Passive, automatic detection of network server performance anomalies in large networks

Jeff Terrell

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2009

Approved by:

Kevin Jeffay, Advisor

F. Donelson Smith, Reader

Ketan Mayer-Patel, Reader

Vern Paxson, Reader

Jim Gogan, Reader

Abstract

JEFF TERRELL

Passive, automatic detection of network server performance anomalies in large networks

(Under the direction of Kevin Jeffay.)

"Network management" in a large organization often involves—whether explicitly or implicitly—the responsibility for ensuring the availability and responsiveness of network resources attached to the network, such as servers and printers. Users often think of the services they rely on, such as web sites and email, as part of "the network". Although tools exist for ensuring the availability of the servers running these services, ensuring their performance is a more difficult problem.

In this dissertation, I introduce a novel approach to managing the performance of servers within a large network broadly and cheaply. I continuously monitor the border link of an enterprise network, building for each inbound connection an abstract model of the application-level dialog contained therein without affecting the operation of the server in any way. The model includes, for each request/response exchange, a measurement of the *server response time*, which is the fundamental unit of performance I use. I then aggregate the response times for a particular server into daily distributions. Over many days, I use these distributions to define a *profile* of the typical response time distribution for that server. New distributions of response times are then compared to the profile to determine whether they are anomalous.

I applied this method to monitoring the performance of servers on the UNC campus. I tested three months of continuous measurements for anomalies, for over two hundred UNC servers. I found that most of the servers saw at least one anomaly, although for many servers the anomalies were minor. I found seven servers that had severe anomalies corresponding to real performance issues. These performance issues were found without any involvement of UNC network managers, although I verified two of the issues by speaking with network managers. I

investigated each of these issues using the contextual and structural information measured at the border link in order to diagnose the cause of the issue. I found a variety of causes: overloaded servers from increased demand, heavy-hitting clients, authentication problems, changes in the nature of the request traffic, server misconfigurations, *etc.* Furthermore, information about the structure of connections proved valuable in diagnosing the issues.

To the dearest and best, the Lord Jesus Christ. You are my rock.

Acknowledgments

Thanks to Joni Keller, John McGarrigle, Matt Heinze, Joe Morris, Mike Seda, and the many fine folks at UNC ITS who supported me and helped me trace the cause of issues.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACK	acknowledgement
ADU	abstract data unit
API	application programming interface
AS	autonomous system
CDF	cumulative distribution function
DHCP	dynamic host configuration protocol
DNS	domain name system
Gbps	giga-bits per second
GHz	giga-hertz
HTTP	hyper-text transfer protocol
HTTPs	secure hyper-text transfer protocol
IMAP	Internet message access protocol
ISN	initial sequence number
ISP	Internet service provider
IP	Internet protocol
Mbps	mega-bits per second
MSS	maximum segment size
NIC	network interface card
PCA	principal components analysis
pcap	packet capture (library/file format)
PDF	probability distribution function
PoP	point-of-presence
POP3	post office protocol, version 3
QF	quantile function
RTO	retransmission timeout
RTT	round-trip time

SNMP	simple network management protocol
SMTP	simple mail transfer protocol
SQL	structured query language
TCP	transmission control protocol
UDP	user datagram protocol
UNC	University of North Carolina (at Chapel Hill)
VLAN	virtual local area network

Chapter 1

Introduction

"Network management" in an enterprise environment often carries an implicit responsibility for ensuring the performance of network resources such as servers and printers attached to the network. For better or worse, users tend to view the "network" broadly to encompass the services they rely on as well as the interconnections between machines. Therefore, it is in the network manager's interest to proactively manage the end system hosts that implement services in the network.

Broadly speaking, there are primarily two aspects of the end systems that should be managed: whether they are operational, and how well they are performing. The former is called *fault management*. I will focus on the latter problem, called *performance management*. Specifically, my primary concern is *detecting* performance issues with servers in a large network. An example of a performance issue is when a server that typically takes a few tens of milliseconds to respond to requests begins to take a few seconds to respond. My secondary concern is providing information that is useful in *diagnosing* the cause of a performance issue.

A common approach to server performance management is to measure the CPU utilization, memory utilization, and other metrics of the server operating system. One problem is that such metrics do not necessarily correlate with the user-perceived quality of service. Another problem with this approach is that the act of measuring computational resources consumes the very same resources that the service itself needs.

