
Feedback Control Theory and Processing System Log Streams

by Wei Xu

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor D. A. Patterson
Research Advisor

(Date)

* * * * *

Professor M. Franklin
Second Reader

(Date)

Abstract

In this project, we focus both on building a general tool that can be used by system operators and machine learning researchers for analyzing data, and on exploring general techniques of applying feedback control theory to distributed computer systems.

We have built a scalable and flexible system log processing system using TelegraphCQ [36] as the building blocks. The report is divided into two parts.

In the first part, we discuss the needs for processing system log data as data streams and our general design of parallel and multi-tier architectures.

In the second part of the report, we address problems regarding to this stream processing system as being a complex distributed system itself. We focus on how random disturbances can have an impact on the performance and reliability. All problems addressed are solved systematically with feedback-control-theory. We discuss three uses of the feedback-control-theory. First, as the workload regulator, then the load balancer and, last, the workload simulator. We illustrate the general techniques for the system identification and the controller design by presenting the three applications.

Contents

I	Processing System Event Data as Data Streams	4
1	Introduction	5
1.1	Motivation	5
1.2	Summary of Results	11
2	Processing system logs as data streams	14
2.1	The required processing on system logs	14
2.2	Traditional methods are not suitable for this application	17
2.2.1	Ad-hoc scripts are not enough	17
2.2.2	Problems with relational databases	19
2.3	Stream model of system log data	19
2.4	Building a parallel system using TCQ as building block	22
3	Related Work	24
3.1	Automated system problem detection	24
3.2	System monitoring and management	26
3.3	Stream data processing and mining	27
3.4	Applying control theory to computer systems	28
4	A Flexible Architecture for Processing System Logs	29
4.1	Key component, Telegraph Continuous Query Processor	29
4.2	Description of the parallel architecture	30
4.2.1	Two-tier parallel architecture	30
4.2.2	Building blocks	31
4.2.3	Implementation	33
II	Improving System Reliability and Performance with Control Theory	34
5	TCQ Flow control	36
5.1	Flow control issue in TCQ	36
5.2	Control problem formulation	38
5.3	System Identification and Controller Design	40
5.3.1	System Identification	41
5.3.2	Controller Design	44
5.4	Assessment	47

6	Building the load balancer with control theory	50
6.1	Effect of imbalances	50
6.2	Control problem formulation	52
6.3	System Identification and Controller Design	53
6.3.1	System Identification	53
6.3.2	Controller Design	54
6.4	Assessments	55
7	Discussion	56
7.1	The advantages of using control theory in computer systems	56
7.2	Limitations of control theory in computer systems	58
8	Conclusion and Future Work	60
A	Controller for load simulator	62
A.1	The inaccuracy of load simulator	62
A.2	Control problem formulation	63
A.3	System Identification and Controller Design	64
A.3.1	System Identification	64
A.3.2	Controller Design	65
A.4	Assessment	69

List of Figures

4.1	A general structure of our parallel system	30
5.1	Behavior of TCQ node without regulating result queue length.	37
5.2	PI Controller for regulating the free space in the queue.	38
5.3	Not carefully chosen workload causes the queue length to fill up.	41
5.4	Workload for queue length system identification	42
5.5	Models Constructed with good and bad workloads.	43
5.6	Matlab routine that does the parameter estimation.	44
5.7	Matlab routine for calculating K_P and K_I of PI controller	48
5.8	TCQ node with result queue length controller, under CPU contention	49
6.1	The Effect of disturbances with and without control	51
6.2	The block diagram of controller in load balancer	52
7.1	An unstable control system	57
7.2	Block diagram of the unstable control system	57
A.1	A naive implementation of load simulator cannot achieve the desired load	63
A.2	Block diagram of workload simulator with P controller	64
A.3	Block diagram of workload simulator with PI controller	64
A.4	Effect of P controller in workload simulator	70
A.5	Effect of PI controller in workload simulator	70
A.6	Effect of P controller in workload simulator (zoom in)	71
A.7	Effect of PI controller in workload simulator (zoom in)	71

Part I

Processing System Event Data as Data Streams

Chapter 1

Introduction

In this chapter, we will introduce of the idea of using computer system logs for automated failure detection. We first discuss the difficulties of dealing with a large amount of system logs data. We then present our dealing with those logs as data streams, and build a flexible stream processing system using TelegraphCQ [36, 28] as building blocks. This stream processing system itself is a distributed system with many components running in parallel. We conclude by discussing issues in its scalability and reliability, and our approach to solving those problems with feedback-control-theory.

1.1 Motivation

System fails and we have to cope with it

Most on-line services, such as Amazon and eBay, suffers from various user-visible failures. As reported by various authors (such as Oppenheimer *et al.* [30]), the most common causes of failures in Internet services are *software bugs*, *human operator errors* and *hardware failures*, respectively. These failures frequently cause system to crash, which is very expensive; for example, Patterson estimates that one hour of downtime of a large on-line service can cost up to one million U.S dollars [32]. Predicting, detecting and localizing these failures are very difficult tasks, as some systems consist of hundreds of software components running on up to 50,000 servers [29].

Failure-free computing, if ever possible, is still in the future. *Recovery oriented computing (ROC)* specialized in coping with inevitable failures through fast recovery [31]. This approach suggests the goal of ROC - reducing *meantime to recover (MTTR)*. System availability A is defined as $A = MTBF / (MTBF + MTTR)$, where MTBF is the meantime between failures. Since $MTBF \gg MTTR$ applies to all systems, reducing MTTR even a little bit can greatly improve the system availability.

Two factors can affect MTTR: the time to diagnose a problem and the time to fix it. Between the two, the diagnose time is the dominant factor[30]. The results of ROC project provide techniques for separating applications and session states store[23], organizing components for fast rebooting[5], and supporting system-wide undo operations[4]. With these techniques, we can not only reduce the time required to fix the problem, but also enable the automated system detections and recoveries. This is because even though we may not be very sure about the problem, we can still try the recovery methods out, as we know that our operations would only cause predictable effects and would only affect the system behavior for a relatively short period of time.

In the RADS (Reliable Adaptive Distributed Services) project, we are combining statistical learning algorithms in order to detect the system failures, localizing the root cause of a system failure, and automatically recovering the system from failures with the techniques discussed above. We also applied feedback-control-theory to computer systems, which has proved to be a useful tool for building self-adaptive systems and for increasing system reliability.

System logs are valuable for detecting system failures

Most software systems are instrumented so that they will record various activities. Some of them will also periodically report various statistics, such as processor, memory and network

usage. Developers usually would record information about the current state of the application to facilitate the debugging. In some well managed systems, human operators also record all changes made to the system and all problems found in the system. We refer to all this information as *system logs*.

System logs contain information that is useful to many applications. Traditionally, different kinds of logs are used for different purposes. Access logs are usually used in business data mining, such as modeling user behaviors or supporting business decision making[38]. Lower level logs such as the processor and memory usage logs are used by the system administrators to monitor the system performance and by the managers for resource planning. Application-specific logs are used by developers for debugging and performance profiling.

Recent research has found that all kinds of logs could be used for fast detection of intrusions [19, 35] or system failures[42, 10].

By simply visualizing certain summarized information of the log to human operators, system problems can be diagnosed much faster. Bodik *et al.* shows that by looking at the request counts of webpage transitions, the operators are able to notice the system errors hours or even days earlier [3]. This is because for human operators, when the data are represented in the right form (i.e., the right statistics plus the right representations), can detect normal patterns from abnormal patterns easily and quickly. Another way to analyze the system failures is to use statistical learning algorithms to detect the abnormal patterns of the system data. Many algorithms have been used to solve various problems, and we will discuss works in this area in Section 3.1.

Our belief is that the best way to improve system availability is to combine automated failure detection tools based on machine learning algorithms and visualizing key statistics. In this way, we can both reduce stress on system operators by detecting and fixing simpler problems automatically, and by eliminating the problems of the false positives or false

negatives, which happens in almost all statistical methods. We will investigate methods of doing both in the RADS project.

Both goals require processing logs in real time, which is harder than the traditional business data mining. The latter mainly deals with the historical data. The throughput of this online processing must be sufficient to process a sample large enough to yield a statistically correct analysis. Since this processing is on the critical path of diagnosing the system failures, the delay incurred will directly affect the MTTR, and thus must be abbreviated as much as possible.

Practical problem of analyzing system logs

There is a practical problem that makes the real time processing of log data a hard job: the logs are too large and too complex. The systems today generate as much as 1 TB of log data per day [9]. Using more fine-grained instrumentation for debugging generates many times more log data.

The complexity of system log processing not only comes from the size, but also the arbitrary format a log can have. Widely used system log format of Unix System Logger, *syslog*, contains only three fields in its schema specification, which are timestamp, priority level, and a text string showing the content of the message [25]. Lacking of a specific schema makes the format very flexible, but leaves all the difficult parsing and analyzing jobs to the log users.

These complexities seriously limit the ability for system administrators to make use of the logs. Our experience shows that using ad-hoc scripts for processing data is very tedious and inefficient. Maintaining ad-hoc scripts is another hard task, especially when the log schemas or the system architectures change. From private communications to several system operators of large online services, most system logs are never processed.

These complexities cause problems for machine learning researchers as well. A lot of preprocessing (such as sampling, adding/removing attributes, merging data from different sources, and so on) is required before the data can be used in machine learning algorithms. Machine learning researchers spend a huge amount of time on this preprocessing. Also, machine learning algorithms are usually complex to implement and requires constant modification (like all research codes). When these algorithm implementations are combined with preprocessing, the resulting codes can be hard to read and maintain.

To make things worse, difficulties of processing system logs limit the opportunities of researchers to obtain real data from companies. Logs need to be processed by the companies before they can be donated to the research community. This is because logs usually contain sensitive information. For example, to protect privacy of all clients, a company doesn't want others to know how many requests are coming from a particular client, and what page a specific client is requesting. Therefore, some fields in the logs are masked (usually done by replacing their values with a hash value). Also, the logs may be sampled so that the information of the traffic patterns to those companies is not revealed. With ad-hoc scripts, the company needs to pay a lot of the administrators' time to do these processings, which is unaffordable to many companies.

In order to address the problems above, we need a better data model for the system log analysis, and a scalable, modular architecture that can be distributed over a cluster of machines to process the data quickly.

Making the log processing system scalable and reliable

To address problems above, we designed a software system that processes system logs as data streams. The system makes use of TelegraphCQ (TCQ in short)[36], a general purpose data stream processing engine, and multi-tier processing of system event logs.

Initial experiments show great scalability and flexibility of our data stream processing system. However, this system is a complex distributed system assembled with off-the-shelf components, so it suffers from many common problems as other distributed system. We found the following four issues most serious:

1. The black-box building block may have inherent limits. For example, Chapter 5 shows that TCQ system has some flow control issues that results in data loss.
2. Load balancing is extremely critical to the performance of the entire system. This is true for all parallel computational systems involving synchronization, since the faster nodes must wait for the slow nodes.
3. Each component running in parallel may encounter problems or have additional administrative loads. These random disturbances cause unpredictable performance of the stream processing, causing delays or even losing of data.
4. Sometimes not all the useful measurements can be measured in a black-box system. For example, total queue length in TCQ system is hard to measure without significant changes in TCQ, making the load balancing more complicated.

As this log processing system shares many common problem with general distributed system, we decid to use it as a testbed for general RADS design concepts. As the initial step, we address the issues discussed above by applying feedback control theory. This technique proves to be very useful in distributed system design.

Feedback-control-theory provides a set of mathematical tools for modeling systems and designing controllers, which can be used to regulate certain values, such as the queue length and the data rate, by continuously correcting current errors.

There are many examples of applying the control theory to computer systems. Hellerstein *et al.* provides an extensive review of the current techniques for applying the control

theory to computer systems [18] . We discuss more works related to applying the control theory to computer systems in Section 3.4.

We will provide more background knowledge for readers who are not familiar with the control theory in Chapter 5, with our discussion of coping with the flow control issue in TCQ. We will show that the classical linear control theory and very simple models works well in this system.

1.2 Summary of Results

We developed a scalable software architecture for processing system event logs for large, distributed systems using stream data model and continuous queries, with TelegraphCQ[36] as building blocks. We also used this monitoring systems as a testbed for exploring general techniques of making distributed systems scalable, reliable and manageable by applying the feedback control theory.

A flexible system to process system logs

We built a scalable and flexible system log processing system using Telegraph Continuous Query System [36] as building blocks. This system allows us to preprocess data for any off-line or on-line statistical learning algorithm against massive quantities of system logs.

This system inherited the powerful and efficient continuous query processing features from TCQ. These features include: writing queries with SQL language; defining the log schemas with customizable data types; running queries on multiple streams or static data tables. The evaluation of all continuous queries share intermediate results, and thus is quite efficient. Also, query evaluation plans are adaptive to changes of the data streams.

By running TCQ in parallel and in multiple tiers, we are able to make the system scalable in order to handle the fast rate of the system logs data. Our system can be easily configured to be distributed over a cluster of machines and new log sources and algorithm

can be added on-the-fly. Our system is also self-adaptive to disturbances either from the machine or from the data rate of the incoming log streams. It automatically balances the load over parallel machines to achieve optimal response time and throughput.

This log processing system can be used by system operators to collect, analyze, archive logs, or by statistical learning researchers to preprocess of the raw log data and prepare input for their algorithms for automatic system diagnoses. Of course, this system can be used in situations where other types of stream data needs to be analyzed.

