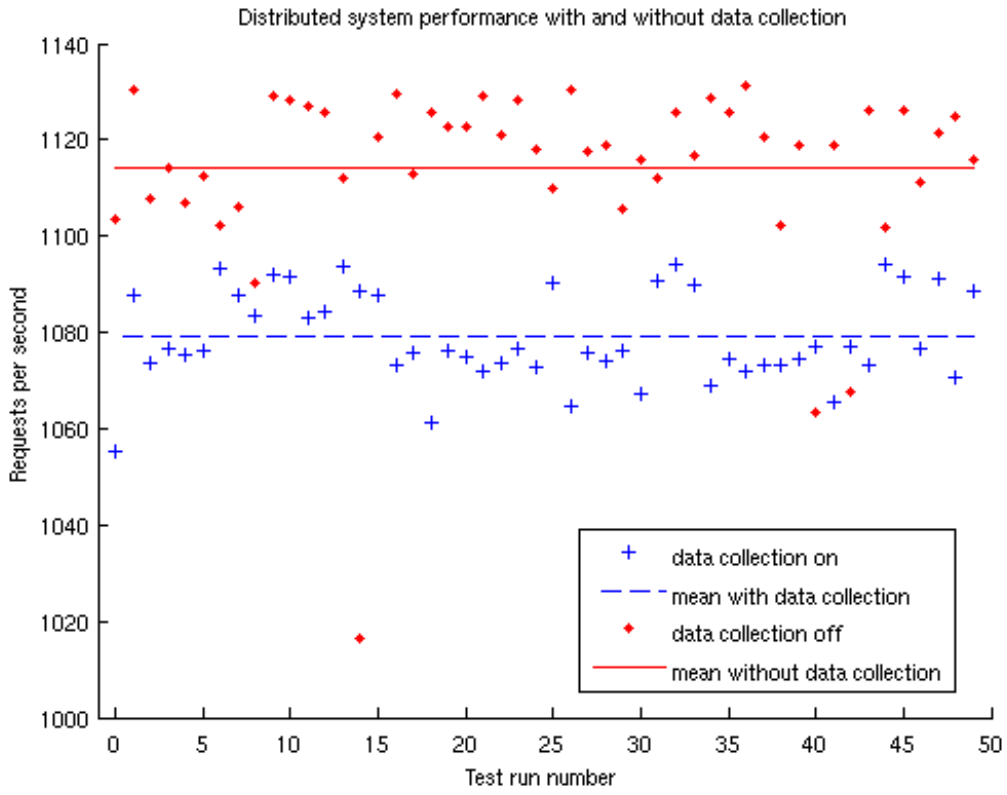

Figure 15: Distributed system performance with and without data collection

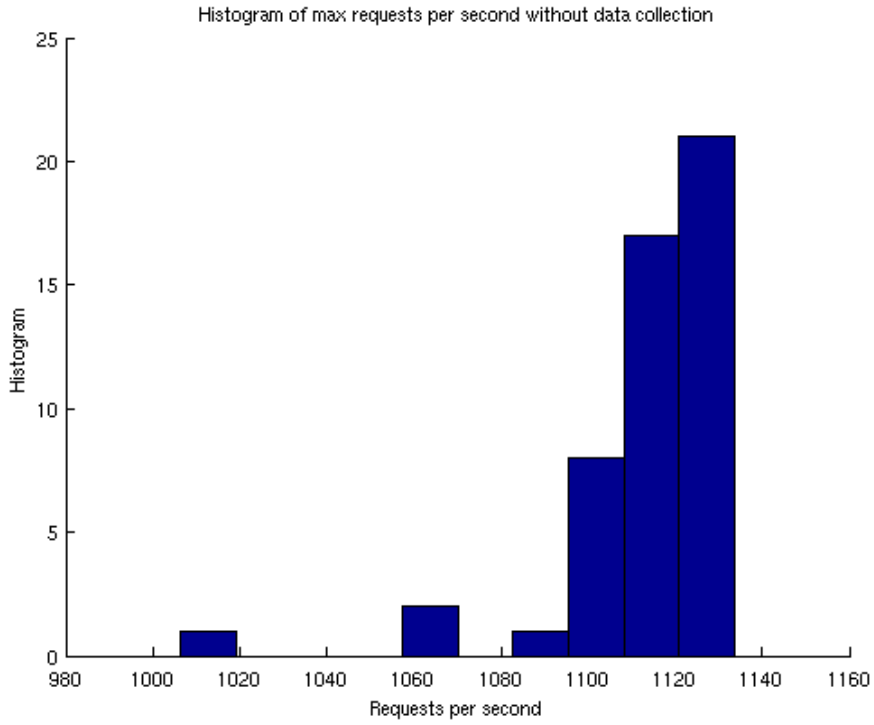

Figure 16: Histogram of distributed system performance without data collection

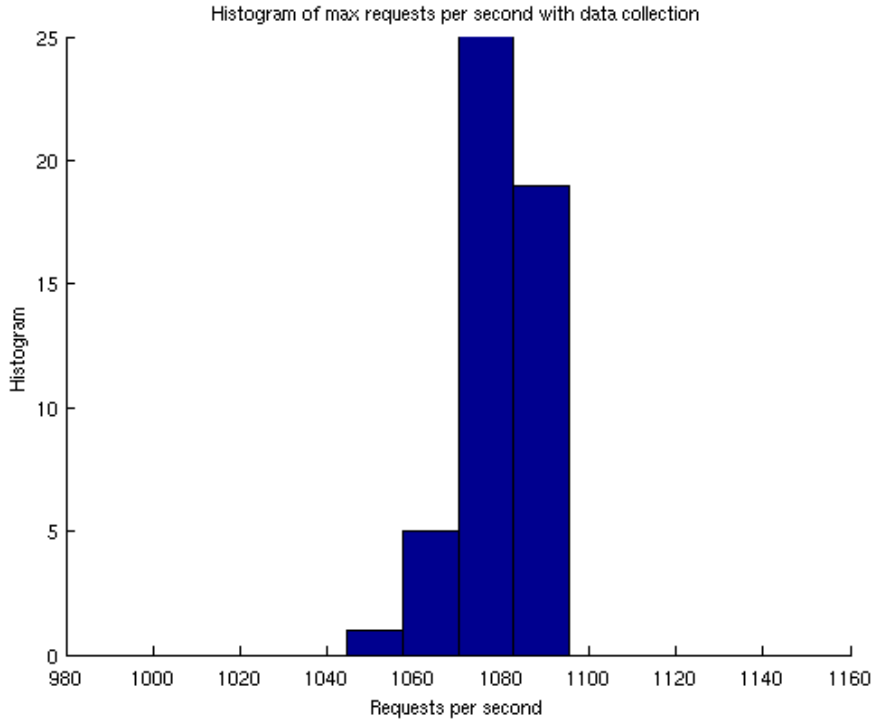

Figure 17: Histogram of distributed system performance with data collection

This does raise the possibility that the CPU and memory usage of the data receiver will affect the performance of the distributed system, especially since the receiver inserts data into a queue structure, which can expand in size indefinitely if the readers cannot keep up with the speed at which data is inserted. Tests were run where the maximum RAM and CPU usage were monitored for the distributed system running with and without the data collection system. With data collection turned off, the RAM usage of the distributed system for this test stayed around 0.2-0.3% of RAM (Figure 18 and Figure 19). With data collection on, the RAM usage jumps to between 40% and 70% (Figure 20). Clearly it is important to monitor the RAM usage during experiments to ensure that the current test configuration does not push the machine into swap, which would slow down the experiment and give unreliable results. For the test distributed system in this thesis, this could be controlled by adding additional RAM to the distributed system machine to keep it from going into swap, or by changing the data being sent to the system under test (Figure 27). The difference in CPU loads with and without data collection is not nearly as dramatic as the difference in RAM usage, with the average changing from 51% without data collection (Figure 21 and Figure 22) to 54% with data collection (Figure 23).