Another approach is called *probing* or *active measurement*, in which the server is given actual requests, and the response time is measured. This approach more accurately reflects the quality

of service as perceived by the user. However, the measurement negatively affects the service itself—sometimes noticeably. For example, when the quality of service is suffering because of high load, the act of measuring the quality of service will only exacerbate the problem.

In contrast, I propose a measurement solution that is purely *passive*. No aspect of the measurement perturbs the operation of the service in any way. Instead, the source of data for my measurements is the network traffic already flowing to and from the server. Furthermore, rather than putting a measurement device in line with the network path (thus affecting the service by adding a delay to the delivery of the packets), the measurement device sits in parallel with the network path, receiving a *copy* of the packets.

The passive approach I adopt does not make my solution unique by itself; its *generality* over application services is another essential component of its novelty. Consider the most common types of servers. HTTP (or web) servers get requests for individual HTTP objects (referenced by a URL), and produce the appropriate response: either the requested object or an error code. SMTP servers receive email messages that are sent by a client, and send them on their way to the destination email server. IMAP servers deliver email messages to email clients and modify email folders as requested by the client. All of these servers provide different application services, but at a high level, they are all responding to requests. It is at this higher level that my approach models the server communication in order to detect and diagnose server performance issues.

From a user's perspective, the *response time* is a natural measure of performance: the longer the response time, the worse the performance. For example, if there is a noticeable delay between clicking on a "get new e-mail" button and seeing new e-mail appear in one's inbox, then the response time is high, and the performance is low. Such a conception of performance is naturally supported within the request-response paradigm.

Current passive solutions to monitoring server performance will typically target the most common types of servers, ignoring hundreds of other types. (See Chapter 2 for examples.) If a new type of service is created, the developers of the monitoring solution will need to understand the application-level protocol upon which the service is based, and incorporate the protocol into their solution. If the protocol is proprietary, then they must guess at its operation. In contrast, my solution incorporates no knowledge about specific application-level protocols, so it works

equally well for all TCP-based request/response services, independent of whether they are based on new, proprietary, or standard application-level protocols. (Note that my solution does *not* work for UDP and other non-TCP types of traffic. Most servers of interest are TCP-based. Notable exceptions include voice-over-IP and streaming media.) As a result, my measurements of server performance do not require access to the application-level information present in a server's protocol messages.

A common problem in many network measurement techniques is the inability to monitor traffic in which the application payload is encrypted and thus unintelligible without the private decryption key. Because these techniques inspect the application payload, encryption foils their measurements. My technique depends only on the packet headers, and thus can be used with traffic that is encrypted. To use an analogy, consider packets as envelopes. Unlike other approaches, my approach looks only at address and other information on the outside of the envelope. Thus, when an envelope is securely sealed (as in the case of encryption), being unable to open the envelope is not a problem for my approach.

1.1 Contributions

The primary contribution of this dissertation is to establish the feasibility of measuring and analyzing the performance of services, based only on the transport-level information in TCP/IP protocol headers on packets sent and received by servers. Specifically, this contribution comprises two sub-contributions. First, I *measure* the server performance by building a compact representation of the application-level dialog carried over the connection (Chapter 3). Second, I *analyze* the performance in three parts: building a profile of typical (or "normal") server performance (Chapter 4); detecting performance anomalies, or deviances from this profile (Chapter 4); and diagnosing the cause of such anomalies (Chapter 5).

My thesis statement is as follows:

> It is possible to *detect* and, to an extent, *diagnose* server performance issues in a network with hundreds of servers providing multiple diverse types of services without perturbing those services, by accessing only the information available in the TCP/IP headers of the packets the server receives and sends.

I will now introduce my solution and list my novel contributions.

I use the *server response time* to measure the server performance. The response time is the amount of time elapsed between a request from the client to the server and the server's subsequent response. (I do not claim that this measure is novel.) Intuitively, the longer the response time, the worse the performance. A naive approach to detecting server performance issues using the server response time will not work. We cannot simply flag long response times as indicative of anomalies for two reasons. First, the definition of what is anomalous is different for different servers. For example, an SQL database server might be expected to take minutes or even hours to respond, whereas a second might be a long time for a chat server's response. Second, any server will have at least a few long response times as part of its normal operation. For example, most email messages sent to a outgoing mail server can be sent in less than a second, but the occasional five megabyte attachment will take longer, and we would not call that longer response time an anomaly.