Using control theory to make this infrastructure scalable

Using this log processing system itself as a testbed, we explored general techniques of improving distributed system reliability and scalability using the feedback-control-theory.

We demonstrated three applications of the control theory:

1. Using off-the-shelf system as building blocks, and cope with their problems without changing its internal structures.
2. Dealing with disturbances in the systems
3. Creating new applications with control theory. For example, we created a workload simulator that can generate exact workload despite of the disturbances. We also created a load balancer that split the load automatically to achieve minimal response time.

We also discuss our experience on the advantages and issues of using control theory in computer systems. Applying control theory in computer systems results in the following advantages over heuristics:

1. Correctness can be analyzed.
2. Implementation and system management are easier.

3. It encourages a good system design.

Successfully applying the control-theory to computer systems is not trivial. These problems must be considered:

1. Whether the quantities we want to regulate can be controlled. It is not always that easy to find an controllable knob that we can use.
2. It is not easy to build an easy model of the target system due to the complexity and randomness involved in the software design.
3. Many legacy systems are not designed to be easily measurable.

We show methods of dealing with these difficulties, as well as suggesting new principles for the system design. This is to make a system more “control theory friendly”, which will greatly improve both the system performance and its reliability.

This the report is organized in two parts, the first part discuss about the general idea of using general stream processing techniques to process system log data, the problem and motivation (Chapter 2), and our experimental system design (Chapter 4). We also discuss works in related research areas in Chapter 3. The second part describe how to make our log processing system scalable and reliable. In Chapter 5, we discuss a flow control problem with original TCQ and methods to use control theory to solve the problem. We also describe the process of system identification and PI controller design for people who are more interested the technical details. Chapter 6 describes the implementation of load balancer using control theory. Chapter 7 discusses the lessens learned from this project, especially issues with control theory. Chapter A in appendix describes the design of an accurate workload generator, and also compares P controller with PI controller. Chapter 8 concludes the report.

Chapter 2

Processing system logs as data streams

Distributed computer systems are becoming larger and much more complex. Monitoring the system is no longer a trivial job to do. Systems today can generate as much as 1 TB of log data every day [9]. We used two data sets from online service companies in this project. The first one is an application log consisting of 2.5TB data over a 20 days period, and the other is a DNS access log that has on average 20-30 millions of entries per hour.

When presented with terabytes of data, we need to think about how to handle it efficiently. In this chapter, we describe our experience with analyzing huge amounts of data. We first discuss what kinds of processing of log data we want, and our early attempts to use ad-hoc Python scripts and traditional relational database systems for this purpose. We show that both of them have their limitations. We then show the advantages of modeling them as data streams.

2.1 The required processing on system logs

As described in Section 1.1, system logs contain more useful information than operators can discern. The data might be used to predict that a particular machine is likely to crash in a few minutes, or that a certain software component has a bug or even help to localize the causes of failures. A fully automated analysis or visualizing the information contained in

system logs to system operators greatly helps to reduce the MTTR.

However, both tasks require close to real time processing of system logs. This is because unlike data mining for business analysis, for which historical data are still very relevant, system failure must be detected when they happens (or the best is to predict them before they happen). Studies show that the value of logs decreases quickly over time[17].

Also, as detecting system failures is a new application of many types of logs. The information we need to visualize or feed into Statistical Learning Theory (SLT) algorithm may not be explicitly contained in the log. The SLT algorithms are already complex enough to implement, so we always want to separate the log preprocessing from the machine learning algorithm implementation. In this report, we mainly focus on what preprocessing is done, and how they can be done efficiently. From our experience, useful preprocessing includes the following:

- **Sampling:** we need to sample the original log because we are not able – and it is not necessary – to look at all the observed data. Other reasons for sampling include temporally variable data rate, dealing with unbalanced data sets, removing duplicate entries or removing entries that do not report the class attribute. Thus, a reconfigurable sampling algorithm must be supported.
- **Generating aggregate values:** Neither a system operator nor a complex statistical learning algorithms can get any result by directly looking at the raw log. Intuitively, the data needs to be presented as their “aggregated” values, such as *average* or *count*.
- **Cleaning the data:** we need to filter out unnecessary attributes from each of the event log entries: attributes not reported by any sample and attributes with constant values.
- **Adding new attributes:** original attributes in the raw data are often neither enough nor suitable for analysis using SLT algorithms. Some attributes can be generated

easily, such as; *"is there an error message reported?"*. However, other attributes – such as *"is this log entry anomalous?"* – might require running a simple algorithm.

- **Integrating streams from multiple sources:** System logs are generated on separate machines describing different aspect of the system. For example, our data set contains performance statistics, request log and problem tickets. It is also often necessary to integrate data sources unanticipated at design time, since we might find more related information as our familiarity of the system increases.
- **Generating multiple algorithms:** Applying several different algorithms to our data is an important part of our research. However, different types of algorithms require different experiment setup and many publicly available implementations require different format of input data. Because of huge amounts of raw log data, accessing it is a very expensive operation. Thus, we need to produce a separate output file for each algorithm with one scan of the raw data.

To make this idea concrete, we provide some queries examples we usually need to handle system logs. The simplest queries we want to run are simple projections and selections, such as *"get all logs from a specific machine A"*.

There are two kinds of more complicated queries.

- We sometimes need to process the join of two streams or a join from a stream with a static database table. For example,
 - *what is the average CPU usage 5 minutes before and after a machine that reported an error?*
 - *select all entries that the source IP addresses is assigned to UC Berkeley.*

The latter requires a join with a static database containing the information on IP address assignments.

- Using current statistics as filters. For example,
 - *select the abnormal event. An event is considered abnormal iff it has reported an error code and it reports a response time that is greater than the (average response time + 2* stddev) of the previous minutes.*

This example requires both intra entry processing and inter entry processing, since one node is not enough to process everything to calculate the average and standard deviation. So the processing looks like: the first tier calculates a sum, sum of the squares and count, sending these three to second tier, where the average is calculated and feedback to the first tier.

2.2 Traditional methods are not suitable for this application

Our experience shows that, traditional methods, such as ad hoc scripts or relational databases, are not able to meet all the requirements of system log processing discussed above.

2.2.1 Ad-hoc scripts are not enough

Most of the logs, if ever analyzed, are done by experienced system administrators by writing scripts in languages such as PERL or Python. Our experience shows that this approach can be time consuming, resource inefficient, and error-prone.

At the early stages of this project, we did not realize all of the practical problems discussed in the previous section. In order to get started experimenting SLT algorithms as soon as possible, we began writing Python scripts to process the data.

Python is a scripting language, which is very efficient (in terms of code length) in processing text files. The simplest preprocessing (scan through the data, project certain at-

tributes of each entry and output them as a text file) can be expressed in about 50 lines of code, and some of our original results were obtained with the data preprocessed in this manner.

However, as we were trying out more algorithms, we found ourselves frustrated by the following problems:

- **It takes a huge amount of time to scan through the data.** In our case, a single scan through one minute of log data takes more than 10-12 minutes. Most of the time is spent on reading the data from disk, uncompressing it, and parsing it.
- **It is hard to handle multiple queries with a single script and share intermediate results** Producing data for multiple algorithms in a single script makes the Python scripts significantly more complex. A couple of aggregation queries take about 150 lines of Python code. It became even more complex when we wanted to save some of the intermediate results to disk for future use.
- **It is hard to add/modify existing queries.** Adding one query to the code may require changing the code for existing queries, because we share the buffer and intermediate results among the queries.
- **Fine grained parallelism is much harder to achieve.** We observed that preprocessing is CPU bound (instead of I/O bound) on our cluster, so speedup can only be obtained by using multiple processors. Parallel processing of the data is very hard to implement in fine granularities (such as at single-request level).

In summary, our experience shows that even though ad-hoc scripts are enough for small data sets, they can be very difficult to maintain.

2.2.2 Problems with relational databases

We also considered using traditional relational databases for our data, since the preprocessing can be specified as SQL queries, reducing the complexity of preprocessing scripts. We rejected this approach for the following reasons:

- **System logs do not have a fixed schema.** System logs usually have no fixed schema. Efficient relational database operations require a well designed schema; both logical (tables) and physical (file organization, indexing and so on). Changing schema is assumed rare and expensive. However, the format of system logs change as the system evolves.
- **One-time-queries are not suitable for generating multiple data output.** The queries in a relational database are *one-time queries*; generating another result usually requires a separate query. If a scan is required on a terabyte data base, each of the queries will take a long time to run.
- **It is hard to support queries involving temporal properties of data.** However, this is essential in temporal data analysis.
- **The cost of using a relational database is high.** Importing multi-terabytes of data into a relational database would have brought to us very high initial cost (both I/O and CPU) and storage cost.

2.3 Stream model of system log data

Traditional methods of analyzing data does not work well in this case because the data model they are based on is not suitable for the characteristics of system log data.

A *data model* is a collection of high-level data description constructs that hides the underlying low-level storage details [33]. It helps people to understand the data better and

they can build proper data processing systems to manipulate the data.

Both ad hoc scripts and relational databases treat the data (system logs in this case), as persistent entities and runs queries against them over and over again. To answer a question, a query must be evaluated. This data model is not suitable in the case of system logs, which has the following characteristics:

- **Log data entries arrive on-line.** The data rate is determined by the source (the systems that generates the log), and the temporal rate variation can be large. The log processor not have any control over the order in which data elements arrive. In large scale distributed computer systems, logs are generated continuously on each of the machines with different data rate.
- **The system log is an infinitely long sequence of log entries, but the memory on the log processor is limited.** Once a block of log data is processed, it has to be either discarded or archived, which makes it hard, if not impossible, to find it again. In fact, old system events are much less valuable for failure detection in our application, so it is completely fine to discard those entries once the desired statistics have been obtained. Typically, companies archive raw system log data for a few weeks before discarding them, but the statistics and data representing interesting system events are preserved.
- **A time stamp is attached to log entry explicitly or implicitly.** (i.e., the arrival time at the stream processing system). Therefore, temporal properties of the log data can be easily obtained. Actually, temporal correlations of system events are very important in system failure detection [26, 41, 17].

Research in data models suggests that system logs should be modeled as data streams, instead of relations [2].

The idea of text streams are not new. For example, *grep* in UNIX is a program that processes stream queries specified as regular expressions. However, lacking of schema definition in the text stream results in ad-hoc and complex solutions to larger problems such as processing large system log files.

The benefits of using data stream over ad-hoc scripts and relational databases are the following:

- **Continuous queries.** The queries on data streams are usually *continuous queries*, in contrast to one-time-queries. One-time-queries run on a snapshot of the data, and return a single result to the user, while continuous queries are evaluated as data elements in the stream arrive. Here is an example of a continuous query:

– *Which machine has handled 5 times more requests than any other machine over the last 10 minutes?*

This query has to be re-evaluated each time a new system log entry arrives, and thus produces an output data stream containing the name of machines. The output stream can be used directly as an indicator of system failures or can be used as a regular data stream, for example, as an input to an SLT algorithm. Continuous queries can either be pre-defined or ad-hoc, and multiple queries can run on a single data stream concurrently. The ability to perform continuous queries has great advantage for preparing data for SLT algorithms.

- **Off-line SLT algorithms can be used too.** As described in Section 3.1, many commonly used SLT algorithms are off-line algorithms (e.g., our decision tree algorithm), which work on *chunks* of data instead of streams. It is trivial to accumulate preprocessed stream in a buffer to get a data chunk large enough for the off-line algorithm. It is better than traditional methods in that preprocessing the data *before* buffering

it allows us to save only the data we want, thus making the buffering much more efficient.

- **Easy-to-change schemas.** It is easy to change stream schema, since there is no data stored in database. This makes it easy to add new streams and modify the output desired.

2.4 Building a parallel system using TCQ as building block

We wanted to build an infrastructure to support data analysis research of system log data. A major concern is simplicity. It should be simple enough that the initial configuration should either be automatically generated or be specified with a high-level description. The interface between our architecture and the system monitored should allow easy deployment in production environment. This architecture should be flexible enough to accommodate many algorithms, both on-line and off-line, without significant re-configuration. It should also be easy to add or remove data streams and components.

The purpose of the system is also to make the algorithm implementation as easy as possible, so that SLT researchers can focus on the algorithm rather than on tedious job of accommodating various input formats of raw data.

We tried to make use of available software from other research projects. The major component we use is TCQ.

The use of TCQ helped us to easily specify and add/remove continuous queries, which solved the second and third problem discussed in Section 2.2.1.

Turn-around time is our next concern, or more specifically, the delay before one can start evaluating an SLT algorithm. Our software architecture is build on TCQ, which allows user to specify fine-grained parallel execution over a computer cluster to achieve short turn-around time and scalability. The main features include:

- All data flows in the system are modelled as data streams, which are easy to understand and manipulate. Design of the system is driven by the flow of data. The output of one stream processor can be used as input of another. Any stream can be buffered and used by an off-line algorithm. Modelling everything as a stream also makes it easier for people to understand and handle the data, since most people are familiar with UNIX-style filters. Note that since most complex buffering is done in TCQ instances, most of the filters can be implemented without keeping much state, making the implementation of filters easier.
- It is easy to buffer a stream of data for a certain period of time to support off-line algorithms that require chunks of data. Result-saving policy can be specified separately for each stream in order to deal with temporal variation of stream data rate, importance of different streams and storage constraints.
- It is also simple to cache/store any intermediate stream to disk and reuse it later. This is especially important for research purposes, as we are constrained by the hardware resources available to us.
- If users are unfamiliar with SQL, they can write filters in other languages such as Java.