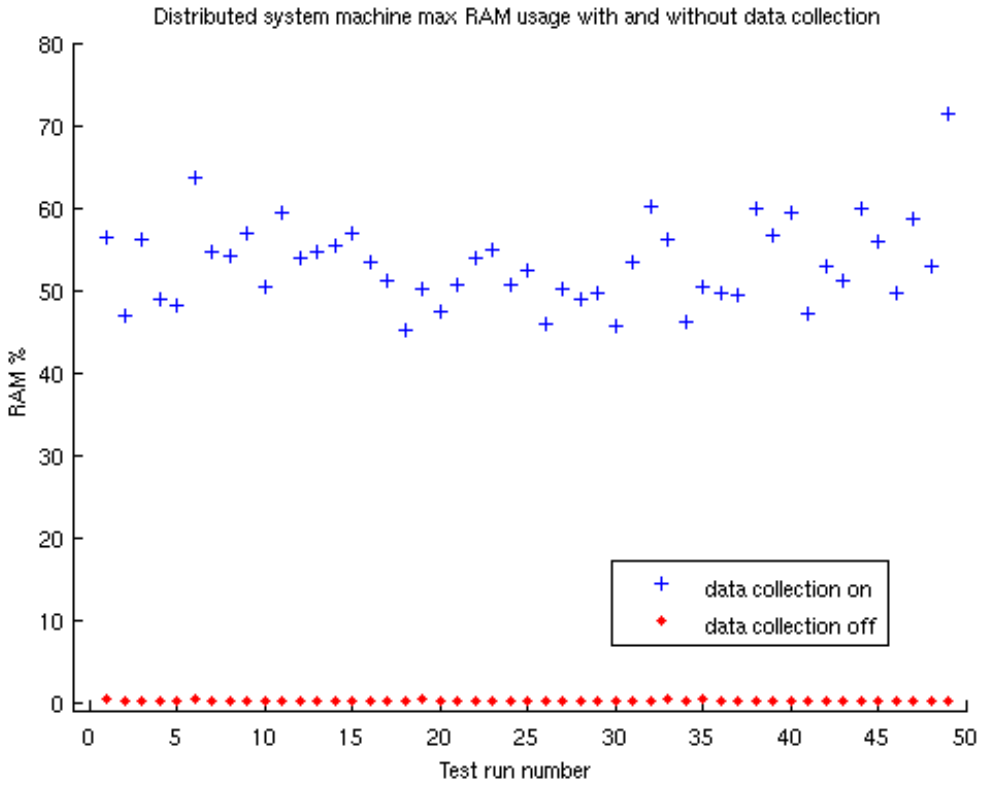

Figure 18: Distributed system max RAM usage with and without data collection

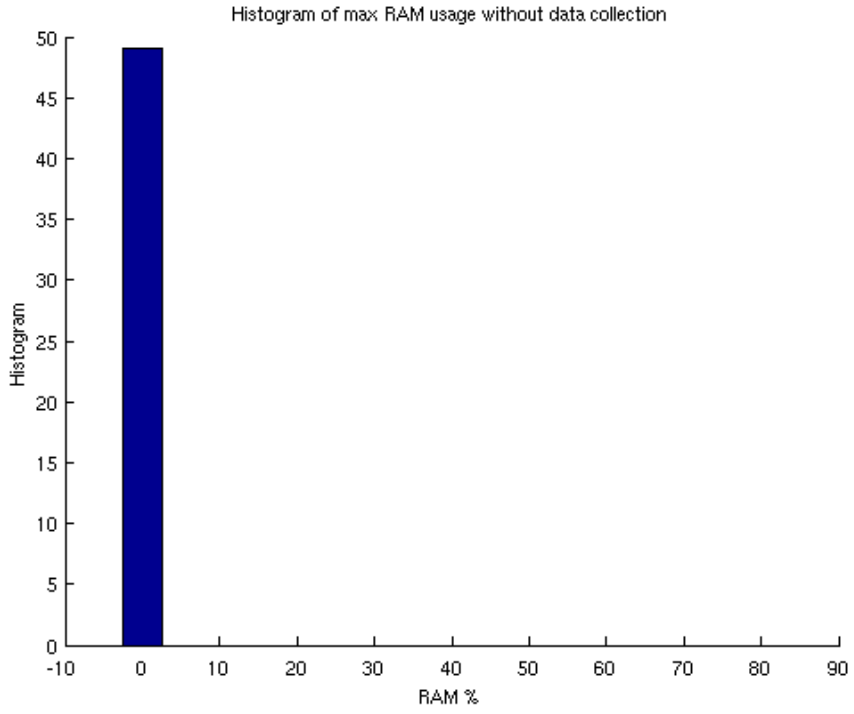

Figure 19: Histogram of max distributed system RAM usage without data collection

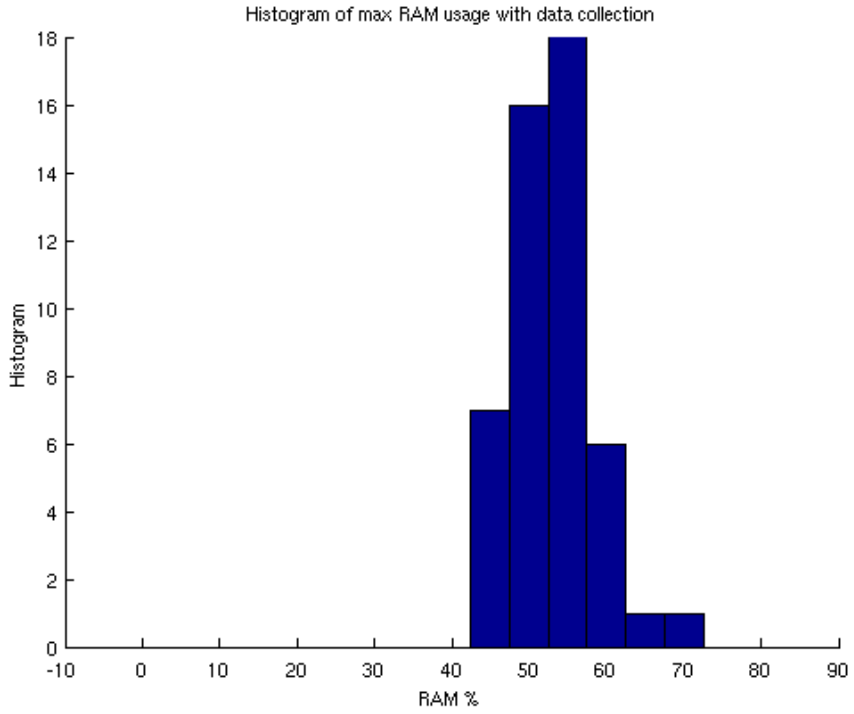

Figure 20: Histogram of max distributed system RAM usage with data collection

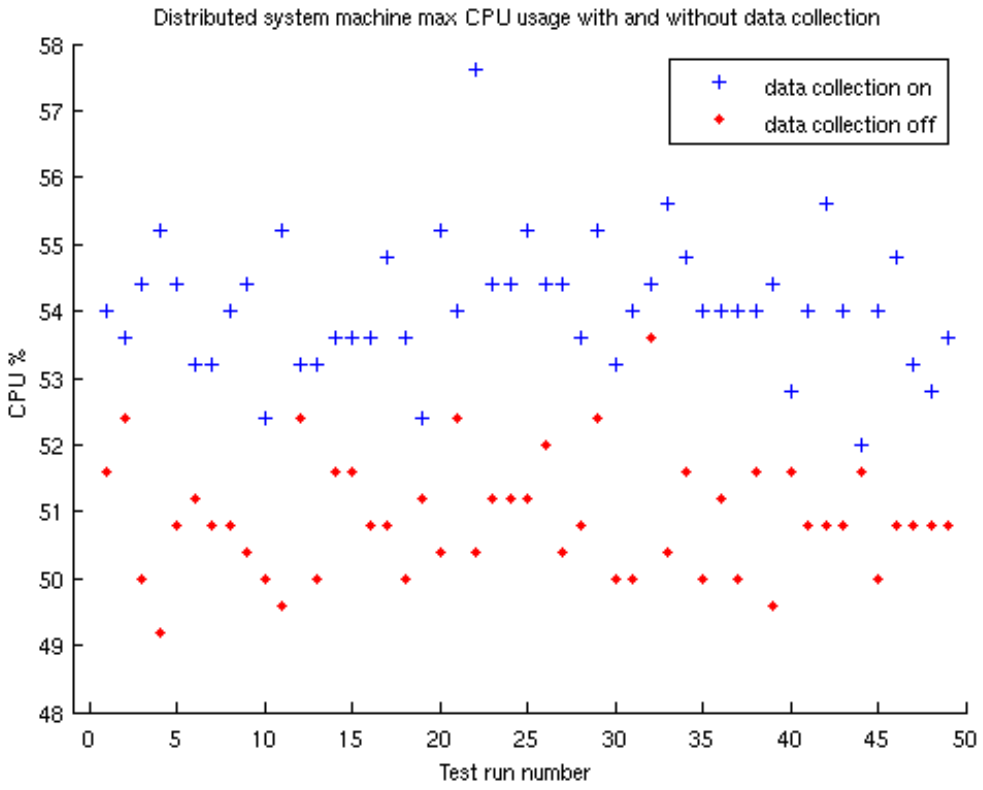

Figure 21: Distributed system max CPU usage with and without data collection

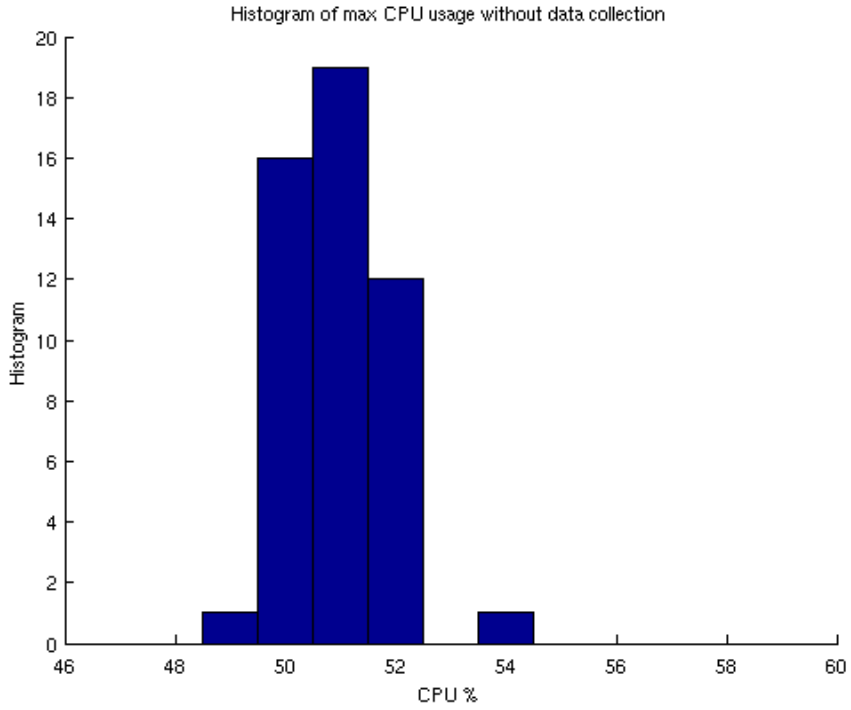

Figure 22: Histogram of max distributed system CPU usage without data collection

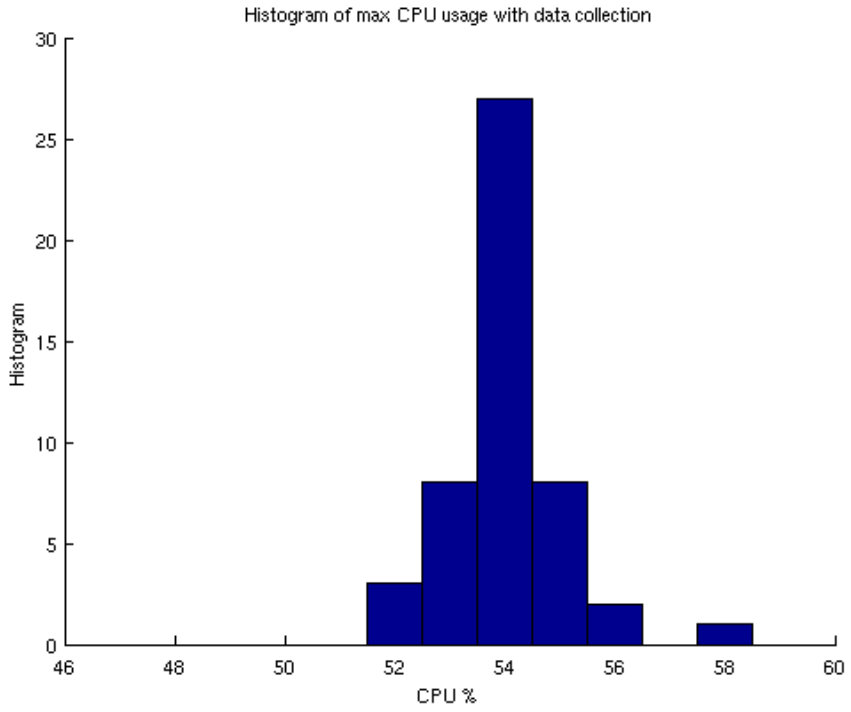

Figure 23: Histogram of max distributed system CPU usage with data collection

One possible method of reducing RAM usage was to add additional reader threads to try and remove strings from the queue faster. Since the queue is held in memory, any time the queue writer is inserting data faster than the queue readers can remove it, RAM usage on that machine steadily increases. The RAM plot with and without data collection (Figure 18) shows that RAM usage is only high when data collection system is turned on, and this increase is solely due to growth of the data collection queue. As every line of data passes through the distributed system, it is sent to the data receiver twice. This means that the data collection system stores twice as much data as the distributed system does, which explains why the data collection queue readers cannot keep up with the rate of insertion into the queue. The theory is that if the data collector were able to remove strings from the queue faster by creating more queue reader threads, the RAM usage would drop. Since the distributed system is configured with 5 reader threads for its queue, the number of reader threads for data collection was varied from 5 to 10. As expected, the max RAM usage seen appears to decrease as the number of reader threads increases (Figure 24). The improvement is small, since there is only one queue and a mutex lock is being used to restrict access to the queue to only one reader thread at a time. Adding additional queues would not decrease the RAM usage, as all queues would still have to run on the same physical hardware and would still be stored in memory on that machine. Storing the queues on other machines would reintroduce the same negative performance effects discussed earlier in this thesis, as the data would have to be transmitted to those machines over a TCP or UDP socket. In cases where RAM usage on the distributed system machines is affecting the performance of the system, additional RAM may need to be added to those machines. On the other hand, the CPU usage in these tests remains relatively constant across the varying number of threads and multiple test runs (Figure 25), so changing the number of data collection reader threads does not appear to have any effect. It should be noted that these tests were run with shorter workload lines on a machine with more RAM, so the actual RAM and CPU usage values in Figure 24 and Figure 25 cannot be directly compared to the results in Figure 18 and Figure 21, respectively.

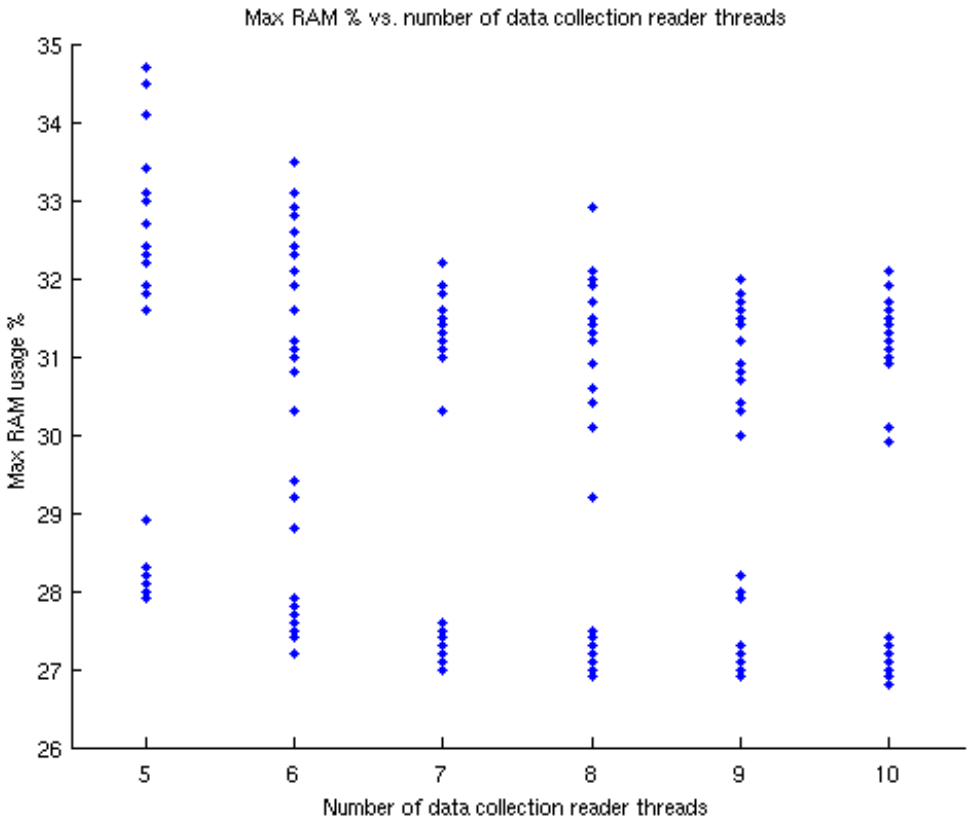

Figure 24: Max RAM usage with changing number of reader threads for the data collection