To demonstrate the feasibility of my solution, I measured server response time by using a network traffic *monitor* to observe the traffic flowing between UNC and its commodity Internet up-link. The monitor observes any TCP connection involving a UNC host and an external host[1]. This setup achieves passive measurement, as I will discuss in more detail in Section 3.1.1.

Instead of using a simple threshold on the server response time to detect anomalies, my approach builds, for each server, a profile of the typical *distribution* of response times for that server. The profile is, in a sense, a *summary* of many response times, and detecting a performance issue is fundamentally a determination of how well the summary of recent response times compares to the historical profile (or profiles). Because each server has its own profile, observing ten-second response times might be indicative of a performance issue for some servers but not for others.

Not only is a profile unique to a given server, but they also vary by different amounts. For example, the individual distributions comprising the profile for one server might be nearly identical. In that case, a new distribution that deviates from the profile even a small amount might be considered highly unusual. On the other hand, other servers might have a profile

[1]except for certain networks that are routed on one of the other routes to the Internet, such as educational institutions (on Internet2)

containing individual distributions that vary widely. In that case, a new distribution would need to be extremely different from the profile before it was considered unusual. Even more subtle, a profile might tolerate wide variation in some places but little variation in other places. For example, many services see wide variation among the tail of response time distributions, but little variation elsewhere, so it makes sense to penalize a distribution that deviates from the profile only in the tail less than one that deviates from the profile only in the body.

The approach corresponds with a user's intuitive understanding of performance issues. A user is not likely to complain about the performance of a server that is always slow; instead, the user learns to expect that behavior from the server. Also, if the response times vary greatly for the server, the occasional long response time will probably not bother the user greatly; however, a long response time in a typically very responsive service such as an interactive remote-desktop session would be more likely to elicit a complaint. Thus, there is a correlation between what a user would deem problematic performance and what my methods detect as problematic performance. I will demonstrate this correlation more quantitatively in Section 5.4. Furthermore, note that, by continually applying this approach, the network manager can now *proactively* discover problems and hopefully fix them before they become severe enough to elicit user complaints.

It is not just performance *degradations* that are interesting to a network manager, but, more generally, performance *anomalies*. A good solution should detect when the performance for a server is unusually bad and also when it is unusually *good*. For example, a misconfiguration might be introduced by a web server upgrade, causing the web server to serve only error pages and yielding faster response times than usual. My approach detects this sort of performance anomaly as well.

In addition to discovering performance issues, I also provide a way to *diagnose* the issues. This diagnosis is necessarily from an outside perspective, and I cannot incorporate knowledge of packet payloads. However, we will see in Chapter 5 that the diagnosis can be quite helpful to the network manager. To diagnose issues, I can use contextual information provided by my measurement tool, such as the number of connections to a server, as well as the structure of the application-layer dialogs of connections. For example, one diagnosis might be that the third response time of each connection is the one causing the performance anomaly.

In the process of evaluating my thesis, I have collected a multi-terabyte dataset of information about UNC network traffic. This dataset may conceivably have uses other than server performance management, and I will make an anonymized version available to network researchers through the Internet Measurement Data Catalog[2].

Because my approach has a single monitoring point as a data source, it is easily deployed. Furthermore, the hardware driving my approach is relatively inexpensive, especially compared to some commercial solutions requiring more extensive deployment of hardware throughout the network.

Another aspect of my method that is important in large networks is the ability to automatically discover servers and build a profile for them. It is difficult in a large organization to know about all the servers in the network, especially when individual departments and divisions are free to setup their own servers without the aid of the technology support department.

Another important aspect of my solution is that they are tunable to various situations. For example, we will see in Section 5.5 a server that experienced a profound, ongoing shift in the response times. My methods provide the network manager the ability to redefine what "normal" traffic looks like for that server, to avoid being bombarded with alarms. Also, various parameters control the performance anomaly detection mechanism, which I discuss in more detail in Sections 4.7 and 4.8.

In summary, my novel contributions are as follows:

- A one-pass algorithm for continuous measurement of server performance from TCP/IP headers at high traffic rates (Section 3.1)
- A tool for measuring server performance using a one-pass algorithm that can keep up with high traffic rates (Section 3.1)
- A multi-terabyte, months-long dataset of server performance and other information (Section 3.3)
- A method for building a profile of "normal" performance for a server (Chapter 4)
  - ...regardless of the *scale* of response times

[2]http://imdc.datcat.org

  - ...regardless of the *distribution* of response times
  - ...regardless of how widely the performance typically varies for the server

- A metric characterizing the extent to which a distribution of response times is anomalous relative to the normal profile (Section 4.4)
- The ability to **diagnose** performance issues (Chapter 5).