Chapter 3

Related Work

This research involves multiple areas including statistical learning theory (SLT), data mining, dependable system design, data warehousing, data stream processing and control theory. In this chapter, we discuss related work in each of these areas.

3.1 Automated system problem detection

Automated system problem detection involves discovering problems from both outside the system (intrusions) and within the system itself. Both areas require fast processing of system logs.

There have been many techniques for intrusion detection. Hofmeyr *et al.* uses short sequences of system calls executed by processes as discriminator between normal and abnormal operations[19]. The normal and abnormal system call sequences are obtained both by static analysis and runtime logging. Singh *et al.* proposes an automated approach for detecting previously unknown worms and viruses [35]. This approach, called “content sitting”, is based on the fast detection of worm behavior that is different from normal traffic. The algorithm used is specific to this problem. The resulting system can run at network wire-speed. Instead of targeting at performance, we make the flexibility as our most important goal.

Coodbook [43] approach considers that each problem causes many *symptoms events*.

The set of events caused by a problem are treated as a “code” that identifies the problem and correlation is treated as “decoding” the set of observed symptoms. The codebook is an optimal subset of events that must be monitored. However, dealing with the “noises” in the codebook requires complicated algorithms.

More and more sophisticated SLT algorithms are being used for detecting and localizing system failures and software bugs in runtime. Most of them involve time-related information in the logs (i.e., a log entry is not processed by itself, but the correlation among a sequence of logs are used).

The execution paths, which involve a sequence of log entries, have proved to be important. Chen *et al.* uses decision trees for localization of failures on the eBay web site [9]. Each executed request reports attributes such as name, type, machine, version and status of the request. A decision tree is trained to predict the status attribute and the generated rules are used to localize what machine, type of request, or version of software is causing problems.

In Pinpoint [8, 21], Kiciman *et al.* instrumented the JBoss application server so that a J2EE application reports *execution paths* of all requests. The path is a list of J2EE components that the particular request used. Pinpoint can detect anomalous paths and correlate them to identify the failed components.

Vilalta *et al.* apply temporal data mining and time series analysis to predict critical events in computer system such as *high CPU utilization* or *imminent router failure* [40, 41, 34].

As more and more data are collected from the system, the correlations among them become not that trivial to analyze. Cohen *et al.* uses Tree-Augmented Bayes Nets for automated performance analysis [10]. They measure 124 types of performance metrics on a sample server and the induced model is used for prediction of Service Level Objective

violation.

The system can be instrumented to it writes arbitrary complex information into logs. Liblit proposes a sampling infrastructure for gathering information about execution of C programs [22]. He instruments the source code of the program at every branch, assignment and function call. The recorded information from runs of the crashed program is correlated to obtain the possible bugs. Getting these data and analyze them in larger computers systems will become very difficult due to the size of these logs.

3.2 System monitoring and management

There has been a lot of efforts on monitoring systems in both academia and industry. Simple Network Management Protocol (SNMP) [7] allows user to instrument and monitor aggregated performance of heterogeneous component in a network environment. It provides a visualized and hierarchical infrastructure to support high volume data collection and separating management boundaries.

There are commercial tools that allow user to monitor and do simple analysis on the data collected. The major tools include HP OpenView[11], IBM Tivoli[12] or Microsoft Operations Manager[13]. These tools allow user to navigate through the collected and stored data, and run statistical analysis on them. However, they are not designed for preparing data for statistical learning algorithms.

Traditionally, the collected data are sent to some centralized servers which may waste bandwidth. Both Astrolabe [39] and PIER [20] manage to collect and analyze the data on the node where they are generated. Astrolabe makes use of gossip protocol and the architecture is formed in a hierarchical structure of domains. PIER is implemented on a DHT [37]. Both allow user to run queries in SQL which are then evaluated in a distributed way in the system.

Bodik *et al.* show that by combining automated detection of system failures with visualizations of system status information (such as transitions from one page to another) to system operators can reduce system problem detection time [3].

System events are not only used to predict system errors but also used as the infrastructure of the network. Siena [6] is a content-based networking infrastructure, that uses a specific component called *publish/subscribe event-notification service* to allow the efficient dispatch of events among all the components in the system.

3.3 Stream data processing and mining

Our work is also related to the stream processing and data mining work in database community. Stream processing addressed the issue of dealing with data that arrive in multiple, continuous, rapid and time-varying data streams [2].

A number of stream processing systems have been proposed to handle continuous queries over the data stream. TCQ addressed this problem with eddy query processing framework that adapts the temporal variation of data streams in data rate and statistical characteristic of the data stream [28]. It also allows to share evaluation path among multiple queries.

Several new algorithms that are suitable for mining data streams are proposed. The characteristic of most of these algorithms is that they only look at every tuple in the stream once [17]. In contrast, for most of the SLT algorithms it is not enough to look at each data tuple just once. Buffering and caching of old data are supported in our work to solve this problem.

Stream processing is also used in sensor network data monitoring and analysis [27]. Though the data rate from sensor network can also be high, it is much less complex than logs generated by a large cluster of computers.

3.4 Applying control theory to computer systems

Control theory has a long history and has been successfully applied to many areas of engineering. However, the application of control theory to computer systems are quite a new topic. Hellerstein *et al.* provides a introduction to the basic techniques as well as an overview of recent research in the area of applying control theory in computer systems [18]. The general techniques of controller design and analysis in this report follows this book closely.

Control theory has been successfully applied to many applications in computer systems. It is usually used for adaptive configuration tuning. The controlled configuration parameter can be resource (e.g. CPU time) allocation [24], buffer size [16, 15], or some other configurable parameters such as max user allowed or HTTP keep alive [14].

The goal of controller can be regulation or optimization. For example, Liu *et al.* use control theory achieve the goal of meeting service level objectives (SLOs) on response time in a CPU that is shared by multiple processes [24]. The response time can also be minimized, and thus this is formulated as a optimization problem [14].

Sometimes it is not easy to find the function to optimize, especially when the requirements are complex. The function to be minimized can be a cost function, which is an artificial function that express the relationship between violating the objective and the cost of control action [16].

Unlike many physical systems, there are few first principle models for software systems, due to their complexity. However, queueing theory often provides alternative ways to design a controller if desired metrics are not easy to obtain [16, 14].

Chapter 4

A Flexible Architecture for Processing System Logs

4.1 Key component, Telegraph Continuous Query Processor

General stream processing techniques have been studied in database community in great depth [1, 17]. A number of general purpose stream processing systems have been built [2, 28]. We use TCQ.

The queries are specified in PostgreSQL SQL, with all data types and functions. One query is usually specified by a few lines of SQL and all query plans are automatically optimized. This makes adding and modifying queries significantly easier than ad-hoc scripts.

As the characteristics of the data stream change, the query execution changes adaptively. For example, during a system failure the average delay may suddenly go very high and thus a selection condition “delay > 10 seconds”, which normally throws away almost all tuples, suddenly becomes not selective. Without adaptive query execution, the query evaluation may become very inefficient for other operators in the query plan.

TCQ supports running multiple queries on a single data stream and generating multiple outputs concurrently. This best fits our case of running different SLT algorithms requiring different input data on a single log file. The computation and storage are shared aggressively, so running more queries on the same stream does not increase workload significantly.

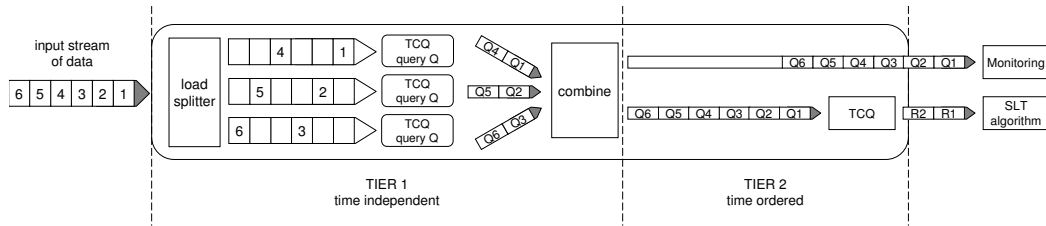


Figure 4.1: A general structure of our parallel system. We used TCQ as our major building block, and other components are written in Java. We use a load balancer to split the the stream to two TCQ instances. The first tier of TCQ nodes performs queries that are independent of time (i.e., queries that do not need a time window). The output streams are reconstruct in time order at the combiner. After the streams are combined, the second tier of TCQ instances performs time dependent queries to generate the final output, which can be used in monitoring programs or as input to statistical learning algorithms for automatic problem detection

TCQ is still in its early research stage [36]. The released version is functional, but not optimized for performance. A single node running TCQ processes data at a maximum throughput of about 1,800 tuples per second (see Chapter 5 for the experiment in which we obtained this result).

In order to make the system scale to the data rate a large distributed system could generate, we used multiple instances of TCQ running in parallel. We are interacting with the TCQ research team to investigate higher performance and new features.

4.2 Description of the parallel architecture

Modularity is an important goal of our design. We designed the system so that it consists of simple building blocks (see Figure 4.1) that communicate with each other using sockets. They can be deployed on a single physical node or over multiple nodes in a cluster. These components were written in Java and comprise about 750 lines of code.

4.2.1 Two-tier parallel architecture

We use a load balancer to split the the stream to two TCQ instances. The first tier of TCQ nodes performs queries that are independent of time (i.e., queries that do not need a time

window). The output streams are reconstruct in time order at the combiner.

After the streams are combined, the second tier of TCQ instances performs time dependent queries to generate the final output, which can be used in monitoring programs or as input to statistical learning algorithms for automatic problem detection

4.2.2 Building blocks

The system is composed of five building blocks

- **Data source** is the interface for getting the various kinds of data, translating them into data streams and feed them into the stream processing system. It provides a small interface to the production system, and can be overridden to use multiple types of data, such as logs stored on disk, network monitoring readings, or live stream of system event reports. We used a load simulator in our experiments, the design of which is discussed in Chapter A.
- **Load splitter/balancer** is a small component used for load balancing that takes a single input stream, divides it into multiple streams and redirects them on to multiple nodes. When the data rate cannot be handled by a single TCQ instance, we create multiple instances and use the load splitter to route the stream to all the instances. The data processing within the load splitter should be simple and fast, since it is on the critical path of the system and always sees a large data rate. However, the load balancing must tolerate the disturbances in the system, so our initial round-robin load balancer has serious problem when one of the node gets slow down. The detail of this problem and our solution using control theory is discussed in Chapter 6.
- **Stream combiner** is a component used to combine multiple streams generated by load splitter to re-create the original order of entries. It works on the original time-stamp attached to each entry in the stream. If the time-stamp is too coarse grained

to order the entries (for example, there are 600 events in a single second and some of them have causal dependency), we attach a unique sequence number to each entry when it is pushed into the system.

The stream combiner acts as a barrier in common parallel computation systems. This requires the load balanced on each component. We discuss this this problem in Chapter 6.

- **TCQ instances**, as described in Section 4.1, are the key components in the system. They take in multiple SQL queries, multiple data streams and output the results of the queries as data streams. The output data streams can be buffered for off-line algorithms or written to files for future use. The raw stream can be configured to be archived. The TCQ instances also output its own performance statistics as data streams (which is called introspective query) to centralized controllers.
- **Applications** are defined as a components taking data streams or a file as input and output another data stream. The data stream can be interpreted by a GUI component for human administrators to review or achieved for future reference. Publicly available algorithms can be plugged into the system with only a minor wrapper for reading data streams (for example, through JDBC or simply through a UNIX pipe).

Note that the data streams between each component are not necessary in the same format. They can be implemented as text streams, but we can also use binary streams with type definition which saves parse time. Changing the format of a stream is simple; it only requires to change the output format of the sender and input format of the receiver or add a separate wrapper around the receiver.

All the components can easily be changed, even while the system is running, with another component with the same interface. This is useful. For example, we substitute the

simple round-robin load balancer for one with feedback control in less than one minute time of modifying the system configurations. This is also useful if the system operators are not familiar with SQL and want to replace the TCQ nodes with his/her ad-hoc scripts.

4.2.3 Implementation

We provided a simple way to build the system from components with simple object oriented specifications. All the components are modeled as a class. To build the system, one only needs to specify the parameters of the components or override some of the functionalities and the interconnections among them. A program we provide automatically generates a shell script that starts the components on multiple machines in the cluster. There are separate scripts for adding and removing streams and queries from each.

There are two steps to add a new SLT algorithm. First, the algorithm "subscribes" a stream from the system, which is done by specifying a new query. Second, the user may need to write a simple wrapper to generate the correct format and/or add a header.

The design and implementation of this architecture made use of many ROC principles of distributed system design. For example, making system states measurable, separating permutate states with temporal states etc. All the log data generated by this system have a well-typed stream schema so that the logs can easily be manipulated by other monitoring and control systems.

Part II

Improving System Reliability and Performance with Control Theory

During the experiments with our parallel architecture, we found some problems with the reliability of the system. For example, we need to enhance the flow control to handle random disturbances. We solved this problem using feedback control theory.