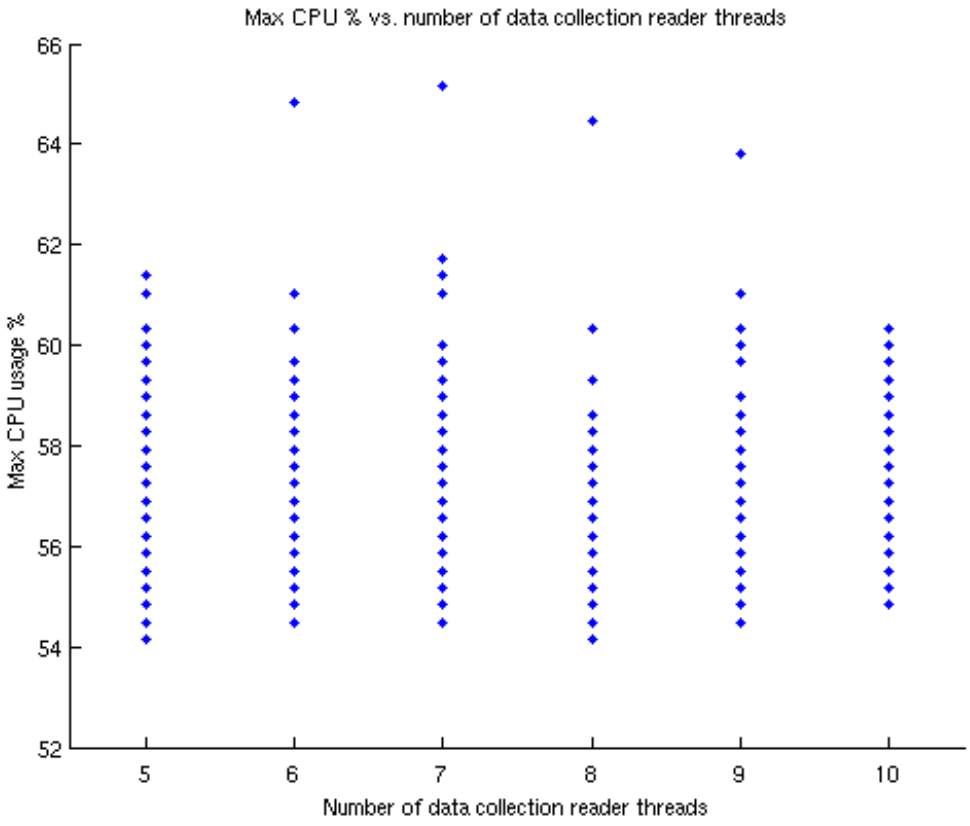

Figure 25: Max CPU usage with changing number of reader threads for the data collection

3.3 Distributed system results

3.3.1 Introduction

Once the test harness' effect on the prototype distributed system was established, it was used to examine the behaviour of the system itself. Three types of tests were run to explore how they changed the distributed system's performance: changing the number of characters in each line of workload data, changing the number of queue readers in the system, and running consecutive throughput tests. The results of the first two types of tests indicate which configuration options for the system would produce the best performance. The throughput tests show that there are artifacts that occur between test runs which need to be acknowledged and addressed.

3.3.2 Characters per line of workload data

One test configuration option that could affect the prototype distributed system performance is the number of characters in each line of test data being generated. Each file sent by the workload generator contains one or more lines of test data, and each file is sent to the distributed system via a socket. The distributed system checks every arriving character looking for a newline (\n) which indicates that a full line has been received, then adds that line to the queue before waiting for more incoming data. As the number of characters per line increases, the number of requests per second handled by the distributed system decreases, since more information needs to be processed (Figure 26). The histogram plot of RAM usage, seen from two different viewing angles (Figure 27 and Figure 28) , also shows a decrease for longer lines, since the queue writer spends more time waiting to receive data from the socket rather than inserting information into the queue. It is clear from these plots that RAM usage is consistently high with fewer characters in each line of data, and shows much more variation across test runs when the lines are longer. Increasing the number of characters per line also results in a decrease in CPU usage (Figure 29 and Figure 30), though the change is not as drastic as the change in RAM usage. The second peak around 40% CPU usage is created at the end of each test, when the distributed system itself has finished processing data, but the data collection portion of the test harness is still handling the results.

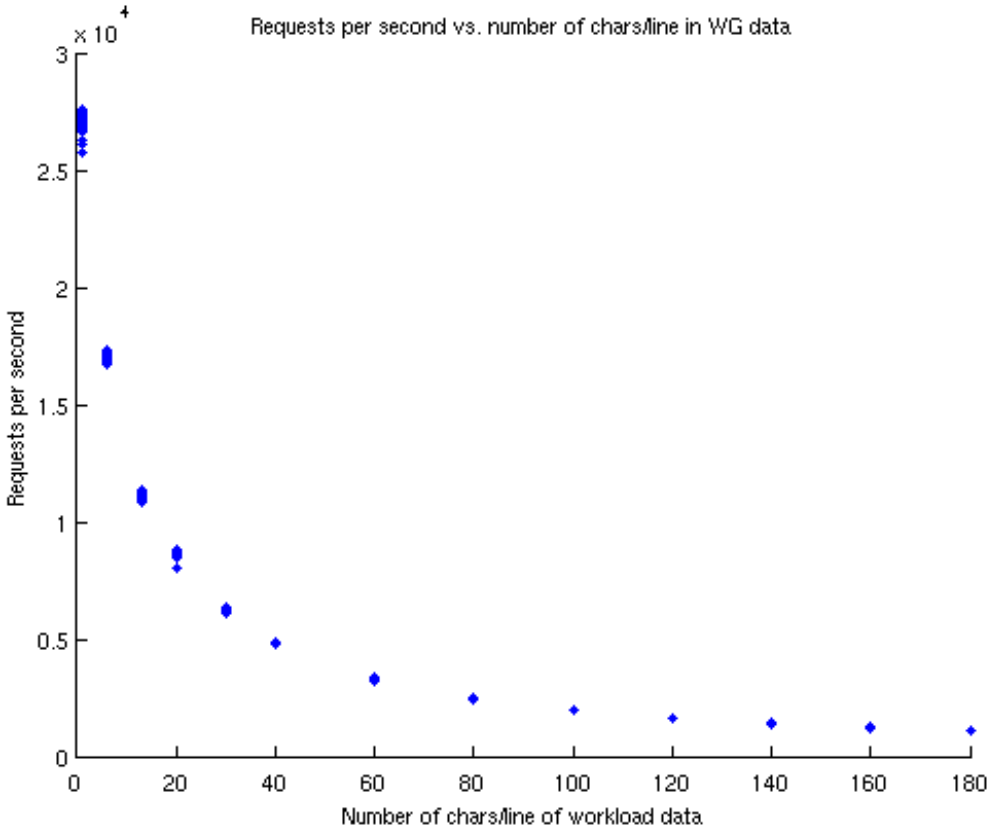

Figure 26: Requests per second for the distributed system with characters/line changing