The rest of this dissertation is organized as follows. First, the rest of this chapter provides an overview of the dissertation. Chapter 2 provides some background and discusses related work. Chapter 3 discusses the measurement of server performance, including the algorithms involved and the data generated by the measurement. Chapter 4 details the performance anomaly detection techniques I used, which are based on statistical techniques originally developed for a medical imaging application. Chapter 5 gives a series of eight case studies involving servers in the UNC network that had severe anomalies, to showcase the diagnosis abilities afford to a network manager by my approach. Chapter 7 concludes, and Appendix A gives the full anomaly detection results.

1.2 Overview

Chapter 2 reviews the background and related work. First, I review the sliding window mechanism of TCP in Section 2.1.1. The sliding window is the mechanism by which a byte stream is reliably transported from the sender to the receiver over the unreliable communication channel of a computer network. To accomplish reliable transport, it is necessary to associate each byte of the application's byte stream with a *sequence number*. Then, each packet refers to a specific range of sequence numbers. The sequence numbers are the essential association by which I can infer the request and response boundaries intended by the application—an inference that I discuss in detail in Chapter 3.

These boundaries inform the creation of a compact representation of the application-level dialog between two end systems, called a *connection vector*. A connection vector is an instance of the A–B–T model, which is reviewed in Section 2.1.2. At a high level, the model includes three components: requests, or application-defined chunks of data sent from the connection

initiator ("client") to the connection acceptor ("server"), responses, or chunks of data sent in the opposite direction, and *think-times*, or the times between such chunks. The server-side think-times, or the time elapsed between the end of a request and the beginning of a response, are called "response times". These response times are the fundamental unit of service performance upon which my performance anomaly detection methods are based, as detailed in Chapter 4.

Section 2.2 reviews related work. First, I review other research using passive network measurement in Section 2.2.1. I place special emphasis on systems that are not only passive but also support *continuous* measurement. Next, in Section 2.2.2, I discuss the FCAPS model of network management, which provides a description of the space of problems related to network management. I also discuss where my efforts fall within this space. Then I review other network management solutions in Section 2.2.3. Lastly, I review other models of network connections in Section 2.2.4.

Chapter 3 discusses `adudump`, the tool that I use to measure response times and other components of the A–B–T model in real-time on high-speed links. I begin by discussing the measurement setup that I have used in the UNC network in Section 3.1.1. The monitor, or the system on which `adudump` executes, receives a copy of every packet transmitted across the border link. Thus, the monitor observes all connections involving one local UNC host and one external Internet host. However, only connections for which the connection acceptor (or server) is on the local side are observed. In other words, because of my focus on the performance of servers within one network management domain, I ignore connections for which the server is external.

The packets are copied by a fiber-optic splitter, which splits the optical signal into two paths, so the network tap is passive, introducing no perturbations to the network. A copy of each packet is sent to the monitor, which passes the information to the `adudump` process. I used an Endace DAG card to capture and timestamp each packet at the monitor, but a standard fiber-optic NIC would work also.

The border link routinely carries 600 Mbps of traffic, with spikes up to 1 Gbps. Because of the efficient algorithms of `adudump` and the special purposes DAG card hardware, the monitor can keep up with this rate of traffic with no packet losses, even on relatively old hardware (a

1.4 GHz Xeon processor and 1.25 GB of RAM).

My `adudump` tool is not the first tool to infer A–B–T instances (called *connection vectors*) from a sequence of TCP packet headers. However, it is the first tool to do so in a *one-pass* fashion. Earlier approaches [HC06] captured a packet header trace, then analyzed it offline. My approach observes a packet, modifying the saved state for its connection as necessary, and discards it. This approach is simple in concept, but complex in realization. TCP is designed to provide reliable transport of a byte-stream over an unreliable medium, so it must provide mechanisms for loss detection and retransmission. As such, my algorithms must make inferences based on what is often incomplete knowledge about the application's behavior. For example, if the first packet of a response is lost before it reaches the vantage point of the monitor, but the second packet reaches the monitor, I must infer (based on the next expected sequence number in the byte-stream) that the responding application intended to start its response earlier. Multiple losses at key points in the connection make for complicated inference algorithms.