In this part, we mainly discuss three applications of control theory: solving a flow control problem of TCQ without breaking the black-box abstraction (Chapter 5); implementing a self-adaptive load balancer that achieves optimal response time and throughput even under random disturbances (Chapter 6); implementing a load simulator that provides exactly the specified amount of tuples to the system being tested (Chapter A in appendix). All three applications used linear, first-order model for target system, and simple proportional or proportional-integral controller, with are very simple control theory.

For readers who are not familiar with control theory, we provide detailed introduction to components of a feedback control system, namely, *system identification* (i.e., estimate the parameter for system model), and *controller design* with the discussion of fixing TCQ flow control problem. We also discuss the differences and tradeoffs between proportional (P) controller and proportional-integral (PI) controller, together with the design and assessment of the workload simulator in Chapter A in appendix.

In Chapter 7, we summarize of experience of applying control theory to computer systems, including the difficulties we found and our solution, the advantages of applying control theory to traditional ad-hoc methods, and the limitations of classical control theory.

Chapter 5

TCQ Flow control

In order to make the log stream processing system scalable, we want to run TCQ in multiple tiers, and in parallel within each tier, in order to handle the huge data rate of event logs.

Figure 4.1 shows that we connect multiple TCQ instances. In this part of the report, we mainly focus on the first tier where the original stream is sampled and sanitized, because it absorbs most of the load. A load balancer decides which TCQ node an event log entry is sent to, and a timestamp is attached to every log entry so that time order can be reconstructed in the combiner.

We begin by describing the flow control problem we found in TCQ, and its impact on our application. We then discuss how to solve this problem as a control problem and show the high level design of the control system. We present the technical details of system identification and controller design in Section 5.3 and the evaluation of the system in Section 5.4.

5.1 Flow control issue in TCQ

There are some problems with TCQ flow control that cause data loss in some cases. In order to understand this issue, we first take a closer look at the internal structure of TCQ.

TCQ is structured as a multi-process system. Every data tuple goes through the following sequence of processing:

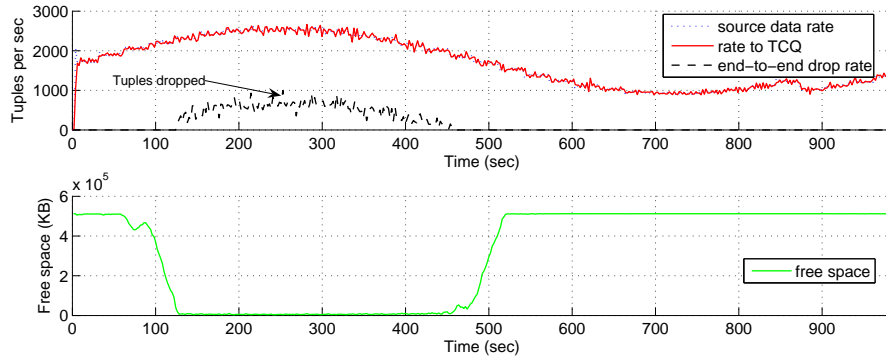


Figure 5.1: Behavior of TCQ node without regulating result queue length. The top plot shows the source data rate, the data rate enters TCQ backend, and the data being dropped per second. The bottom plot shows the size of free space on result queue. We can see that tuples are dropped even if overloading is only transient.

1. Wrapper cleaning house, which parses input data and translates them into the internal data structures.
2. TCQ backend, where the tuple is processed by multiple relational operators, such as projection, selection, join and aggregation.
3. The resulting data are sent to a result queue, where they are fetched by a frontend process[28, 36].

Unfortunately, when the backend processes data faster than the frontend process, Figure 5.1 shows the result queue fills up and results are dropped.

Having a full queue is a serious problem in our application. If the tuple can get dropped randomly after sampling, there is no way to guarantee the statistical distribution of the output, which is required by most of our automated failure detection applications. What is worse, it makes load balancing hard, since one can never tell in the beginning whether the next log entry will cause the TCQ node to be overloaded until the tuple gets processed.

This problem is hard to fix internally, too. A naive idea is to let the process block when result queue is full, i.e. apply back pressure on the on the earlier stage of the processing. However, since all operators are shared and connected as multiple evaluation trees, blocking

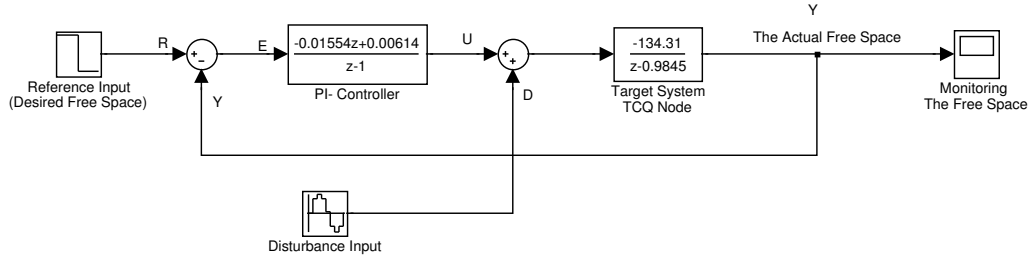


Figure 5.2: PI Controller for regulating the free space in the queue. There are two inputs to the whole system. The reference input R is the desired free space in the TCQ result queue and disturbance input D models the unpredictable load to the system. The output Y is measured free space. The two transfer functions models the controller and target system, respectively. The P part and I part of controller are combined in a single transfer function.

on one operator will cause unpredictable effects on other queries running in the same node.

There are also two uncertainties that make statically regulating the result queue length infeasible:

1. Transient disturbances in the system cause throughput changes;
2. The percentage of the input will get to the result queue (a.k.a. *selectivity* of the query) is unknown in advance.

Thus, we need some dynamic and self-adaptive way to regulate the result queue length.

5.2 Control problem formulation

We found that this problem can be formed as the following *control problem*: We need to prevent the free space on the result queue from getting to zero, and maximize the utilization of TCQ node, by controlling the input data rate. We also need to tolerate disturbances such as a slow node or change of selectivity.

We construct the following model of the system, shown in Figure 5.2. In this case, the *target system* is the TCQ node. The *transfer function* of the target system is:

$$y(k + 1) = au(k) + by(k) \tag{5.1}$$

In equation 5.1, y is called the *measured output*. In this case, it is the free space of result queue, measured in KB. This can be obtained by reading TCQ logs. The *control input*, u , is the data rate pushed into the TCQ system (measured in *tuples/sec*). Parameters a and b relate the current output and input to the *predicted* output of the next time periods.

Parameters a and b are obtained by a process called *system identification*. We need to conduct experiments to collect data for $u(k)$ and the corresponding $y(k + 1)$ and estimate a and b with *least squares regression*, a well known procedure that can be found in many textbooks and software packages.

System identification is not always easy in practice. First, we need to choose an operation point of the target system, i.e., the typical workload we want TCQ to handle. Then we need to find the *linear operation range*, which is the input range in which the output has an linear relationship with the input. Linear models can not only make controller design easier but also helps to prevent overfitting the model and thus better tolerate uncertainties of the system. We also need to provide a carefully designed workload, which causes the result queue to fill up / empty at different rates, but not too fast for us to observe. The tricky process of designing a proper workload for system identification is discussed next in Section 5.3.

From the process of system identification is presented in Section 5.3, we measured on our experiment platform that $a = 0.985$ and $b = -134$. Note that a is a positive small number and b is negative. a is the coefficient of $u(k)$, the last period input tuple rate, it is positive, which means that increasing the workload will cause an increase in queue length. A large negative b is the coefficient of y the last period queue length, and b shows how fast the queue empties. This makes sense intuitively.

Having obtained the model for target system, we consider the controller design. The job of the controller is, by looking at the *control error* e and other current and previous states of

the system, decides the next control input u . We calculate e by $e(k) = r(k) - y(k)$, where r is the *reference input*. In our system, r is the size of free space we want to maintain in the queue. Setting the desired free space smaller than the max size can ensure that the system is operating at its max capacity, because the input data rate equals the output data rate.

We used Proportional-Integral (PI) controllers, a widely used controller in control theory, because it tolerates the disturbances well. The *control law* for PI controller is:

$$u(k) = u(k - 1) + (K_P + K_I)e(k) - K_P e(k - 1) \quad (5.2)$$

where K_P and K_I are controller parameters we need to design. It can be proved that PI-controller will always drive the steady state error to zero, despite the disturbances.

The parameters K_p and K_I , are chosen with a technique known as *pole placement*, which allows us to predict the stability, time required to converge, and maximum overshoot before implementing the system. The detailed calculation of pole placement is shown in Section 5.3.

Figure 5.2 shows the complete feedback control block diagram. Note that in this diagram, the transfer functions and signals are represented in Z-domain, which is similar to S-Domain in Laplace Transformation for analyzing continuous signals, but used in analyzing discrete signals.

5.3 System Identification and Controller Design

In this section, we provide some technical details on system identification and controller design. We also provide a short tutorial for readers who are not familiar with control theory. However, this section can be skipped without affecting further reading.

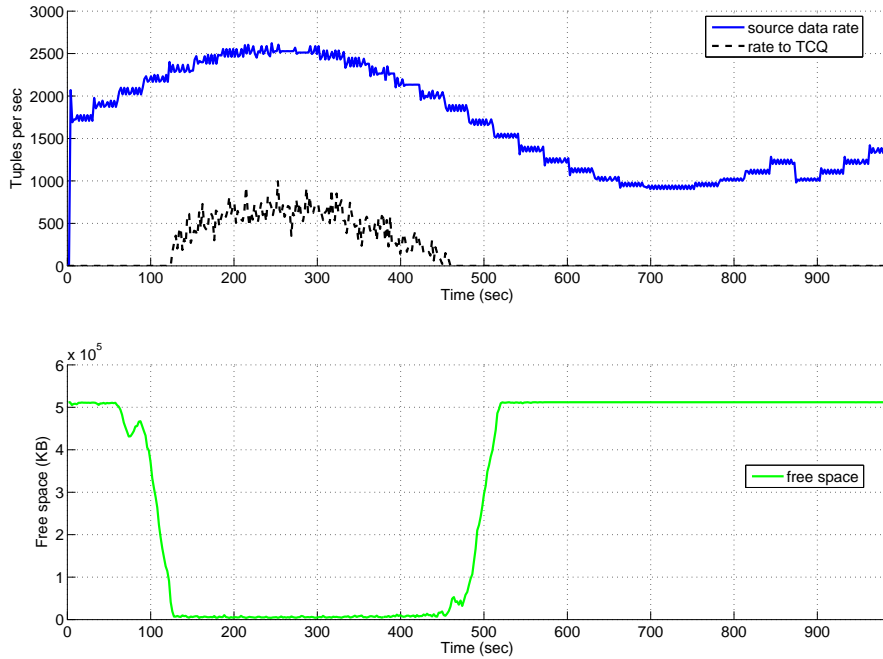


Figure 5.3: Not carefully chosen workload cause the queue length to fill up. If the work load for system identification is not well chosen, the queue fills up shortly, even if the operation point and linear operation range is correctly chosen. Comparing to Figure 5.4, this will result in a bad model for target system.

5.3.1 System Identification

In order to estimate parameters a and b in Eq. 5.1, we first conduct experiment on an off-the-shelf version of TCQ. We want to measure the change of y at different data rate u .

We vary the input data rate to make the result queue fill up or empty. The design of this workload is very tricky, because the queue length is an integral of the input data rate. So if we keep a high data rate for a relatively long time, the queue fills up (see Figure 5.3). This will cause a lot of data points to cluster at a corner (see the left of Figure 5.5). In order to solve this problem, we need to provide a carefully designed workload, which allows the system to fill up / release the queue at different rates, but not filling up /releasing too fast for us to observe. Figure 5.4 shows the workload we used.