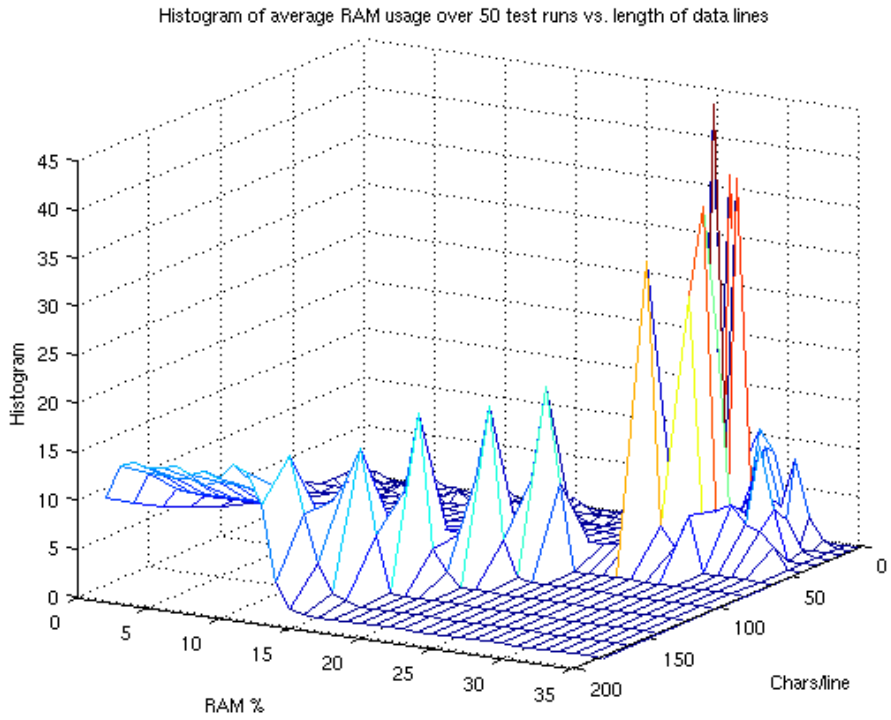

Figure 27: Histogram of observed RAM usage while varying chars/line, first view

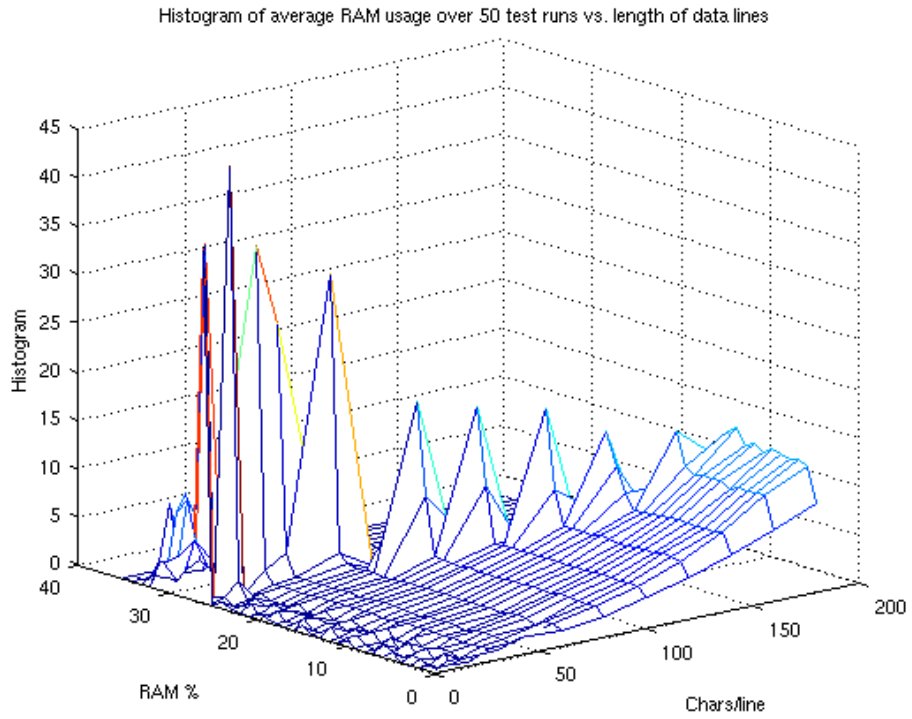

Figure 28: Histogram of observed RAM usage while varying chars/line, second view

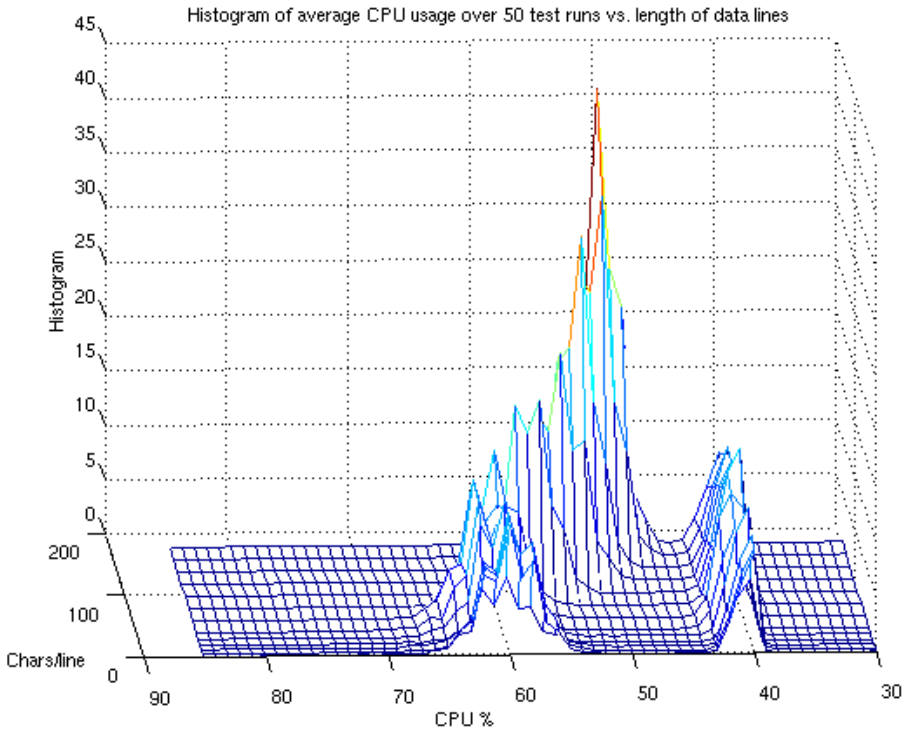

Figure 29: Histogram of observed CPU usage while varying chars/line, first view

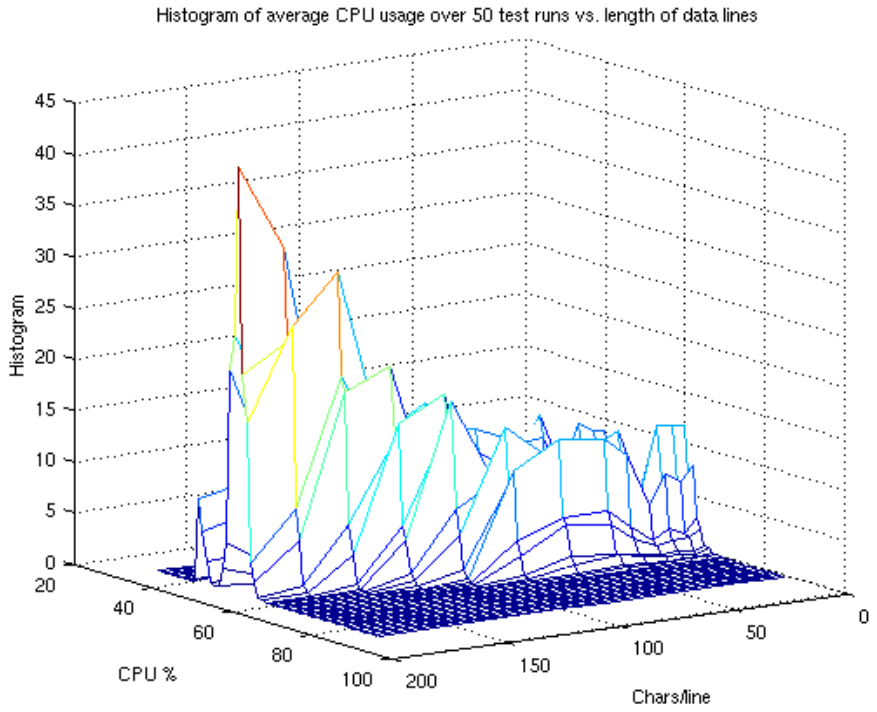

Figure 30: Histogram of observed CPU usage while varying chars/line, second view

3.3.3 Number of queue readers

Since the distributed system's performance relies heavily on how quickly it can get data into and out of the internal queue, it is worth testing whether a larger number of queue reader threads improves the system's performance. It is also important to determine whether this has any adverse effects on the RAM and CPU usage of the system. As shown in Figure 31, increasing the number of readers actually decreases performance past a certain point. This is not a surprising conclusion, given that each writer or reader thread has to lock the queue with a mutex while it is removing data in order to avoid data duplication or missing data due to multiple threads pulling the same information off the queue. As the number of reader threads increases, the data may be pulled off the queue faster, but it is locked most of the time waiting for reader threads to finish accessing the information. If the queue is locked too often, the writer threads are unable to grab the lock themselves and cannot insert data into the queue quickly enough, slowing down the whole system.

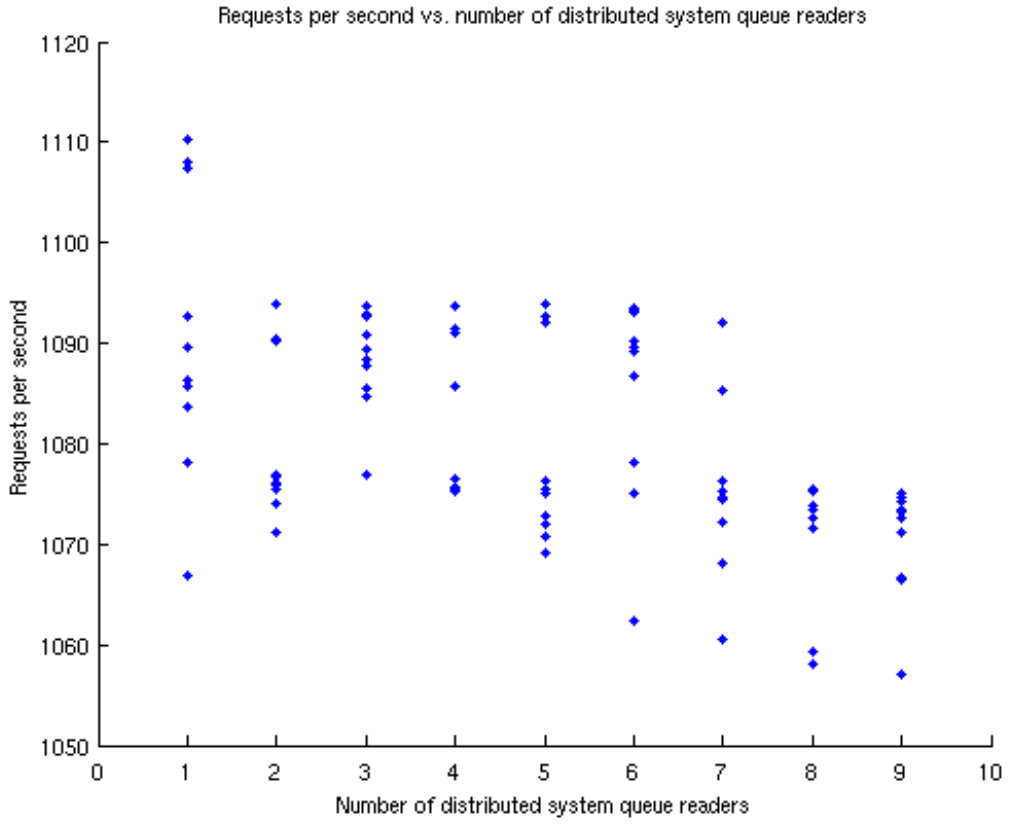

Figure 31: Performance over a varying number of distributed system queue readers

Before running the tests, it was expected that the maximum RAM usage (Figure 32) would decrease as the number of readers increased, since the readers should be able to pull data off the queue quickly enough that the system does not get backed up and use large amounts of RAM. However, since the distributed system queue is locked by a mutex, only one reader thread can remove data from the queue at a time. As the number of readers increases, they reach a point where most of their time is spent trying to gain control of the queue lock rather than actually reading data from the queue and there is no benefit from additional readers beyond this point. CPU usage for these tests is also relatively constant with respect to the number of reader threads (Figure 33), except for the large variation seen when only one thread is being used. The source of this variation is unknown, but could be caused by overheads incurred as the operating system creates and manages threads. The impact of these overheads on performance is likely much more pronounced for one thread than it is for multiple threads.

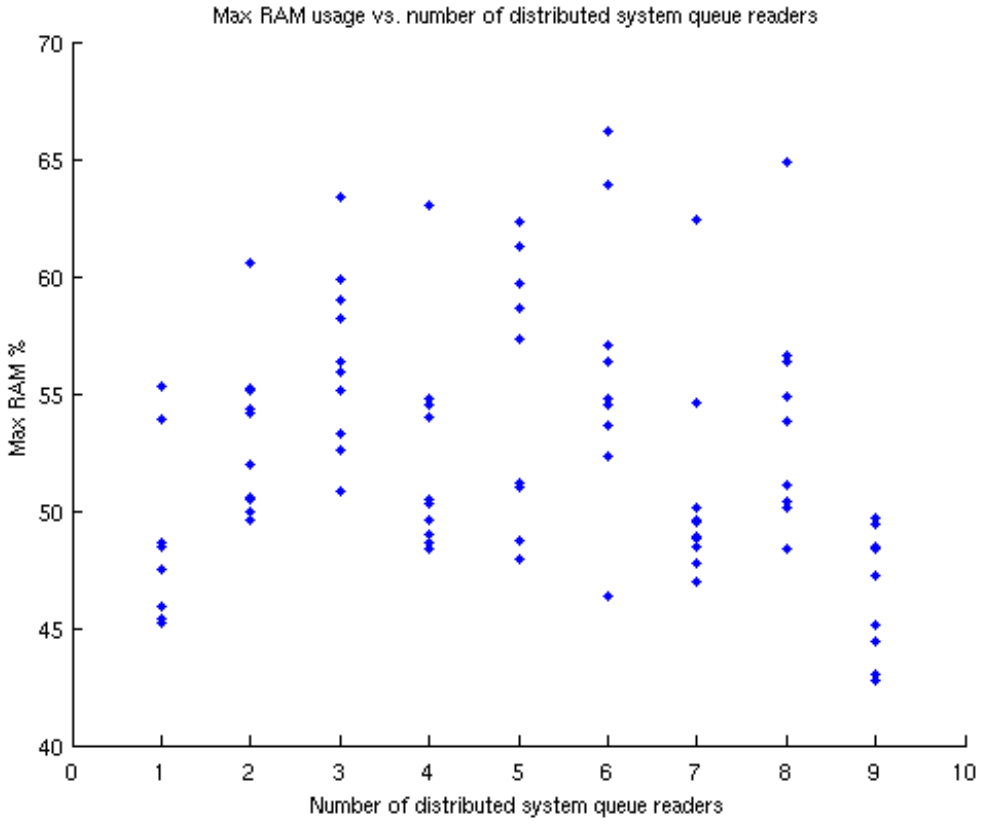

Figure 32: Max RAM usage over a varying number of distributed system queue readers

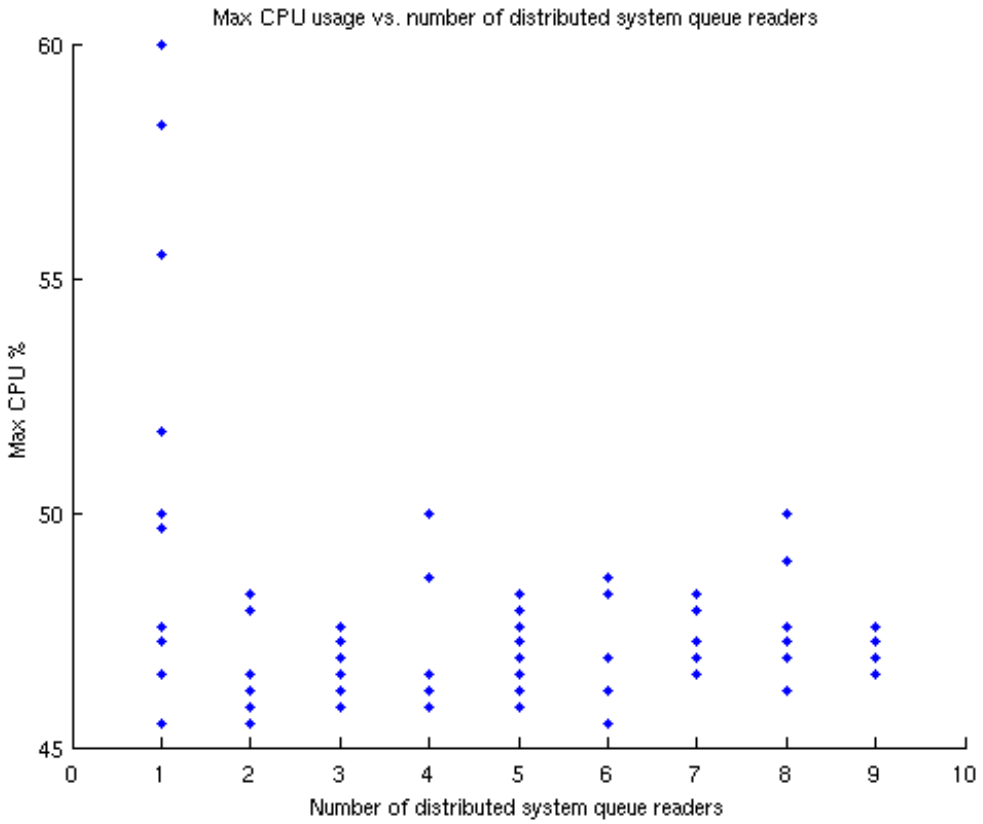

Figure 33: Max CPU usage over a varying number of distributed system queue readers

3.3.4 Throughput tests

Tests were conducted to determine the time required for each line of data to pass from the workload generator through to the distributed system database. The workload generator adds a starting timestamp with second-level resolution to the last line in each file, and all lines are timestamped as they arrive at the database. For the tests below there were 5 lines of data in the file, so the time spent in the distributed system could be calculated for 20% of the data received. Each of the 50 tests processed one million lines of data, but only 200,000 were timestamped on entry. Another Perl script was written to average the times over every 10 values, resulting in 20,000 data points for each test run, as shown in the histogram in Figure 34.