Running the experiment long enough to get N data points, the data obtained are two

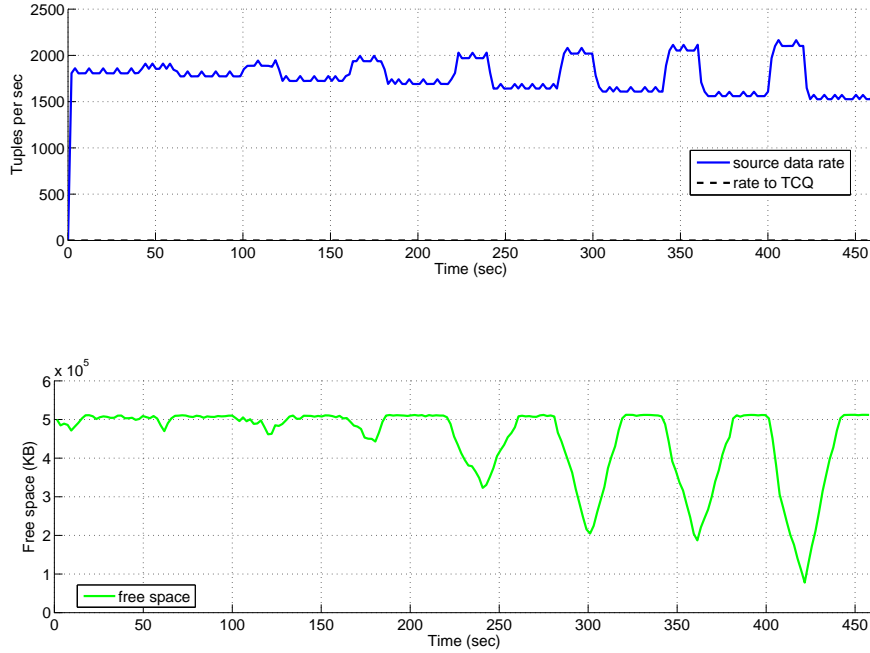


Figure 5.4: workload used in system identification. We set the operation point to be 1,850 tuples/sec, which is a little less than the max throughput of TCQ with settings described in Section 5.4. Then we let the datarate fluctuate around this value in order to make the queue fill up/empty. In the bottom figure, we can see that the free space in queue changes with the change of the input data rate, but never goes to zero.

sequences of tuples $\{\tilde{u}(k), \tilde{y}(k)\}$, $1 \leq k \leq N + 1$. We need to normalize the input and output around their operating points to do the regression. Let \bar{u} be the mean input value, and \bar{y} be the mean output value. We choose (\bar{u}, \bar{y}) to be the operation point. We calculate the offset values $u(k)$ and $y(k)$ as:

$$u(k) = \tilde{u}(k) - \bar{u} \quad (5.3)$$

$$y(k) = \tilde{y}(k) - \bar{y} \quad (5.4)$$

Equation 5.1 is a model to predict $y(k + 1)$ from $y(k)$ and $u(k)$. We denote the predicted value of $y(k + 1)$ as $\hat{y}(k)$.

$$\hat{y}(k + 1) = ay(k) + bu(k) \quad (5.5)$$

Our goal is to find the a and b that makes the model “accurate”. We evaluate the accu-

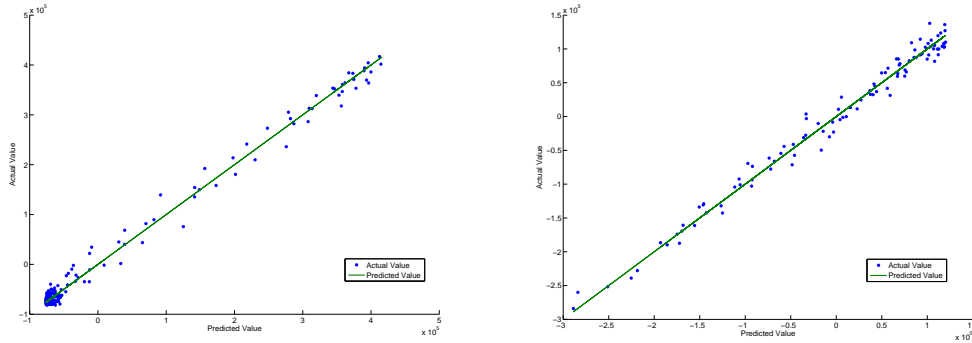


Figure 5.5: Models Constructed with good and bad workloads. The solid red line is the model (a linear function), and the blue points are the actual data point. The left figure shows the model constructed using a workload in Figure 5.3 (the bad one). Many data points are clustered close to a corner. However, with carefully planned workload (Figure 5.4), the actual data points are spread out, which results in a much better model.

racy of the model using is sum of the squared errors, reducing the problem to minimizing the following function:

$$J(a, b) = \sum_{k=1}^N [y(k+1) - \hat{y}(k+1)]^2 = \sum_{k=1}^N [y(k+1) - ay(k) - bu(k)]^2 \quad (5.6)$$

where $N + 1$ is the total number of observations.

This minimum value can be found either by taking derivatives of Equation 5.6 and setting to zero, or with standard optimization routines such as *least squares regression*. For example, this operation is done with `mldivide` function in Matlab. Figure 5.6 shows the code segment that does this regression.

In our experiments, 105 data points are collected and the estimated a and b are $a = 0.985$ and $b = -134$, respectively.

After obtaining a and b , we plot predicted values $\hat{y}(k)$ in the same figure with the actual data points $y(k)$ (see Figure 5.5 (right)). We see that the actual data points are in the whole range and evenly distributed on both sides of the predicted value. This tells us that this prediction works well. We can also use quantitative parameters, such as *variability* to evaluate the model. The variability, R^2 , is defined as:

```

function [a,b,y,u]=paramesti(up,yp)
    u_mean = mean(up(1:end-1))
    y_mean = mean(yp(2:end))
    u = up - u_mean;
    y = yp - y_mean;

    H=[y(1:end-1) u(1:end-1)];
    theta=(H\y(2:end))';
    a=theta(1);
    b=theta(2);

```

Figure 5.6: Matlab routine that does the parameter estimation.

$$R^2 = 1 - \frac{\text{var}(y - \hat{y})}{\text{var}(y)} \quad (5.7)$$

Usually we require R^2 to be larger than 0.8. In this case, $R^2 = 0.98$, which is very good.

5.3.2 Controller Design

Controller design is the mathematical process used to choose the values for the parameters (K_P and K_I in this case) of the controller.

We focus on four properties of the controller, stability, accuracy, settling time and maximum overshoot.

In order to analyze the system and avoid working directly on difference equations like Equation 5.1, we need a better representation of the transfer function that allows us to deal with delay easier. In discrete control theory, we often transform the difference equation that is represented in time domain into a representation in z -domain. We can consider z as a time shifting operator. Converting a time domain representation to z -domain is very mechanical [18]. For example, the z -domain form of Equation 5.1 is:

$$G(z) = \frac{Y(z)}{U(z)} = \frac{b}{1 - a} \quad (5.8)$$

Also, the z -domain form of the control law for PI controller (Equation 5.2), is:

$$K(z) = \frac{U(z)}{E(z)} = K_P + \frac{K_I z}{z - 1} \quad (5.9)$$

Z-domain representation of the transfer functions makes it easy to calculate the combined transfer function of large, complicated systems. We can calculate the transfer function from reference input R to measured output Y using the fact that:

$$F_R(z) = \frac{F_{FF}(z)}{1 + F_{LP}(z)} \quad (5.10)$$

Where $F_R(z)$ is the transfer function from input R to output Y , and F_{FF} is the feed-forward transfer function from R to Y (i.e., the transfer function as if the feedback loop does not exist), and F_{LP} is the loop transfer function.

Using Equation 5.10, we can easily obtain the transfer function $F_R(z)$ for system in Figure 5.2:

$$F_R(z) = \frac{((K_P + K_I)z - K_P)G(z)}{z - 1 + (K_P + K_I)z - K_P)G(z)} \quad (5.11)$$

From Equation 5.11, we can analyze the four properties described above. First we can prove that this system will always get a steady state error of zero, regardless what the disturbance input is [18]. That is to say, it is accurate at steady state. This is also the reason why we use PI controller in this case.

Then we need to choose the parameters K_P and K_I in order to achieve the following goal: 1) the system is stable (i.e., for every bounded-sized input, we have a bounded-sized output, aka, the BIBO property). 2) the settling time (i.e., the time required for the system to reach steady state) does not exceed k_s^* . 3) maximum overshoot does not exceed M_P^* .

All three important properties depends mainly on the *poles* of Equation 5.11. Poles are the values of z that make the denominator of the transfer function to be zero. The

denominator of Equation 5.11 is often called the *characteristic polynomial*. The system is stable if and only if all its poles lay in the unit circle on the complex plane.

Obviously, there are two poles in Equation 5.11, since the characteristic polynomial is quadratic. Let the two solutions be $re^{\pm j\theta}$. Control theory tells us that the settling time $k_s < -4/\log r$, Thus, an upper bound for r is:

$$r = e^{\frac{-4}{k_s^*}} \quad (5.12)$$

Also, we know from control theory analysis that the overshoot is mainly related to θ , $M_P \approx r^{\pi/\theta}$, so we have:

$$\theta = \pi \frac{\log r}{\log M_P^*} \quad (5.13)$$

With θ and r obtained, we can construct the characteristic polynomial as:

$$(z - re^{j\theta})(z - re^{-j\theta}) = z^2 - 2r \cos \theta z + r^2 \quad (5.14)$$

By equalizing the coefficients for each power of z in Equation 5.14 to the coefficients (involving K_P and K_I) of Equation 5.11, we can solve for K_P and K_I .

In this case, we have $a = 0.985$ and $b = -134$, from the calculation from system identification described in last section. $F_R(z)$ is therefore:

$$F_R(z) = \frac{b(K_P + K_I)z - bK_P}{z^2 + [b(K_P + K_I) - 1 - a]z + a - bK_P} \quad (5.15)$$

$$= \frac{-134(K_P + K_I)z + 134K_P}{z^2 + [-134(K_P + K_I) - 1.985]z + 134.985K_P} \quad (5.16)$$

Assume that we want $k_s^* = 5$ (five sample times), and $M_P^* = 0.2$. Using Equation 5.12 and Equation 5.13, we obtain:

$$r = 0.4$$

$$\theta = 1.7$$

Using Equation 5.14, we have the characteristic polynomial of this system to be:

$$z^2 + 0.1031z + 0.160 \quad (5.17)$$

Equalizing the coefficients of each power of z , we get

$$-134(K_P + K_I) - 1.985 = 0.1031 \quad (5.18)$$

$$134.985K_P = 0.160 \quad (5.19)$$

Solve the equations above, we find :

$$K_P = -0.0061 \quad (5.20)$$

$$K_I = -0.0094 \quad (5.21)$$

Therefore, the transfer function of the close-loop system in Figure 5.2 is:

$$F_R(z) = \frac{2.088z - 0.8245}{z^2 + 0.1031z + 0.16} \quad (5.22)$$

In order to verify that this system is stable, we calculate the magnitude of the largest pole of Eq. 5.22. This is done by setting the denominator to be zero. We can obtain the poles: $-0.0515 \pm 0.3967i$. The magnitude of the largest pole is 0.4. Thus the system is stable.

This process can be automated with Matlab. Figure5.7 shows the Matlab routine that does the calculation.

5.4 Assessment

In order to evaluate the effects of the controller, we implemented the controller and other components in Figure 4.1. We used unmodified version of TCQ, with default result queue space set to 512MB.

```

function [KP, KI] = PI_calculate(Ks, Mp, a, b)
    r = floor(exp(-4/Ks) *10) /10
    theta = floor( pi* (log(r))/log(Mp)*10) /10

    char_poly = [1 -2*r*cos(theta) r*r]

    A = [ b b; -b 0]
    Y = [ char_poly(2)+a+1; char_poly(3)-a]

    linsolve(A, Y)

    KP =ans(1)
    KI =ans(2)

```

Figure 5.7: Matlab routine for calculating K_P and K_I for PI controller. It takes in four parameters. Parameters a, b are parameters for target system, obtained from system identification. Parameters k_s and M_P are desired settling time and max overshoot. If k_s and M_P are not possible, the routine will output no solution.

The controller is implemented as a input buffer. It reads the TCQ log to figure out the current queue length every 2 seconds, and uses equation 5.2 to calculate the number of tuples to send to TCQ in the next 2 second. The excessive load remains in the input buffer.

We set the reference input r to be 400MB, which means we always want to keep 400MB empty space on the result queue. This turns out to be very conservative. In fact, as we can see in our experiments, this size can be much smaller with our controller in place.

We can see in Figure 5.8 that output buffer set its output data rate at about 2000 tuples per second and the queue length is stable at around 400MB, as expected.

In Figure 5.8, the free space drops at 100 seconds. This is the result of start up effect (that is, the TCQ system needs to warm up and is thus slow). The controller stabilized it in a very short time. From 50sec to 500 sec, the source data rate is actually higher than the data source can handle, the extra tuples are queued in the input buffer.

In order to test the tolerance of disturbances to the system, we started a CPU intensive process *cpu.hog* on the TCQ node at time 180 seconds. The controller, almost at the same

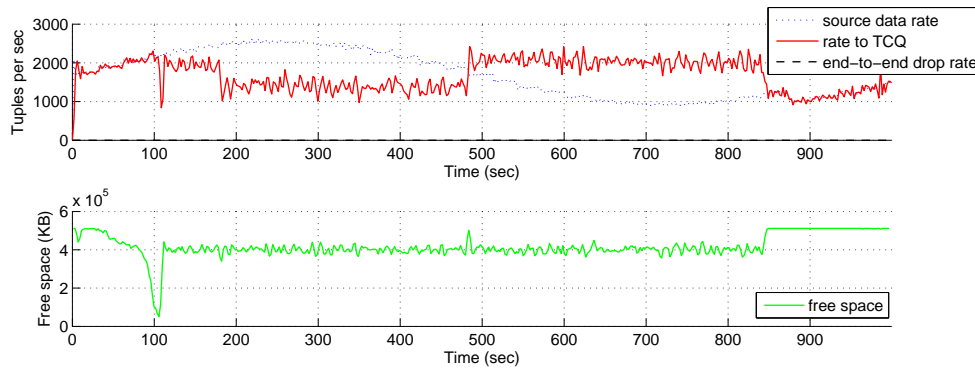


Figure 5.8: TCQ node with result queue length controller, under CPU contention. The top plot shows the source data rate, the data rate entering the TCQ backend and the drop rate (always 0 in this case). The bottom plot shows the free space on the result queue. At time 180, we started a CPU intensive process on the TCQ node. The controller automatically lowered the data rate and kept the free space on the result queue to be constant.

time when *cpu_hog* is started, reduced the output data rate to around 1,500 tuples per second, keeping the free space on the result queue unchanged. At time 480 seconds, we killed the CPU intensive process. We can see that the controller increases the output data rate back to normal level and the queue length still stays at the desired level.