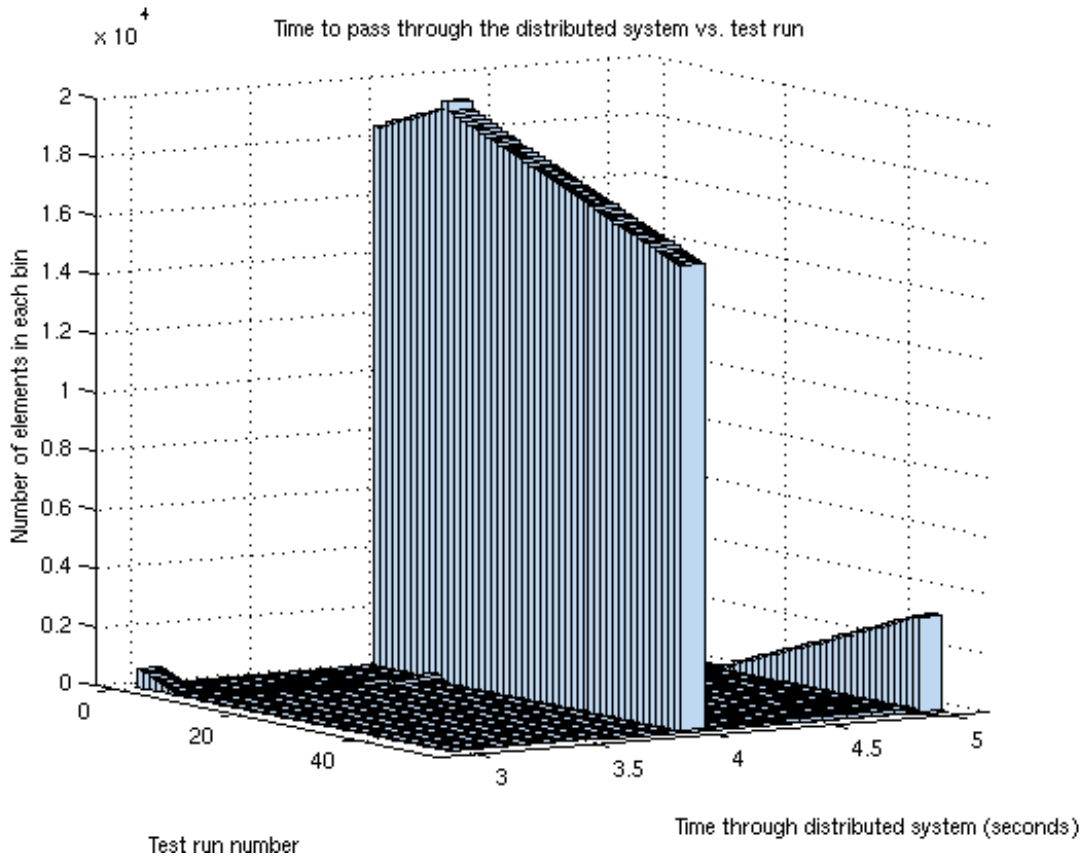

Figure 34: Time through distributed system over 50 runs

All 50 identical tests were run consecutively by the automation script. If the tests were entirely independent, the histogram would look fairly similar across all the runs. Instead, the lines seem to be taking longer and longer to arrive at the database. The distribution of times for the first test runs have the majority of the data at 4 seconds, but with some lines only taking 3 seconds to be received at the database. There are scattered results between 3 and 4 seconds, but none of the lines had greater than 4 second times in these runs. As the tests progress, the tail of the distribution appears to shift toward the higher values, with more and more lines taking 5 seconds to pass through the distributed system. The histogram in Figure 34 was normalized by the number of data points in each test run and the natural logarithm was taken of the results. It was then plotted from two different viewpoints (Figure 35, Figure 36). These plots better highlight that there are some results between the peaks on the graph, but that there are no values between 4 and 5 seconds on the earlier tests, and none between 3 and 4 seconds on the later runs. Figure 36 clearly

shows that the number of lines taking 3 seconds to arrive at the database ramps down as the tests begin, and Figure 35 demonstrates that more and more lines are taking 5 seconds to pass through the system as the tests continue. The increase in 5 second processing times is not entirely due to the elimination of 3 second times, as Figure 34 shows a decrease in 4 second values as well.

The source of these progressively longer response times is currently unknown, though it is likely related to the pattern of disk usage on the database machine. The distributed system mostly holds information in memory, but the database writes to disk. For the first test runs it is likely easier to find stretches of disk space in which to store the database records, but as tests continue the disk becomes more and more fragmented and longer seek times are required.

This behaviour demonstrates that even a fairly simple distributed system can exhibit complex behaviours. The distribution of response times changes substantially across the runs. The distributed system has not actually failed over the course of these tests, and whether or not this behaviour is significant will depend on the purpose and requirements of the system. If the system's developers need to guarantee that every request will be complete within 4 seconds, then this behaviour is important to understand and avoid. If only 5 tests had been run the changing behaviour would not have been noticeable, as response times around 5 seconds are only seen in tests after this point. This makes a case for the need for repeated testing of distributed systems, whose behaviour may change drastically across multiple runs.

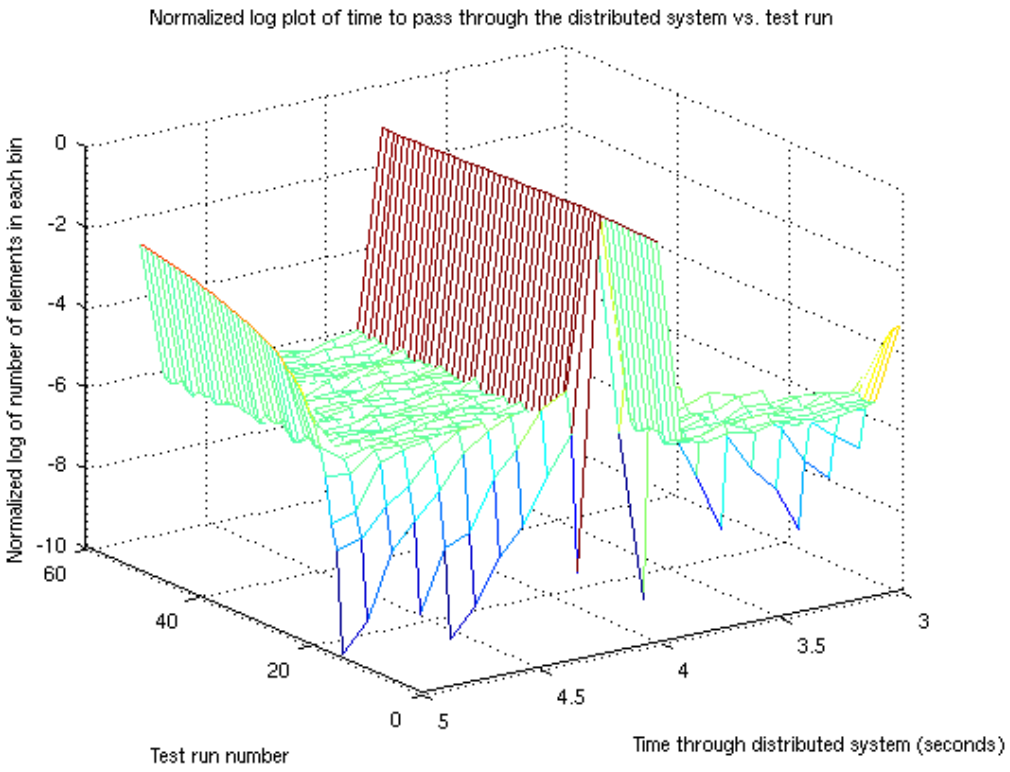

Figure 35: Normalized log plot of time through distributed system, first view

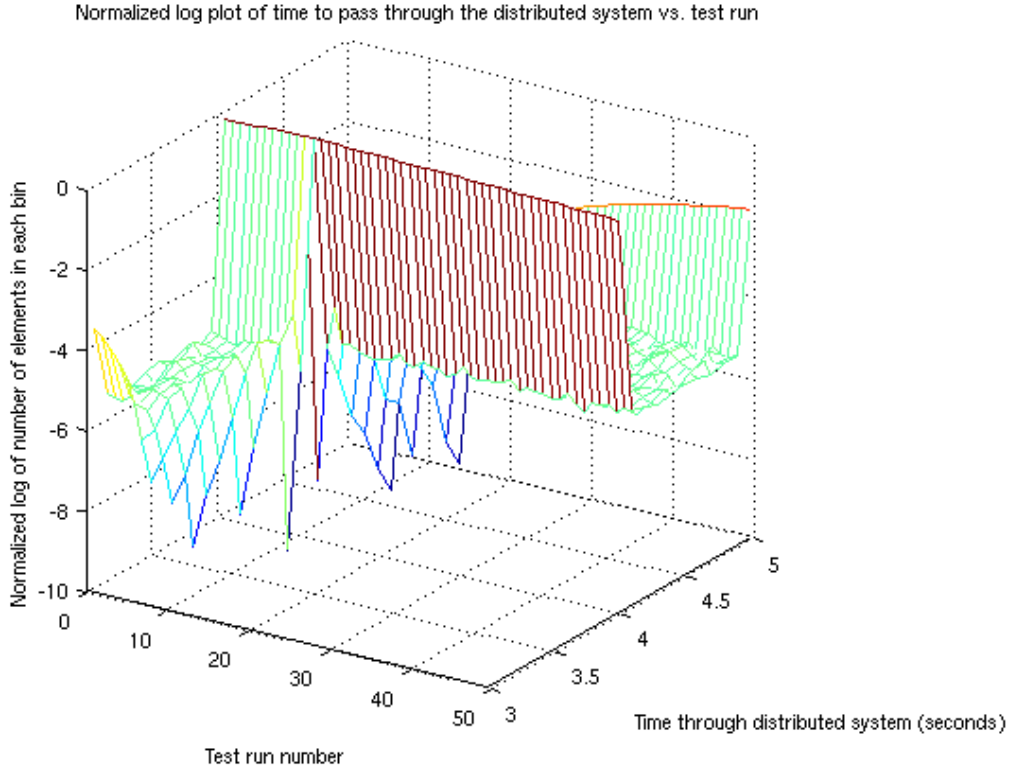

Figure 36: Normalized log plot of time through distributed system, second view

3.4 Summary

A prototype distributed system was developed and the test harness' effect on its performance was explored. While it is impossible to completely prevent the test harness from impacting the performance of the system under test, the current design seems to minimize this impact, with the exception of the increased RAM usage. For distributed systems which already use a great deal of RAM, this impact is significant and may not be easily resolved with the addition of extra RAM to the machines involved. The harness was then used to run several performance tests on the prototype distributed system. These results demonstrate how the behaviour of a distributed system changes with variations in the configuration parameters and between multiple runs of the same test.

Chapter 4: Using the Test Results

4.0 Introduction

The distributed system testbed provides the tools for examining the performance of various distributed systems. The results obtained can be used to improve the operation of the system under test. Two examples of this are discussed here: rewriting the distributed system based on performance results and selecting the optimal parameters to the distributed system.

4.1 Distributed system redesign

The test harness allows a tester to explore the behaviour of a distributed system, and these results can then be used to rewrite the system and improve its performance. This was useful in the case of a prototype proprietary commercial system which was examined using the testbed. This distributed system is a web mining application that receives HTTP posts, parses them for the relevant information and stores it in a database. The number of lines in each post is a key parameter of this system. The testbed was used to explore the effect that changing this value had on the time required for each post to move through the system. The results are shown in Figure 37 with the throughput times normalized. When the line count of each post is small, the throughput of the system is fairly consistent over the course of a test run. As the line count increases, the variation also increases. With 60 lines per post, the posts at the end of the test run take much longer than the ones at the beginning of the test.

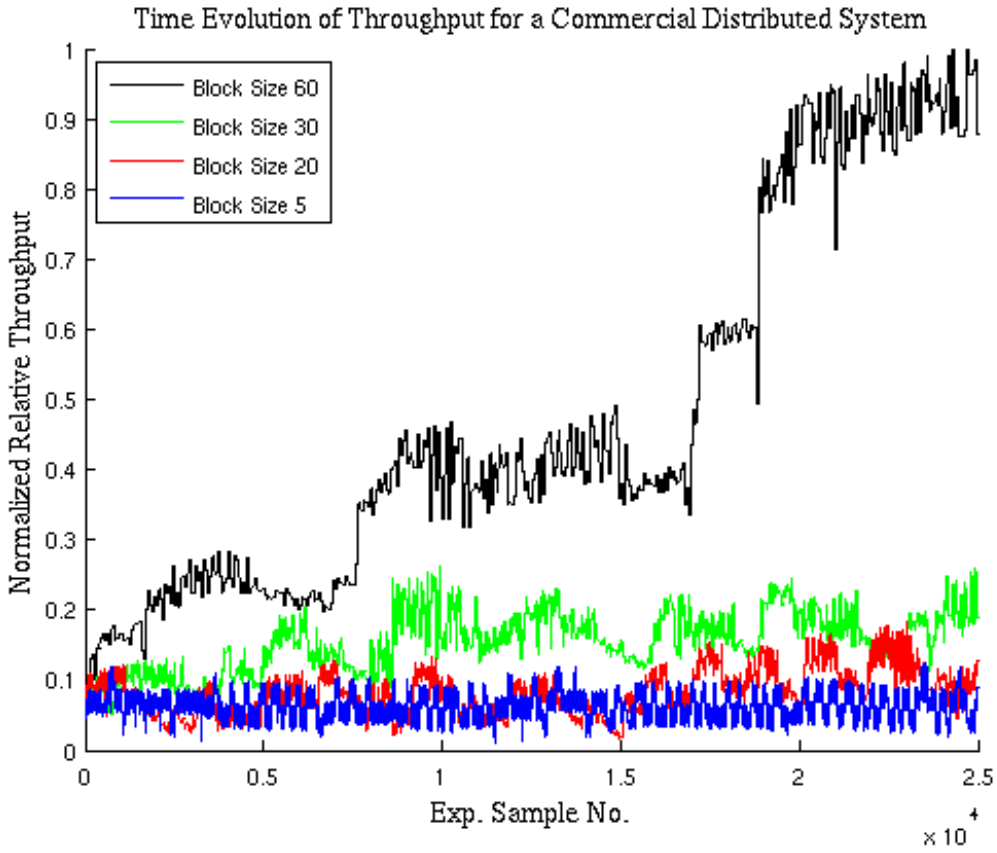

Figure 37: Relative throughput of a prototype commercial distributed system

Figure 38 shows the data of Figure 37 extended to a much longer run involving the servicing of ten million lines instead of the previously shown twenty-five thousand. From this longer run it becomes apparent that the behaviours of Figure 37 only show part of the picture. Figure 38 shows that the upward trend continues until a saturation point is reached, most likely due to thrashing. For the test run with 90 lines per post, the behaviour of the system deteriorates, as the thrashing effects lead to the TCP connection protocol itself becoming backed up. At this point, the overhead associated with dealing with page swaps is large enough that the computer becomes inefficient at dealing with TCP connection requests, producing a rising number of retransmissions which increasingly overload the network. The drop shown at the end of the 10 million data point run is due to the termination of the experiment and is not due to improvements in the performance of the distributed system. It should be noted that this longer test run

required approximately 350 hours of run-time across the full 42 available blade machines, but it does highlight that distributed system behaviours can change considerably over longer test runs.

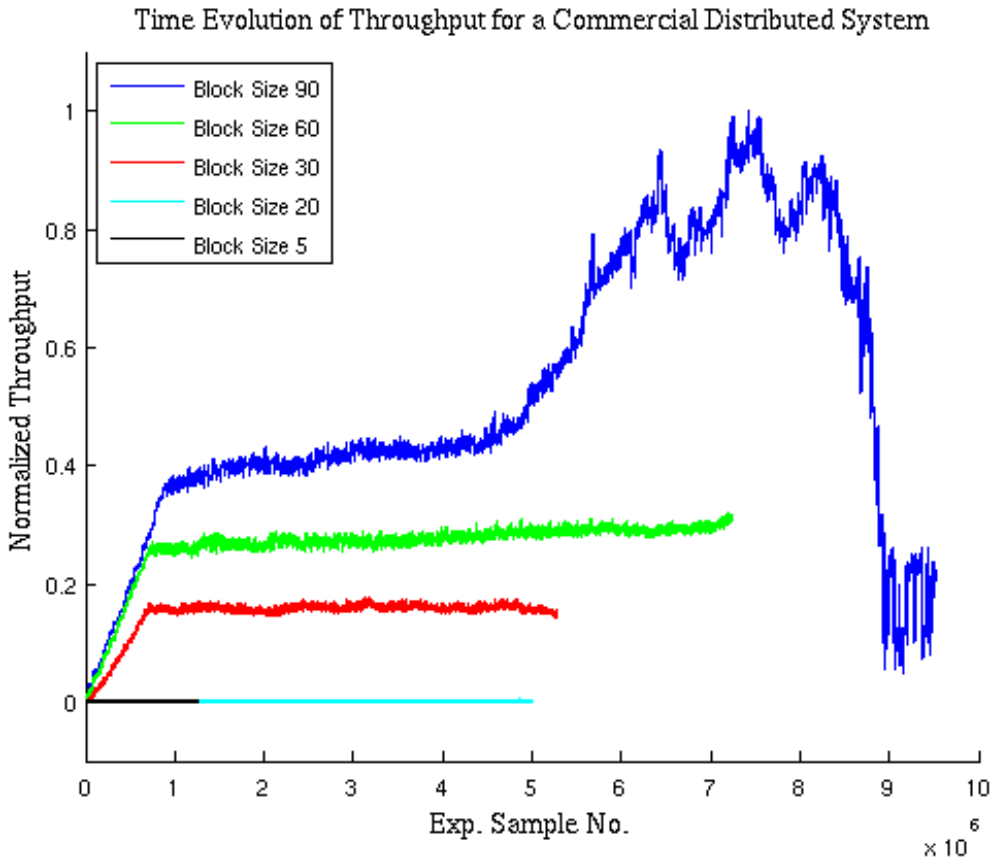

Figure 38: Throughput of a prototype commercial distributed system, 10 million items

To reduce the variation over the course of a test run, the architecture of the commercial system was changed to that of the prototype distributed system discussed in Chapter 3. Initially, this new system was tested without any delays to simulate the time spent parsing lines in the commercial system. This produced the results shown in Figure 39, where the processing time is not affected by the number of lines per post and remains fairly consistent across the course of each run, as demonstrated by the error bars.

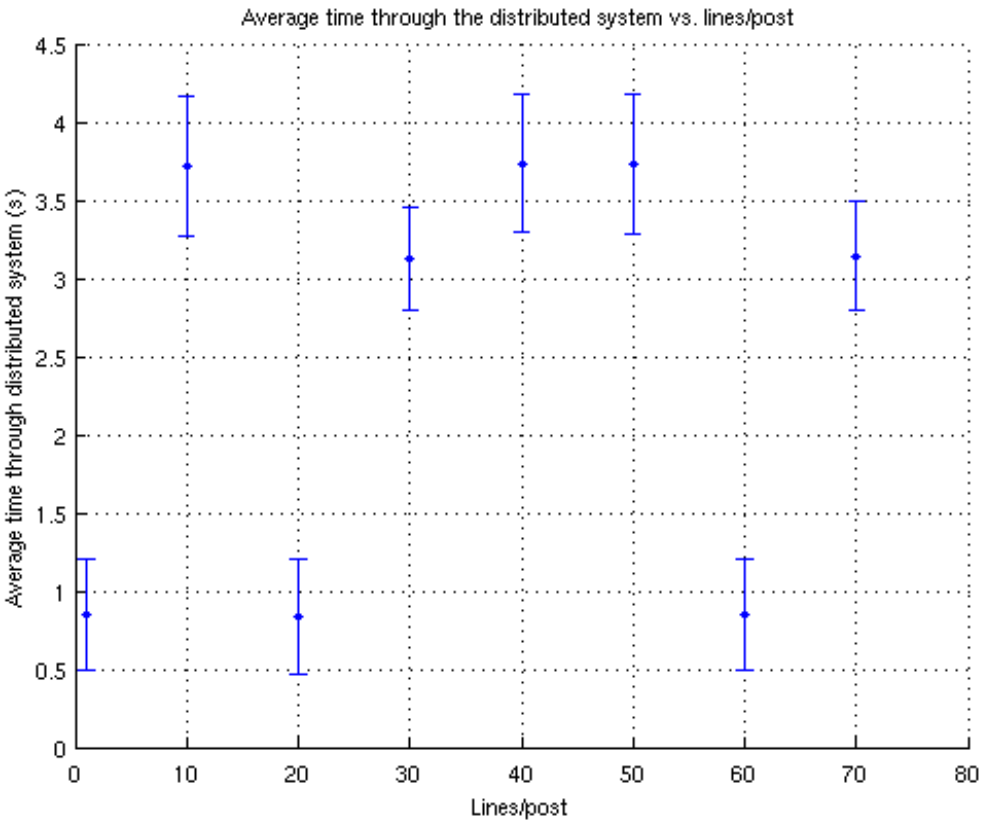

Figure 39: Average time through the distributed system vs. lines/post with no added delays

The prototype distributed system was then altered to include delays in the queue readers. The delay time was selected from an exponential distribution, and the software ran a busy loop for the length of the delay. This simulates the time spent for the queue readers to process each line of data passing through the system. The results over each test run are shown in Figure 40. There is much less variation in the throughput of this distributed system, but the time spent processing each line increases steadily over the course of each test run. This is because the queue writer still runs at top speed inserting each line into the queue, but the queue readers are slower at removing elements from the queue. The entire workload is timestamped and placed in the queue at the beginning of the test and then is removed slowly over the course of the test run, with the last items taking nearly the whole duration of the test run to be received at the database and timestamped there.

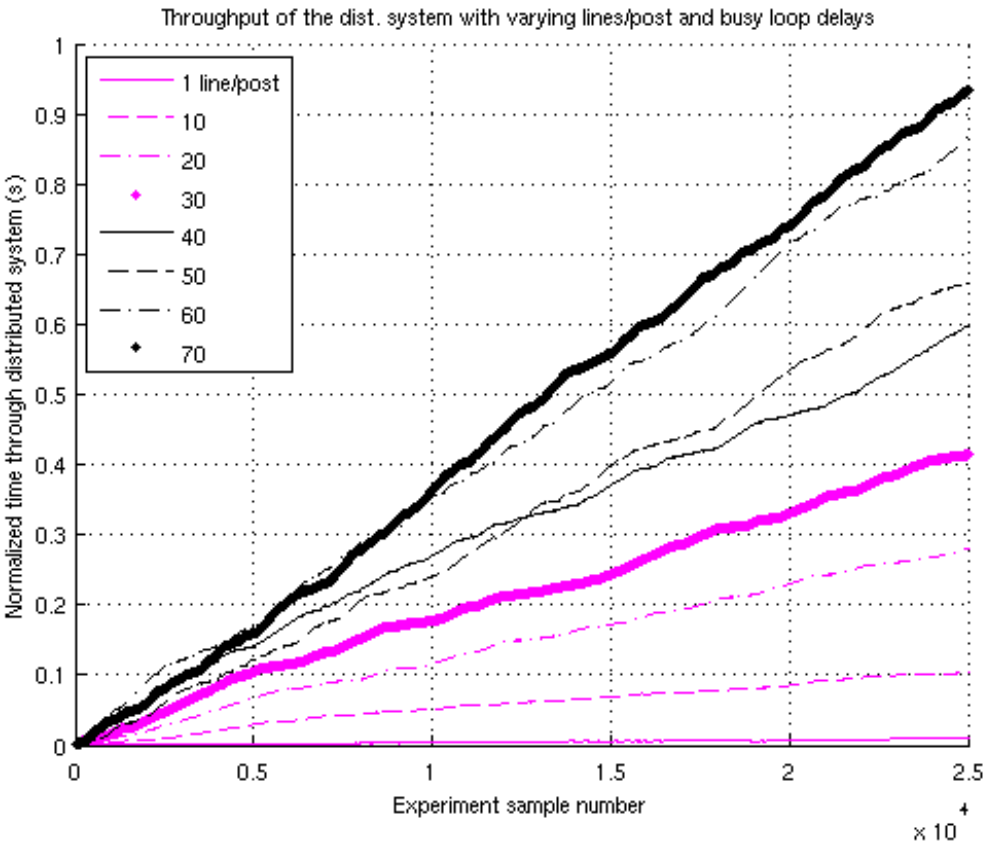

Figure 40: Throughput of the distributed system with varying lines/post and busy loop delays

This redesign is one example of how the distributed system test harness' results could be used to improve the operation of a distributed system. If the system had not been tested with a variety of post sizes, the large variation would not have been obvious. The test harness automates the process of running consecutive tests while changing the number of lines per post for each run.