This property is very useful for at least the following two reasons: 1) it automatically finds the operation point for maximum throughput even under unexpected disturbances; 2) it keeps the correctness of the system, in terms of not dropping tuples, allowing time for diagnosis of system failures, especially when an automated failure detector is used.

Like all other computer systems, this system has limited linear operation range. Starting from time 750 seconds, the input data rate is not big enough to keep the result queue size at desired level; the input data rate is smaller than the output data rate, and the buffer becomes empty. We used a quick-and-dirty fix to let the controller give up when the data rate is very low. We believe using an adaptive controller will be better theoretically, but this fix works fine here.

Chapter 6

Building the load balancer with control theory

Regulating input data rate to TCQ helps us avoid tuple dropping and automatically find the TCQ maximum throughput. However, it is not enough by itself to make the system scale. As Figure 4.1 shows, we want to run multiple instances of TCQ in parallel and reconstruct the time order of input events at the combiner. This creates a load balancing problem.

Like all parallel computation systems that involves synchronization, load balancing is important to minimize delays. A load balancer based on control theory works well in many cases [15, 16]. However, in this case, the control output is not directly measurable. We believe that it is a common issue when using black-box building blocks. We show our ways to address this issue in this section.

6.1 Effect of imbalances

Load balancing is particularly important in parallel computation system involving synchronization, such as a barrier. This is because the fast node, even if got work done faster, needs to wait for the slow nodes. Specifically, in our system, the combiner is a barrier coordinator. Then each event log entry is processed in parallel. The combiner reads output from each TCQ node, and performs a merge sort on timestamp. If one of the input is slower than the other, the combiner waits for the slow input and queue up others. The response time, in this

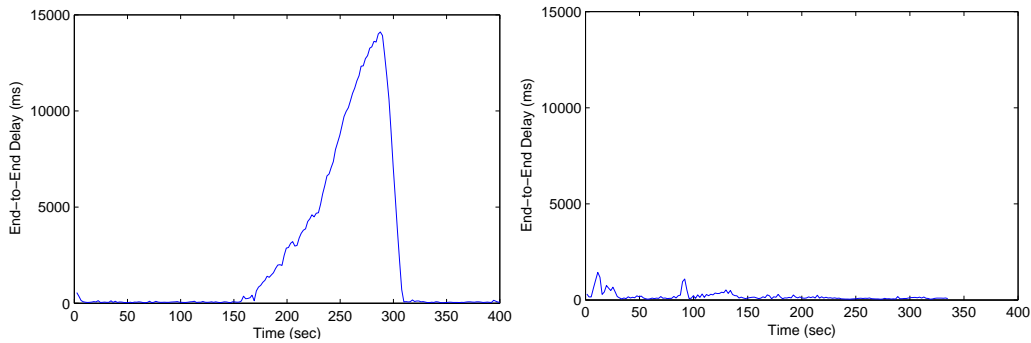


Figure 6.1: The Effect of disturbances with and without control. **LEFT:** Average tuple response time using a simple round-robin load balancer. At time 150 seconds, we started *cpu_hog* process on one of the TCQ nodes, the response time quickly shoot up to an unacceptable 15 seconds. At time 300 seconds, we killed the *cpu_hog* processes, the response time go back to normal. **RIGHT:** Response time when a load balancer with controller is used. At time 80 seconds, we started the *cpu_hog* process. The response time shoots up a little, due to controller delay and over shoot, but it quickly become stable again. At time 300 seconds, the *cpu_hog* is killed and the response time came back to normal.

case, depend on the slowest node.

Figure 6.1 (left) shows this effect in our experiment with a simple round-robin load balancer that sends equal amount of tuples to each node. This experiment is done with a constant data rate of 1000 tuples/seconds, smaller than the throughput of only one normal TCQ node. It runs fine when there are no disturbance. However, when the disturbance process introduced, *cpu_hog*, started on one of the TCQ node at time 150 seconds, even though the other node is completely normal, and by itself can handle the total load, the overall delay still shoots up to an unacceptable 15 seconds. This is because the round-robin load balancer always evenly divides the load onto both nodes, the combiner has to wait for the slower one, so the normal node does not get used.

Usually, load balancing system uses input queue length as the indicator of system load. However, this information cannot always be obtained easily from off-the-shelf systems. This is common, especially for complex systems, which has many internal queues and not designed to reveal its internal information. Unfortunately, TCQ is a system with unmeasurable queue lengths.

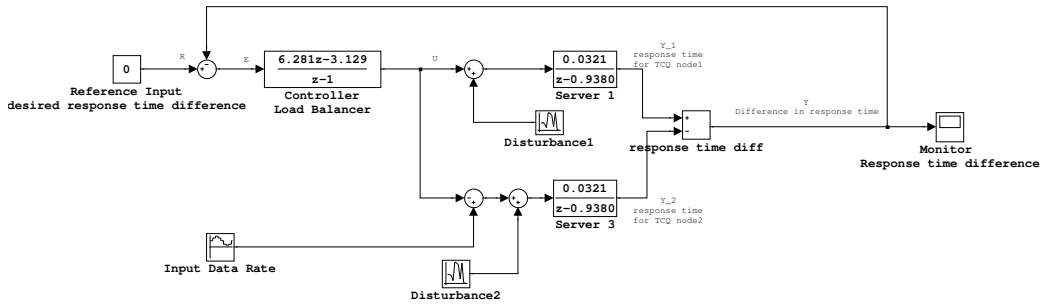


Figure 6.2: The block diagram of controller in load balancer. Reference input R is the desired difference in response time, which is always 0. The real input data rate is modelled as disturbance input since we have no way to control it. Other two disturbance input models the random disturbance. Target system is TCQ nodes (with the result queue length controller described in Chapter 5. The transfer function of target system models response time change due to input data rate change. The controller is still PI-controller. Error input is calculated from difference in response time, and control output is the data rate needs to be sent to TCQ node 1. Whatever data rate left are send to TCQ node 2.

We cope this issue by directly using average response time for each tuple time of the TCQ nodes. Note that in these experiments, we assume that the each data tuple cause about the same amount of work to the system, i.e. average is a metric good enough to let us obtain a model. We treat how to eliminate this limitation as our future work. We get the best end-to-end response time when the response time at each node is equal, because in this case, we don't need to wait for the slower node. We discuss how control theory is used to construct this load balancer in Section 6.2.

6.2 Control problem formulation

Figure 6.2 shows the control diagram. The controller monitors the different in response time for TCQ node 1 and 2, and uses the difference to calculate data input rate at node 1. The input data rate to node 2 is whatever data rate left in the total input. The system has four inputs. The reference input R is desired response time, which should be 0 in ideal case. Note that the total input is modelled as disturbance input, which reflect the fact that this data rate is determined by the data source, which cannot be controlled. The remaining two disturbance input models the random disturbances (e.g. selectivity change, CPU contention

etc.) as described in previous section.

The target system are two TCQ nodes. The transfer function is obtained by experiments and least square regression, exactly the same process as described in Section 5.3. In this case, both nodes have identical transfer functions since we run them on identical nodes, but it can be easily extend to a system with heterogeneous nodes. The controller is also PI controller and parameters are chosen by pole placement.

6.3 System Identification and Controller Design

In this section, we provide technical details on system identification and controller design.

We omitted some calculations that are very similar to those in Section 5.3.

6.3.1 System Identification

The system identification process is exactly the same as what we did in Chapter 5. The only difference is that we used TCQ with the input data rate controller we implemented in the previous chapter. We found that with the input data rate controller, the system identification process is actually easier, since we do not need to worry about the queue filling up.

In the experiment, we vary the input data rate and measure the response time of each tuple, taking average over a period of 2 seconds (the global sample time used in the entire system). The response time is obtained by comparing the timestamp from the input buffer into each TCQ node with the time measured when the tuple get finished processing by TCQ. Since both times are measured on the same node, there is no problem for clock drifting.

Using the procedure shown in Figure 5.6, we obtain

$$a = 0.9380 \quad (6.1)$$

$$b = 0.0321 \quad (6.2)$$

and thus the Z-domain transfer functions are

$$G(z) = \frac{0.0321}{z - 0.9380} \quad (6.3)$$

as Figure 6.2 shows.

6.3.2 Controller Design

Since in this case, we still need to tolerate disturbances that not controllable, so we only consider PI controller. A detailed comparison of P controller with PI controller is provided in Section A.3.

Comparing to the control system of the queue length regulator in Chapter 5, the system shown in Figure 6.2 is more complicated in that it has two target system blocks. The difference of the measured outputs of these two target systems are used as feedback to the controller.

However, we can use very similar method to obtain the close-loop transfer function $F_R(z)$. The only difference is that the algebra is a little more complicated. We briefly sketch the steps of calculating $F_R(z)$ here.

$$F_R(z) = \frac{Y(z)}{R(z)} = \frac{Y_1(z) - Y_2(z)}{R(z)} \quad (6.4)$$

$$= \frac{U(z)G_1(z) - (L(z) - U(z))G_2(z)}{R(z)} \quad (6.5)$$

$$= \frac{K(z)G_1(z) - L(z)G_2(z) + K(z)G_2(z)}{1 + K(z)G_1(z) + K(z)G_2(z)} \quad (6.6)$$

From the control law of PI controller in Equation 5.2

$$K(z) = \frac{(K_P + K_I)z - K_P}{z - 1} \quad (6.7)$$

and the transfer function of the target system

$$G_1(z) = G_2(z) = \frac{0.0321}{z - 0.9380} \quad (6.8)$$

substitute the two equations above into Equation 6.6, we get

$$F_R(z) = \frac{\frac{(K_P+K_I)z-K_P}{z-1} \frac{0.0321}{z-0.9380} - L(z) \frac{0.0321}{z-0.9380} + \frac{(K_P+K_I)z-K_P}{z-1} \frac{0.0321}{z-0.9380}}{1 + \frac{(K_P+K_I)z-K_P}{z-1} \frac{0.0321}{z-0.9380} + \frac{(K_P+K_I)z-K_P}{z-1} \frac{0.0321}{z-0.9380}} \quad (6.9)$$

simplifying Eq. 6.9, we obtain the characteristics polynomial

$$z^2 + [2b(K_P + K_I) - (a + 1)]z - 2K_Pb + a \quad (6.10)$$

Then using the same technique of queue placement, by setting settling time $k_s^* = 2$, and maximum overshoot $M_P^* = 0$, we can obtain

$$K_P = 3.129 \quad (6.11)$$

$$K_I = 3.152 \quad (6.12)$$

Figure 6.2 shows the result.

6.4 Assessments

We used the same configuration as the experiment described in Section 6.1, only letting the load balancer to run with the controller. We can see a dramatic change in performance in Figure 6.1 (right). When we start the *cpu_hog* process, there response time shoot up a little bit and quickly drops to normal level. Taking a closer look at the throughput data, we found that there is a very small number of log entries sent to the slow node, which matches the remaining capacity of this node and the response time on each TCQ node for a log entry is almost equal.

Chapter 7

Discussion

In this section, we discuss some experiences and hope to reveal some general ideas of applying control theory to computer systems, the advantages and limitations.

7.1 The advantages of using control theory in computer systems

Correctness can be analyzed.

Software system design usually involves a lot of heuristics, including ad-hoc feedback. For example, “when the load goes high, reduce the load”, is a very common heuristics even if no control theory is considered. However, feedback can cause problems if it is not analyzed and used correctly. This is because there is always a time delay in the control input change and the change of output. Failure to deal this delay correctly can cause system to become unstable spontaneously. Here we give an example which we experienced during the development of TCQ input control.

Figure 7.1 illustrated the instability caused by careless implementation of controller. The result queue length oscillates wildly, even with a small input data rate. This instability is caused by the delay in measurement of actual throughput. Although the controller design took the delay between the input rate change and the result queue length change, it did not consider the one period delay of calculating actual output data rate. This is a very subtle problem if we only use heuristics. But with a little control theory analysis and simulation,

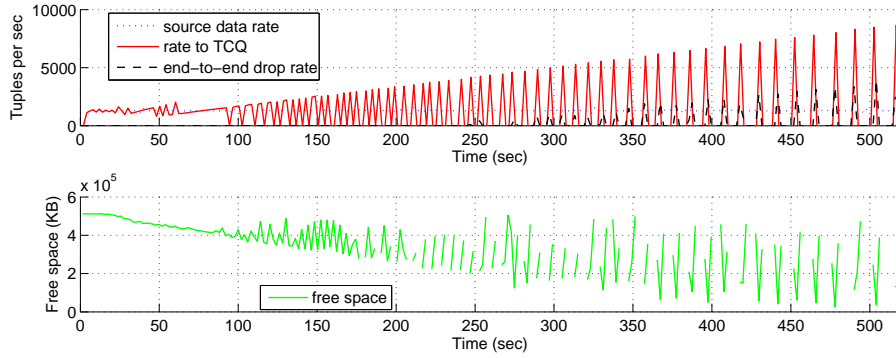


Figure 7.1: An unstable control system. A careless implementation that build a feedback loop for controlling the output rate of data source *inside* the feedback loop of the feedback loop for queue length as shown in Figure7.2. Notice that even the load is small, the system become unstable spontaneously.