The longer ten million data point run was not performed on this re-architected solution due to time constraints and availability issues with the physical cluster. But the new architecture approaches the problem by having the writer insert data onto one end of the queue while the readers remove it from the other end. This design means that the middle of the queue, although it can grow quite large, does not need to be held in main memory and can be page swapped to disk as needed, until the readers require it to be re-loaded in memory. This approach makes it unlikely that the thrashing issues of Figure 38 would

occur in this new architecture. Instead, the worst case would be that page swap delays would be incurred each time the writer filled a new page and needed to request the next blank memory page to write to. This could be made more efficient by requesting a larger block of memory for the writer each time the computer's main memory became full, where this would need to be tailored to match the base operating system's page swap algorithm. Since the thrashing would not occur its cascade effects onto the network would also not occur.

4.2 Parameter search

Another use of the distributed system testbed is to determine the optimal, or near optimal, test parameters to produce a certain result, *e.g.* the highest number of requests per second the distributed system can handle. These results can be used to improve the design of a distributed system. If optimization tests conclude that a specific parameter value produces the optimal distributed system performance, this parameter can then be changed in the production system.

Though the example discussed below only involves a simple search based on one input variable, multivariate optimization problems could be solved via this same approach. The mathematical tools for optimization are well-known, but cannot be used for distributed system optimization unless there is some method of providing input parameters to the system and recording the results. This can be accomplished using the automation features of the testbed, as well as a script that analyses the results of each run between tests and selects better input parameters for the next run. This script is written by the tester using the appropriate optimization algorithm for the particular test.

4.3 Prototype distributed system example

Searching for optimal test parameters will be demonstrated using the prototype distributed system (Chapter 3) along with a search script. This script varies the number of lines in each incoming post to produce the minimal standard deviation in requests per second over multiple runs. The number of lines in each post is a candidate for this search since it is an important parameter in the commercial distributed system described above. Increasing the number of lines per post was also predicted to reduce the effect of the processing overhead per line, providing an opportunity for this parameter search. Figure

41 shows the spread of requests per second (RPS) processed by the distributed system when each test is run 100 times. The large variation for small posts is likely caused by the aforementioned overhead required to process each incoming post, and the effects are reduced as the number of lines increases. The search is designed to reduce the standard deviation of the results by finding the minimal slope of the graph in Figure 42.

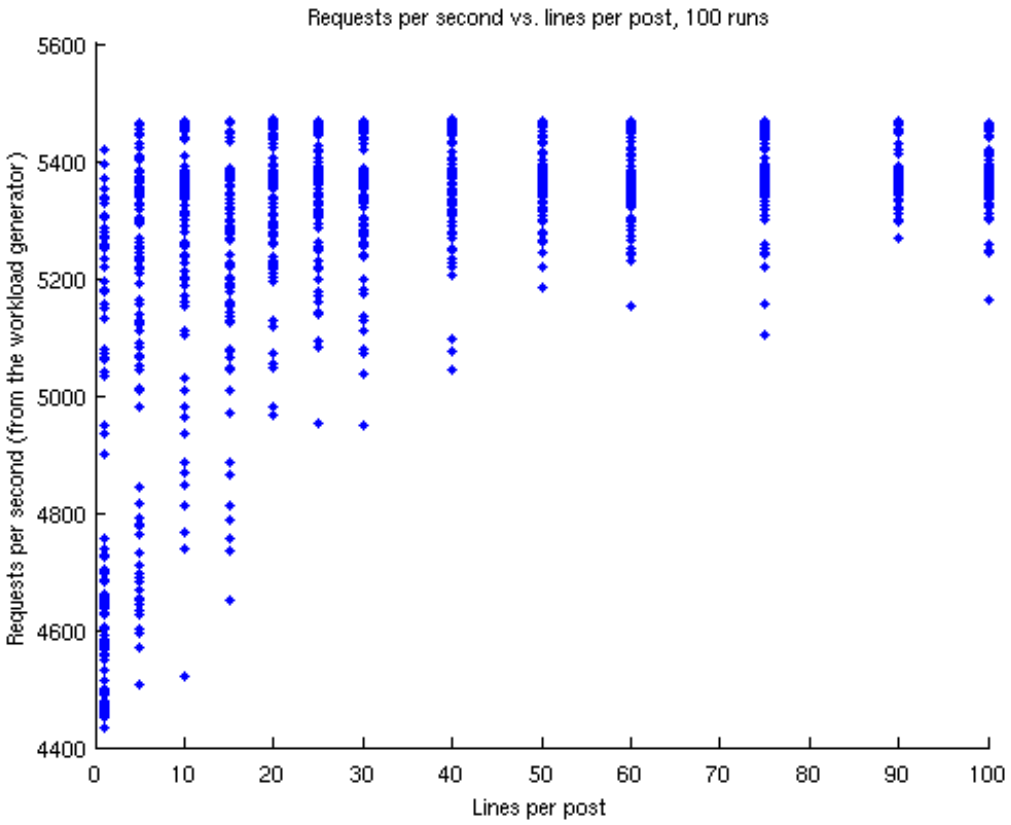

Figure 41: Requests per second vs. lines per post, 100 runs

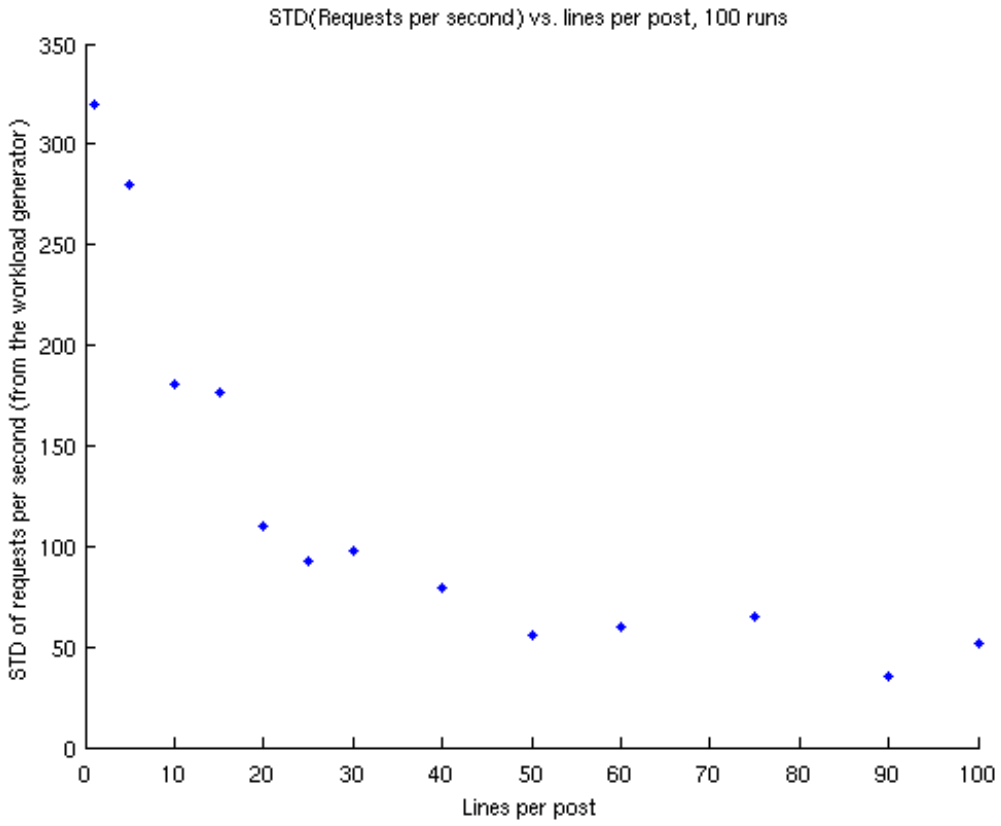

Figure 42: Standard deviation of requests per second over lines per post

Since the standard deviation plot for this set of tests is a simple downwards slope that levels out at the end, a search is performed rather than the more complicated optimization that would be required for a plot with multiple local minima. The search script runs tests until the slope between the standard deviation of the last test runs and the standard deviation of the current runs is less than an acceptable tolerance value. For the prototype distributed system, the tolerance was set to 0.1 based on the results from the initial test run shown in Figure 42. The optimization script was set to start at 1 line per post and increase by 10 lines per post, running each test 20 times. The results are as shown in Figure 43 and Figure 44. The script terminated at 81 lines per post, approximately where expected from the initial runs. At this point, the slope was 0.052, well below the tolerance of 0.1.