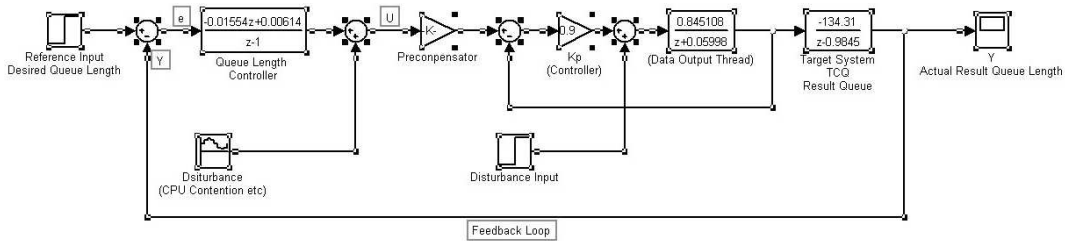


Figure 7.2: Block diagram of the unstable control system. A careless implementation that build a feedback loop for controlling the output rate of data source *inside* the feedback loop of the feedback loop for queue length.

we can reason about this.

It is often hard to build a first principle model for software systems due to their complexity and all the randomness in the operation environment. Models obtained from statistical characteristics of black-box building blocks are usually not accurate enough to be used in feed-forward control (i.e. without the feedback) A model with feedback control is more robust against disturbances, as Chapter 5 and 6 show. It also tolerates, to some extent, the inaccuracy of the model. For example, we used the same set of controller on two computers with completely different configuration and they still works stably (though the settling time becomes a little worse).

Implementation and system management is easier.

Feedback control reduces the complexity of system design. It only takes a few lines of Java code to implement a controller. Of course, we need to spend more code to obtain measurements, such as keeping record of the response time for tuples. We believe that this worth the trouble, because all systems that can be managed should at least be observable. Using control theory implicitly encourages a good practice of software design that will make the system easier to monitor and understood by the system administrators.

7.2 Limitations of control theory in computer systems

In order to successfully apply control theory in computer systems, the target system must have certain properties in addition to those discussed above.

A most important one is that the system, when takes in a finite input, must take a bounded time and space to process it, and produce a output in bounded size. There are certain cases that this property does not hold. In TCQ, when a data stream is joined with a large table on disk, the output caused by one input tuple can be potentially unbounded. We can deal with this problem by arranging data in blocks, and the next block of data is

produced only when user explicitly requests it (make another input).

Also, we need a system to have linear relationships between input and output. The input can be controlled, and the output must be measurable. In certain systems, this relationship is not easy to find. We believe that there are two solutions. First, by using statistical learning techniques, we can find usable relationships from a large number of data observed [10]. Second, with some simple modification of the target system, such as queueing explicitly, we can make the system controllable.

Since not all current software systems are designed to be "control theory friendly", sometimes control theory is hard to apply. An important goal of our future research is to find some good practice to design systems that can be controlled.

Chapter 8

Conclusion and Future Work

In this project, we designed and implemented a scalable, distributed system for system event log processing based on TelegraphCQ. This log processing system can be used in system monitoring and implementing autonomic computing algorithms.

The main features of this system include:

1. The system fully supports the languages and data types provided by TelegraphCQ;
2. It provides an easy way to run TCQ in parallel and the resulting system is scalable;
3. This system is easy to customize and manage;
4. It is self-adaptive to disturbances in the system due to the use of feedback control theory.

We also showed our experiences of applying feedback control theory in data stream processing systems, which can be generalized to other computer system too. We showed how to fix the problem of TCQ flow control, while considering it as a black-box system. We believe this is very common in building distributed systems using off-the-shelf software systems as building blocks. We also implemented a load balancer and load simulator with control theory.

Our experience shows that control theory has the following benefits:

1. The correctness can be analyzed mathematically;
2. Using control theory, we can usually get a self-adaptive system with an easy and clean implementation;
3. It encourages good system design, including making states of system easy to monitor.

We consider the following as our future work.

First, making this stream processing system available to system operators and researchers. We will implement a user interface allowing users to configure and monitor the whole system.

Second, we want to further improve the system by allowing it to scale up and down dynamically with the change of load and dynamic scheduling multiple streams.

Third, we want to investigate the use of adaptive controllers to allow wider operation range of the target system. Adaptive control has been used in [24] to address the problem of the narrow linear range by treating one large non-linear range as several small linear ranges. We want to explore possibility of improving operation range of the load balancer using this technique.

Last but most important, we are investigating how to accommodate more variances in workload, while still keeps the model and controller safe. We want to be able to support both queries running at each node changing and the changing of incoming tuple rate also.

Appendix A

Controller for load simulator

In this chapter, we will present the load simulator we build to do experiment on this system. We will also compare of two types of controllers, Proportional-Integral (PI) controller discussed in Section 5.2 and Proportional (P) controller (with a pre-compensator).

A.1 The inaccuracy of load simulator

As in many cases, we need to accurately simulate the workload we sent to the target system. In this experiment, we used a load simulator that reads input data from a file containing the source data from the disk, puts the data into its own buffer, and sends them out at a specified data rate. The data rate is specified as a function over time.

However, this naive implementation does not work well in practice. This is because time required for thread scheduling, disk read delay, and network latency are all unpredictable and cannot be controlled. Ignoring all those delays (by assuming them to zero) cause the actual data output rate stays always smaller than the desired value (see Figure A.1).

This may not be a serious problem if the inaccuracy of the workload only has performance impact but not affect correctness. However, this is not the case especially for generating workload for system identification (as in Section 5.3). In that case, a smaller or larger workload can cause the queue to fill up, causing inaccuracy in parameter estimation.

We believe this problem is typical to solve with feedback control theory, as it involves

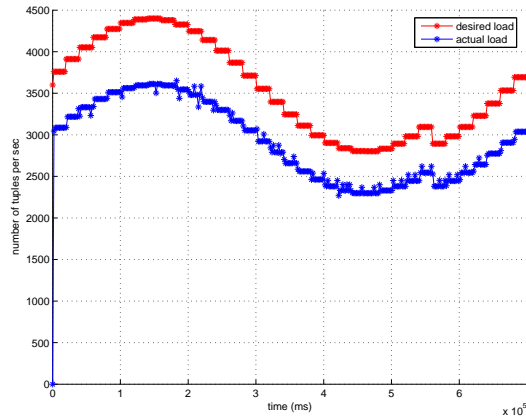


Figure A.1: A naive implementation of load simulator cannot achieve the desired load. The actual data rate is always lower than the desired data rate due to random disturbances in the system. The goal of the controller is to make the actual data rate equals to the desired data rate. Compare this with Figure A.4 and A.5

quantities that are hard to reason about accurately enough.

A.2 Control problem formulation

As we did in the previous examples, we first need to decide the inputs/outputs of the system. We use the desired data rate (specified by the data rate function) as reference input (R) and all the unknown factors, such as disk/network delay, thread scheduling delay etc. as disturbance input (D). The measured output is the actual data rate sent to the next tier (the load balancer in this case). This is obtained by counting the number of tuples we sent to output and average them over the sample time (2 seconds in this case). The 2 seconds interval is a magic number that is consistently used on all data measurements throughout the whole system. This sample time is obtained through observing the tradeoff between accuracy, settling time and smoothness of the observed changes over time. The target system is the output thread that takes in a desired number as data rate and output a data rate according to the desired number (from the previous time period).

The controller computes the control errors, and calculates control input U , which is the “fake” data rate that we want the data rate to generate instead of the lower real one. The

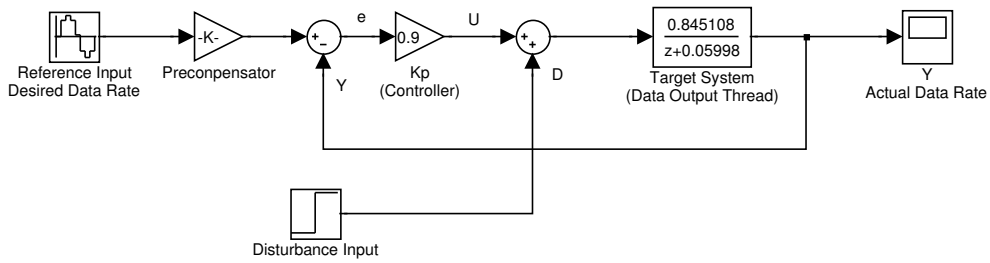


Figure A.2: Block diagram of workload simulator with P controller.

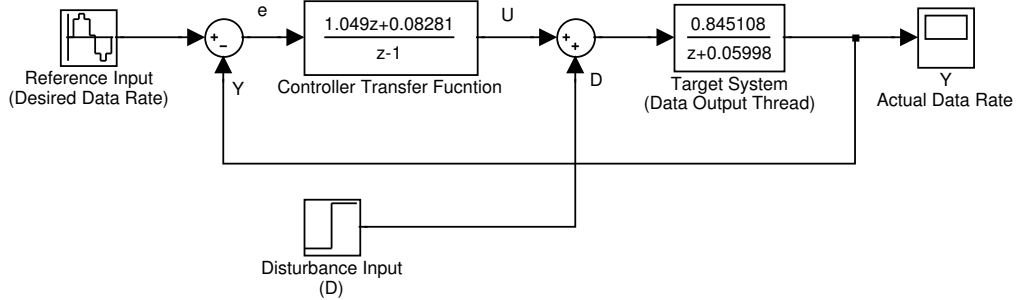


Figure A.3: Block diagram of workload simulator with PI controller.

block diagrams of the feedback control system are shown in Figure A.2 and A.3.

A.3 System Identification and Controller Design

A.3.1 System Identification

The system identification is easier than the case with controlling queue length Section 5.3. This is because the input range is large (in this case, we can use reference input ranging from 100 tuples per second to 4,000 tuples per second). We see that even within this large range, u and y keep a linear relationship and thus the predicted value and the experiment data fit pretty well.

However, as the data rate getting higher, the error between predicted value and experiment get larger; since more time is spend on the network delay and disk I/O. Until at a certain point, no matter how we increase the desired data rate, y almost does not increase any more. This kind of the non-linear relationships is common in computer systems, which is called *saturation*.

We choose the operation point at 2,140 tuples per second for input R , and the corresponding operation point for output Y is 1,775 tuples per second.

Using Matlab code in Figure 5.6, we obtain the model for the target system.

$$y(k + 1) = -0.05998y(k) + 0.8451u(k) \quad (\text{A.1})$$

Or expressed in z-domain as a transfer function as:

$$G(z) = \frac{Y(z)}{U(z)} = \frac{0.8451}{z + 0.5998} \quad (\text{A.2})$$

A.3.2 Controller Design

In this section, we discuss the use of Proportional (P) controllers and Proportional-Integral (PI) controllers, both of which are simple but commonly-used controller designs in real systems.

We implement both P controller and PI controller for load simulator control (Figure A.2 and Figure A.3) and compare the results.

PI controller

Proportional-integral (PI) controllers, as described in Section 5.3, calculate control input using the current error as well as the last error and the control input for last step.

It can be proved that PI controller always drive the steady state error to zero. However, the PI controller it is usually slower than P Controller. We will show the effect of being slower in Section A.4.

The control law for PI controller is (as Eq. 5.2):

$$u(k) = u(k - 1) + (K_p + K_I)e(k) - K_p e(k - 1) \quad (\text{A.3})$$

Following exactly the same procedure as in Section 5.3, we can obtain:

$$K_P = -0.0828 \quad (\text{A.4})$$

$$K_I = 1.06 \quad (\text{A.5})$$

P controller

In P Controller the control input u is simply control error e multiplied by a constant K_p . It is simple and fast to converge to steady state. However, it does not handle disturbances well in the system.

The advantage of P Controller is that it is relatively simple to design, since its control input is control error multiplied by a constant K_P . The control law is:

$$u(k) = K_P e(k - 1) \quad (\text{A.6})$$

where

$$e(k - 1) = r(k - 1) - y(k - 1) \quad (\text{A.7})$$

P-Controller is also fast in terms of the time required to converge to the steady state. This is because it does not use the integral term that requires several cycles to converge.

We also use *pole placement* techniques to choose K_P . Using Eq. 5.10, we can write the the close-loop transfer function from input R to output Y as:

$$F_R(z) = \frac{Y(z)}{R(z)} = \frac{K_P b}{z - a + K_P b} \quad (\text{A.8})$$

There is only one pole of Equation A.8, which is

$$p = a - K_P b \quad (\text{A.9})$$

To make the system stable, the pole must lay within the unit circle on the complex plane.

That is to say

$$|a - K_P b| < 1 \quad (\text{A.10})$$

Or,

$$\frac{a-1}{b} < K_P < \frac{1+a}{b} \quad (\text{A.11})$$

We also want to consider settling time k_s . The same as in Section 5.3, we have

$$k_s = \frac{-4}{\log|a - K_P b|} \quad (\text{A.12})$$

Giving the the maximum settling time we can tolerate, k_s^* , we can solve for the possible range for K_P

$$\frac{-4}{\log|a - K_P b|} < k_s^* \quad (\text{A.13})$$

When the dominate pole p is real and $p \geq 0$, there is no overshoot, so $M_P = 0$. When p is real and $p < 0$, the maximum overshoot $M_P = |p|$. So if we want to place the pole to the negative half of real axis, we need to consider the maximum overshoot M_P^* we can tolerate, so we need to the following condition:

$$M_P = |a - K_P b| < M_P^* \quad (\text{A.14})$$

The last thing we want to consider is the accuracy of the controller, i.e., the steady state error e_{ss} . Basic results in control theory tells us

$$F_R(1) = \frac{y_{ss}}{r_{ss}} \quad (\text{A.15})$$

Thus,

$$e_{ss} = r_{ss}[1 - F_R(1)] \quad (\text{A.16})$$

Limiting the maximum e_{ss} to be e_{ss}^* , we obtain the following condition:

$$r_{ss}[1 - F_R(1)] < e_{ss}^* \quad (\text{A.17})$$

$$r_{ss}\left[1 - \frac{K_P b}{1 - a + K_P b}\right] < e_{ss}^* \quad (\text{A.18})$$

Solving the set of inequalities A.11, A.13, A.14 and A.17 will provide us with the range that K_P can be in. However, those four inequalities may not be able to hold at the same

time. Of course, when this happens, we must keep stability (A.11) satisfied by relaxing other constraints.

The problem with P control is that it is not accurate, that is e_{ss} is always greater than zero. In fact, it can never achieve zero steady state error, since if that happened, e will be zero, and u will also be zero (which means we don't get any data output any more).

We make P controllers more accurate by adding a pre-compensator. A pre-compensator is a transfer function $P(z)$ that changes the reference input in order to make the system accurate.

The feedback system with pre-compensator $P(z)$ has transfer function

$$F'_R(z) = \frac{Y(z)}{R(z)} = \frac{K_P b P(z)}{z - a + K_P b} \quad (\text{A.19})$$

Note that $P(z)$ is not in the feedback loop so that $P(z)$ only appears in the nominator, but not the denominator of $F'_R(z)$.

To make the steady state error e_{ss} to be zero for any r_{ss} in Eq.A.16, we need to make $1 - F'_R(1) = 1$

$$F'_R(1) = \frac{Y(1)}{R(1)} = \frac{K_P b P(1)}{1 - a + K_P b} = 0 \quad (\text{A.20})$$

Solve for $P(z)$, we get

$$P(z) = 1 + \frac{1 - a}{K_P b} \quad (\text{A.21})$$

From the calculation above, we can see that without disturbances, P controller with a pre-compensator designed above always drives the steady state error e_{ss} to zero. However, it cannot eliminate the uncontrolled disturbance input. We will see this effect in the assessment part.

With the data we get from system identification,

$$a = -0.0600 \quad (\text{A.22})$$

$$b = 0.845 \quad (\text{A.23})$$

We notice that K_P must be a positive number, since a “negative” data rate does make sense. This leads to an extra constraint on K_P

$$K_P > 0 \quad (\text{A.24})$$

For stability (Eq. A.11), we get

$$0 < K_P < 1.11 \quad (\text{A.25})$$

We want maximum settling time to be 1 sample times, which is about 2 seconds. We get from Eq. A.13

$$K_P > 0.85 \quad (\text{A.26})$$

Let the maximum overshoot to be 0.05. Using Eq. A.14, we get

$$K_P > -0.01 \quad (\text{A.27})$$

We did not consider e_{ss} , since we want to use the pre-compensator. From inequality constraints A.24,A.25,A.26 and A.27, we get

$$0.85 < K_P < 1.1 \quad (\text{A.28})$$

We choose K_P to be 0.9, and estimate the pre-compensator $P(z)$ using Equation A.21, we have

$$P(z) = 1 + \frac{1-a}{K_P b} = 2.39 \quad (\text{A.29})$$

The results obtained for K_P and $P(z)$ are shown in the block diagram (Figure A.2).

A.4 Assessment

In Figure A.4 and A.5, we can see both of the P and PI controllers make the output data rate much more accurate than the case shown in Figure A.1. We varied the desired data rate as a sine wave around the desired operation point, and we can see that the actual output data rate follows this change very closely.

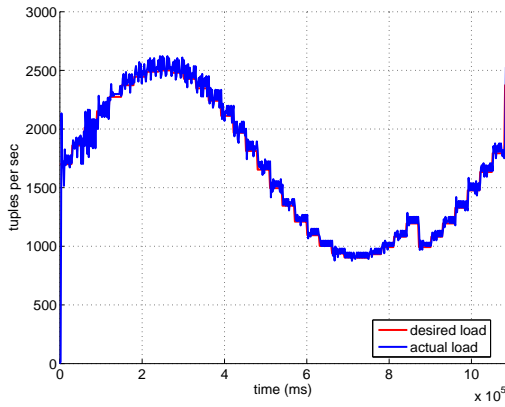


Figure A.4: Effect of P controller in workload simulator

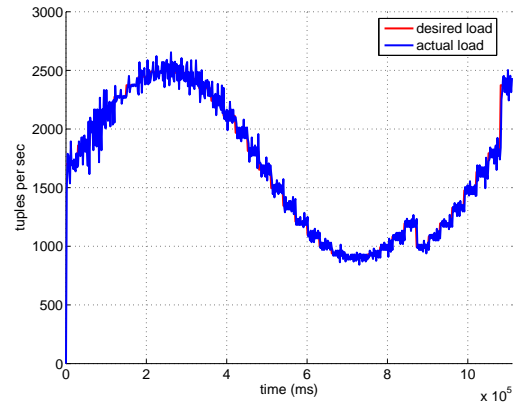


Figure A.5: Effect of PI controller in workload simulator

In order to make comparison between the P controller and PI controller, we take a “zoomed in” version of the figures in Figure A.6 and A.7.

The interesting effect we can see is that the actual output for P controller is always a little higher than the desired data rate, even with the pre-compensator. This is because there are always disturbances in the system which we did not capture during system identification face. These disturbances can be short, such as a background daemon process, or can be long and slow in effect, such as memory leaking. Also, the “linear” assumption is only approximately true. P controller does not help us correct those disturbances, and the error cannot be zero.

In contrast, PI controller can always get steady state error to zero even though there are disturbances that are not captured by the model of system. Intuitively, this is mainly because it accumulates the past errors, and this integral of errors helps to drive the steady state error to zero, regardless of the unknown disturbances.

We noticed that there are regular small oscillations in the both P and PI cases. It turns out that those small disturbances come from Java garbage collection. The garbage collection is running frequently in the workload simulator, because the simulator maintains a fairly large buffer pool as Java objects array, and creating and discarding string objects for every

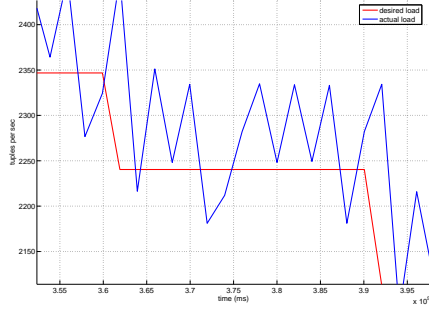


Figure A.6: Effect of P controller in workload simulator (zoom in). The actual data rate is a little higher than desired rate even with the pre-compensator $P(z)$. This is because the random disturbances in the system not captured by the system model. However, comparing to Figure A.7, it has less oscillation, because P controller is faster in reaction.

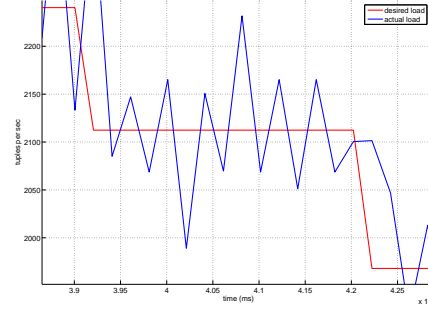


Figure A.7: Effect of PI controller in workload simulator (zoom in). The actual data rate oscillates around the desired data rate. As expected PI controller helps us to get e_{ss} to be zero. However, it oscillates more than P controller when there are lots of small disturbances (such as Java garbage collection) in the system. This is because of the relatively slower settling time k_s .

tuple it reads from disk and output to network.

The default Java garbage collection is not incremental, which will result in huge oscillations once every 20 seconds. Though the controller corrects this disturbance in a very short period of time, the oscillation is still too big to tolerate. After we changed Java garbage collection to be an incremental one, the big disturbances become many small ones, as shown in Figure A.6 and A.7.

As a future topic, we want to eliminate those small disturbances by not letting Java dynamically allocate buffer spaces. This will also improve the performance of our workload simulator.

Bibliography

- [1] Charu C. Aggarwal. A framework for diagnosing changes in evolving data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 575–586. ACM Press, 2003.
- [2] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM Press, 2002.
- [3] Peter Bodik, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, Seattle, 2005.
- [4] Aaron B. Brown and David A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 Usenix Annual Technical Conference*, San Antonio, TX, USA, June 2003.
- [5] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, 2004. USNIX.
- [6] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, July 2000.
- [7] Jeffrey D. Case, Mark S. Fedor, Martin Lee Schoffstall, and James R. Davin. A simple network management protocol (snmp). *RFC1157*, May 1990.
- [8] Mike Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. DSN 2002.
- [9] Mike Chen, Alice Zheng, Jim Lloyd, Michael Jordan, and Eric Brewer. A statistical learning approach to failure diagnosis. In *International Conference on Autonomic Computing (ICAC-04)*, New York, NY, May 2004.
- [10] Ira Cohen, Jeffrey S. Chase, Moisés Goldszmidt, Terence Kelly, and Julie Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, pages 231–244, 2004.

- [11] Hewlett-Packard Development Company. Hp OpenView. <http://www.openview.hp.com/>, 2005.
- [12] IBM Corporation. Tivoli. <http://www.ibm.com/software/tivoli/>.
- [13] Microsoft Corporation. Microsoft Operations Manager. <http://www.microsoft.com/mom/>.
- [14] Yixin Diao, Neha Gandhi, Joseph L. Hellerstein, Sujay Parekh, and Dawn M. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server. In *Proceedings of Network Operations and Management Symposium (NOMS02), 2002*, pages 219–234. IEEE/IFIP, 2002.
- [15] Yixin Diao, Joseph L. Hellerstein, Adam J. Storm, Maheswaran Surendra, Sam Lightstone, Sujay Parekh, and Christian Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *Proceedings of the American Control Conference, 2004.*, volume 3, pages 2045–2050, 2004.
- [16] Yixin Diao, Joseph L. Hellerstein, Adam J. Storm, Maheswaran Surendra, Sam Lightstone, Sujay S. Parekh, and Christian Garcia-Arellano. Incorporating cost of control into the design of a load balancing controller. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 376–387, 2004.
- [17] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Querying and mining data streams: you only get one look a tutorial. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 635–635. ACM Press, 2002.
- [18] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, Aug 2004.
- [19] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [20] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with PIER. In *Proceedings of the 29th VLDB Conference*, 2003.
- [21] Emre Kiciman and Armando Fox. Detecting and localizing anomalous behavior to discover failures in component-based internet services.
- [22] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.
- [23] Benjamin C. Ling, Emre Kiciman, and Armando Fox. Session state: Beyond soft state. In *Proceedings of the 1st Symposium on Networked Systems Design*, page 295C308, San Francisco, CA, 2004. USNIX.
- [24] Xue Liu, Xiaoyun Zhu, Sharad Singhal, and Martin Arlitt. Adaptive entitlement control to resource containers on shared servers. In *Proceedings of the Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*. IEEE, 2005.
- [25] Chris Lonvick. The BSD Syslog Protocol. RFC 3164 (Informational), August 2001.

- [26] Sheng Ma, Joseph L. Hellerstein, Chang shing Perng, and Genady Grabarnik. Progressive and interactive analysis of event data using event miner. In *IEEE International Conference on Data Mining*, 2002.
- [27] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM Press, 2003.
- [28] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 49–60. ACM Press, 2002.
- [29] John Markoff and G. Pascal Zachary. In searching the web, google finds riches. *NY Times*, April 13, 2003.
- [30] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems*, March 2003.
- [31] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. approaches-roc. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science Technical Report, 2002.
- [32] David A. Patterson. A simple way to estimate the cost of downtime. Submission to 16th Systems Administration Conference (LISA '02), 2002.
- [33] Johannes Gehrke Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill Higher Education, 2003.
- [34] Ramendra K. Sahoo, A. Oliner, Irina Rish, Manish Gupta, José E. Moreira, Sheng Ma, Ricardo Vilalta, and Anand Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *KDD*, pages 426–435, 2003.
- [35] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60, 2004.
- [36] Sailesh Krishnamurthy Sirish. Telegraphcq: An architectural status report.
- [37] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In Roch Guerin, editor, *Proceedings of SIGCOMM-01*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, August 27–31 2001. ACM Press.
- [38] Mark Sweiger, Mark Madsen, Jimmy Langston, and Howard Lombard. *Clickstream Data Warehousing*. John Wiley & Sons, 2002.
- [39] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

- [40] Ricardo Vilalta, Chidanand Apté, Joseph L. Hellerstein, Sheng Ma, and Sholom M. Weiss. Predictive algorithms in the management of computer systems. *IBM Systems Journal*, 41(3):461–474, 2002.
- [41] Ricardo Vilalta and Sheng Ma. Predicting rare events in temporal domains using associative classification rules. Technical report, IBM Research, T. J. Watson Research Center, Yorktown Heights, NY, 2002.
- [42] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, pages 245–258, 2004.
- [43] Shaula Alexander Yemini, Shmuel Kliger, Eyal Mozes, Yechiam Yemini, and David Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, pages 82 – 90, 1996.