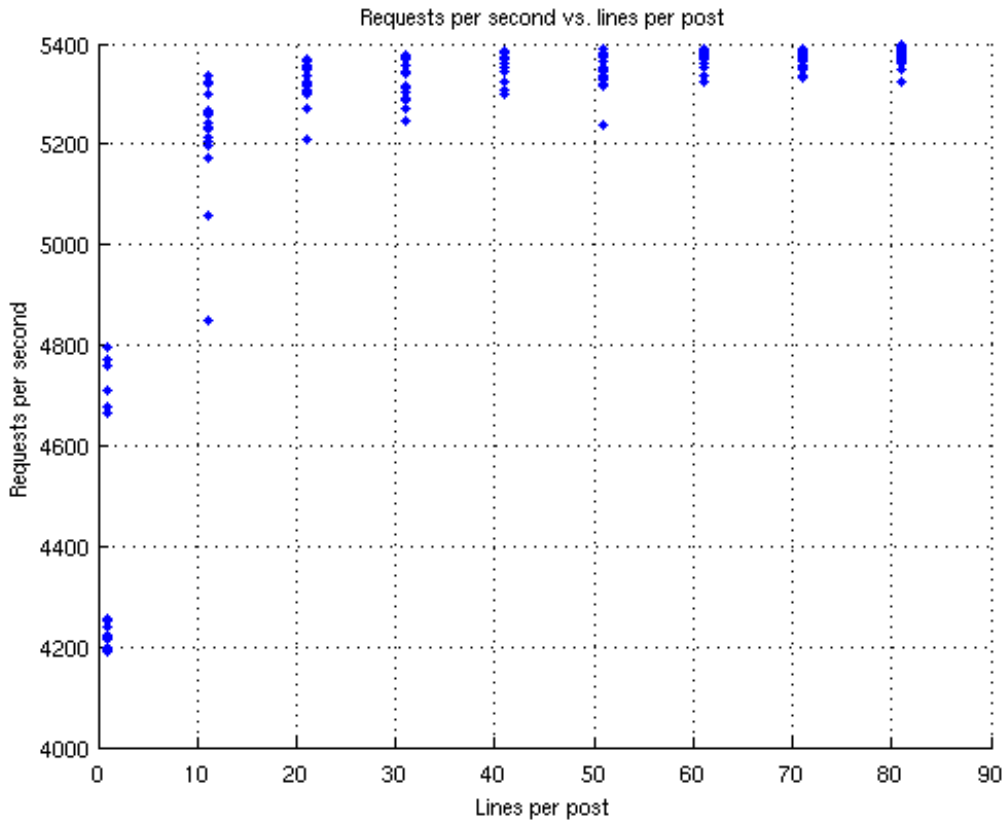

Figure 43: Requests per second vs. lines per post, one search test run

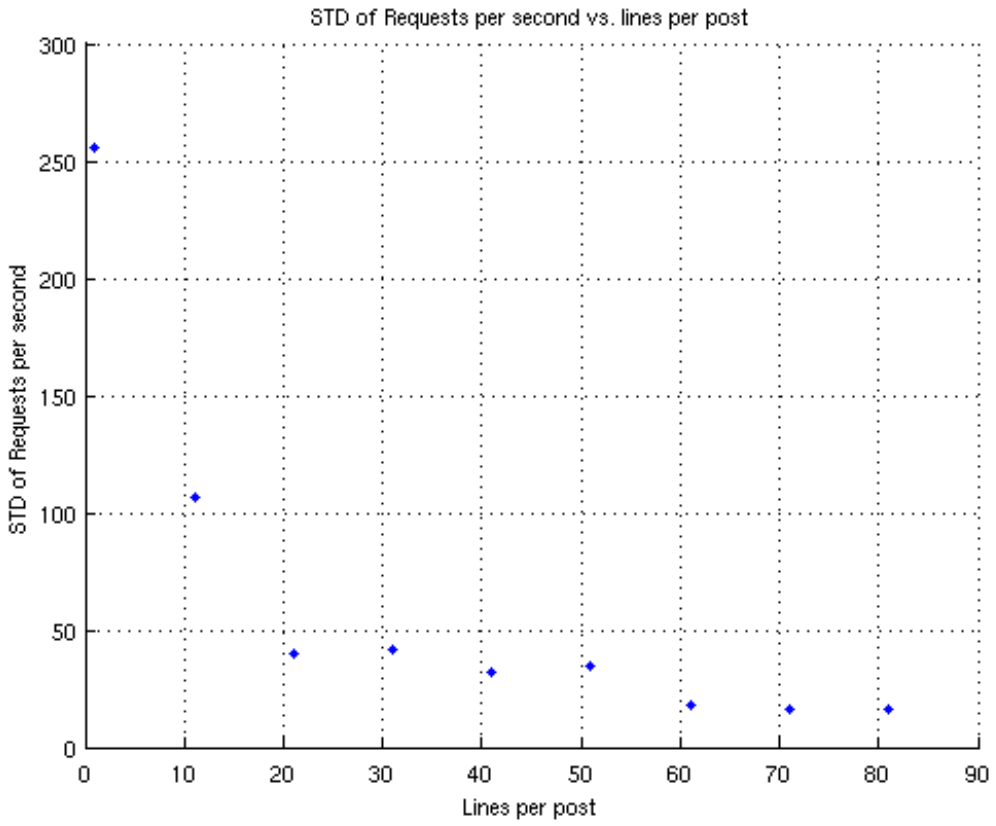

Figure 44: Standard deviation of requests per second, one search test run

Not all test runs will terminate at 81 lines per post. Each time the search script is run, a curve of standard deviations similar to the one in Figure 44 is produced, but there is some variation in the results. To generate Figure 45 below, the average of all these standard deviation points across 15 runs was taken, and plotted along with error bars showing the standard deviation of this value. This graph shows a large variation in the standard deviations of test runs with 1 line per post, with significant variation happening even for post files containing more lines. This variation makes the search difficult, since the curve might plateau earlier than expected on some runs and take a long time to level out on others, depending on where the standard deviation values fell. The search script would need to be run multiple times in order to give a conclusive answer about the optimal parameters for this particular distributed system.

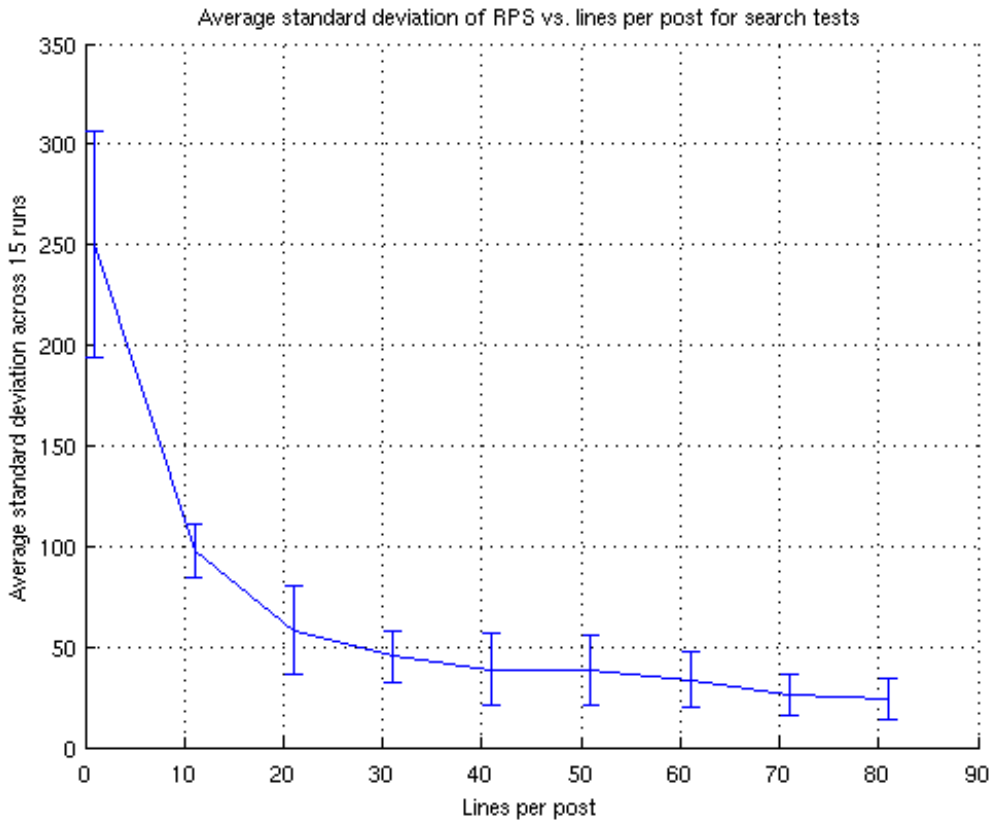

Figure 45: Average standard deviation across 15 tests while varying lines per post

4.4 Summary

Two examples of how the test harness could be used to improve the performance of the distributed system have been presented. In the first example, a commercial distributed system was rewritten to reduce the variation in performance and provide much more predictable results. The second example demonstrates how the performance of an existing distributed system can be improved by finding an optimal value for a configuration parameter.

Chapter 5: Conclusions

5.1 Conclusions

A prototype distributed system test harness has been developed and its components explained in this thesis. It includes a data collection system for receiving and storing results as well as user configuration, test automation, and timestamp processing scripts. These can be used along with the existing workload generator (Appendix A) to run a wide variety of tests on distributed systems. Their performance under changing parameters can be examined, as well as the variation across multiple runs of the same test. The harness can also be used to discover at what point a distributed system fails.

The test harness was used for testing a prototype distributed system. The goal was to minimize the impact of the test harness on the performance of the system, and the final implementation does seem to have the least impact of all the examined solutions. It should be noted that RAM usage did increase on all machines using the data collection hooks. This could be a concern for distributed systems with high RAM usage, and would need to be monitored to ensure that tests on these systems did not exceed the RAM capacity of the system. Several tests were then run using the test harness to explore the operation of the distributed system itself. These tests showed how the distributed system's performance characteristics changed as the input parameters to the system changed, and how the system's performance was not consistent across multiple runs of the same test. These results demonstrate why it is important to run each test on a distributed system more than once, as one run does not give a complete picture of the underlying distribution that the system's measured parameters follow.

The test harness was also used to show how the test results can be used to improve the performance of a distributed system. One example demonstrated how a commercial system was tested and found to have very unstable and unpredictable behaviour. After it was rewritten, the same tests could be run on it to show how the performance was much more consistent over the course of each run. The other example was a simple search procedure to find the optimal parameter value for the distributed system. The result could be used when selecting the parameters for the system in a production environment.

The test harness also allows the tester to substitute a more complex optimization procedure for the simple search procedure, if the system being optimized contains many local maxima and minima.

This test harness provides support for testing distributed systems which contain the hooks required to report results back to the harness. It has been designed to be flexible, allowing the distributed system testers to configure whichever tests they required to fully explore their system.

5.2 Future work

Since the current distributed system test harness is a prototype version, there are several ways it could be improved before being used in a production situation. A fully operational testbed should allow the developers of a distributed system to easily place the hooks required for collecting results into their system and configure and run the relevant performance tests. The automated scripts currently start the prototype distributed system and should be reworked to allow developers to supply startup and teardown commands for the system they would like to test. These scripts also assume that every test will succeed and view success as the arrival of all workload items arriving at a specific database. The testbed should provide for other methods of verifying success and should also provide a timeout configuration option, where a test is considered failed if the timeout is reached. These improvements would provide greater support for a variety of distributed systems.

Bibliography

[1] D. A. Menasce and V. A. F. Almeida. *Capacity Planning for Web Services*. Prentice Hall PTR. Upper Saddle River, NJ, USA. 2002. pp. 111.

[2] J.C. Little, "A Proof of the Queuing Formula L=(lambda)W," *Operations Res.*, vol. 9, pp. 383-387, 1961.

[3] V. Paxson and S. Floyd, "Wide area traffic: The failure of poisson modeling", *IEEE/ACM Transactions on Networking*, vol. 3, issue 3, pp. 226-244, June 1995.

[4] A. Khoumsi, "A Temporal Approach for Testing Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 28, issue 11, pp. 1085-1103, November 2002.

[5] OPNET Technologies Inc., "OPNET Modeler", http://www.opnet.com/solutions/network_rd/modeler.html, last accessed date: 5/6/2008.

[6] H. Balakrishnan, S. Bajaj et al., "Ns-2 network simulator", http://nsnam.isi.edu/nsnam/index.php/Main Page, last accessed date: 5/6/2008.

[7] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for Internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, issue 1, pp. 17-32, February 2003.

[8] L. Peterson, A. Bavier, M. Fiuczynski and S. Muir, "Experiences Building PlanetLab," *Proceedings of the 7th Symposium on Operating System Design and Implementation*, pp. 351-366, 2006.

[9] J. Duerig, R. Ricci, J. Zhang, D. Gebhardt, S. Kasera and J. Lepreau, "Flexlab: A Realistic, Controlled, and Friendly Environment for Evaluating Networked Systems", *Fifth Workshop on Hot Topics in Networks (HotNets-V)*, November 2006.

[10] "Xen". Available at: http://www.xen.org/. Last accessed date: 20/10/2008.

[11] P. Apparao, S. Makineni and D. Newell, "Characterization of network processing

overheads in Xen," *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, pp. 2, 2006.

[12] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," *Proceedings of the 5th Symposium on Operating System Design and Implementation*, pp. 255-270, December 2002.

[13] "FreeBSD Handbook: Jails". Available at: http://www.freebsd.org/doc/en/books/handbook/jails.html. Last accessed date: 14/01/09.

[14] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga and S. Schwab, "Design, Deployment and Use of the DETER Testbed," *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*, pp. 1, August 2007.

[15] DARPA Broad Agency Announcement: National Cyber Range, Strategic Technology Office. DARPA-BAA-08-43. May 5, 2008.

[16] J. Sommers, H. Kim and P. Barford, "Harpoon: a flow-level traffic generator for router and network tests," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, 1st ed., pp. 392, June 2004.

[17] D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," *Proceedings of the 1998 Internet Server Performance Workshop*, pp. 59-67, June 1998.

[18] "ab - Apache HTTP server benchmarking tool". Available at: http://httpd.apache.org/docs/2.2/programs/ab.html. Last accessed date: 27/5/2008.

[19] H.F. El Yamany, M.A.M Capretz and L.F. Capretz, "A Multi-Agent Framework for Testing Distributed Systems", *30th Annual International Computer Software and Applications Conference*, vol. 2, pp. 151-156, September 2006.

[20] D. Hughes, P. Greenwood and G. Coulson, "A Framework for Testing Distributed Systems," *Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pp. 262-263, August 2004.

[21] J. Moe, D.A. Carr and M. Patel, "Using Observation and Refinement to Improve

Distributed Systems Test," *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pp. 153-159, March 2003.

[22] "libconfig". Available at: http://www.hyperrealm.com/libconfig/libconfig.pdf. Last accessed date: 20/10/2008.

Appendix A: Workload Generator

Overview

A workload generator has been developed by Chris Mueller as his Masters thesis. The workload generator produces input to a distributed system which can be tracked as it passes through the system. The workload generator can be distributed across multiple machines in order to produce data faster than could be achieved on a single box. It produces "posts" of data, which contain header and body text. As each post is generated and sent to a distributed system, a Unix timestamp is added to the last line with the suffix "&SendTime=", so the sent time can later be compared to the time this line was logged in the database. Several parameters are used to control the input, including the number of simultaneous connections to make, the total amount of data to send, a file containing the data to be sent, the delay between sends, and the parameters of the statistical distribution to follow when producing data. Several of the commonly used distributions for network traffic are available, such as Poisson, exponential, Pareto and constant rate. Once a distribution is chosen, the specific parameters for that distribution can be selected, *e.g.* lambda describes a Poisson distribution, while alpha and beta are needed for a Gamma distribution. All of these parameters are defined in a file called workload.cfg, which uses libconfig [21], and an example configuration is shown in Figure A-1.

```
workload:
{
        debug = TRUE;
};
connection:
{
        hostname = "b02b";
        port = 8080;
        iterations = 10000;
        tracepath = "/root/workload/data/postak3";
        numthreads = 1;
        persistentConnections = TRUE;
};
arrival:
{
        distribution = "poisson";
        alpha = 1;
        beta = 2;
        lambda = 100;
        mu = 1;
};
payload:
{
        headerBodyDelim = "\n\n";
        header = ["sample header information"];
        content = ["sample content"];
}
```

Figure A-1: Sample workload generator configuration

Configuration files created in libconfig can be read directly into a struct in C, which allows the programmer to define what the types of the variables will be without having to cast them correctly. If the effects of changing parameters are of interest, a series of tests can be run with new parameters for the workload generator on each iteration.