The `adudump` tool reports the individual elements of the A–B–T model as "`ADU`" records. The direction of an ADU indicates whether it is a request or a response. The subsequent think-time is also reported; think-times following a request ADU are server response times. The server and client addresses and port numbers are also reported, as well as an indication of whether the connection is currently considered "sequential" (*i.e.* either application waits for the other to finish its transmission before responding) or "concurrent" (*i.e.* both applications transmit at the same time). In addition, each record has a timestamp in the form of a unix epoch with microsecond precision. The timestamp corresponds to the time at which the current packet was received by the DAG card. In general, `adudump` prints records as soon as it can, which means that *e.g.* the timestamp of a request ADU is the timestamp of the first packet of the server's response (that also acknowledges all previous request bytes); that is, the earliest point at which `adudump` can infer that the request is finished.

Information about the connection context is also reported by `adudump`. A "`SYN`" record indicates that a client has attempted to initiate a TCP connection with a server. A "`RTT`" record measures the time between the TCP SYN segment (the client's attempt to establish a connection) and the TCP SYN/ACK segment (the server's initial response), which is known as the RTT, or round-trip-time. Note that the round-trip-time is relative to the vantage point of

the monitor (*e.g.* the border-link), not the client. A "`SEQ`" record indicates that the final part of the TCP "three-way handshake" is completed, and a TCP connection is officially established. This event is named "`SEQ`" because connections are sequential by default, until there is evidence of concurrency, in which case a "`CNC`" record is printed. When the connection is closed, an "`END`" record is printed. Lastly, if the `adudump` process is killed (or it reaches the end of a packet header trace), it will print an "`INC`" record for every ADU that was in the process of being transmitted.

It is useful to note two caveats about `adudump`. First, it does not report any information from a connection that it does not see the beginning of. It is sometimes difficult to make reasonable inferences even when seeing the exchange of initial sequence numbers in the connection establishment. Determining the connection state from the middle of a connection is an area of ongoing work. The second caveat is that the server response times are not purely application think-times, because they include the network transit time from the monitor's vantage point to the end system and back. However, most network transit times in the UNC network are quite small, so this does not present a problem (see Section 3.1.2).

Section 3.1.3 introduces the data structures used by `adudump` for A–B–T inference. There are two primary data structures. One includes the salient features of the current TCP packet. The other is a table of connection states, which includes, for each connection being tracked by `adudump`, all of the necessary state variables, including the connection state (open, closed, etc.), the local and remote addresses and port numbers, and further state variables for the inbound and outbound flows in the connection.

Section 3.1.4 lists and discusses the inference algorithms used by `adudump`. Each of these algorithms is executed in particular situations. For example, Algorithm 1 is executed when `adudump` sees a TCP SYN segment and has no connection-tracking state for that connection. Algorithm 7 covers the situation in which an inbound data segment arrives in a sequential connection, and the flow in which the segment is sent is marked as "`ACTIVE`" (*i.e.* an ADU is being sent in that flow).

A high level observation of the algorithms is that this problem is nontrivial. It is relatively easy to get the A–B–T inference algorithms of `adudump` working correctly for 99 percent of connections; however, the last 1 percent is difficult. There are many subtleties to consider, and the loss of arbitrary packets complicates every aspect of the inference. Even with the extensive

engineering, however, there are still fundamental limitations to this approach, which I discuss in Section 3.1.5. For example, should the "`last_sent`" flow state variable be updated when the current packet is a retransmission? The answer will determine whether `adudump` splits certain ADU's because of an application quiet time; however, the application intent cannot be accurately inferred in this case, because the information is simply not available with a passive approach.

Because the inference algorithms that `adudump` uses are complex, it is important to validate that they are correct. Section 3.2 discusses the validation of `adudump`. I used a set of synthetic applications to generate and send/receive arbitrary ADUs with arbitrary think-times. I then compared the intent of the applications, as recorded by the application itself, against the inferences made by `adudump`. The differences are very slight; the distributions line up almost perfectly in all cases.

I ran `adudump` to measure A–B–T connection vectors and contextual information continuously for several months, starting in March, 2008, and ending in April, 2009. Each day of measurements was between approximately 15 and 25 gigabytes in size. The entire data collection, which was captured in two primary collection efforts, was approximately 5 TB in size, and represented over 70 billion records and 45 billion ADUs in over 5 billion connections (see Section 3.3).

Chapter 4 introduces the performance anomaly detection methods that I use in Chapters 5 and 6. The methods need to be general (Section 4.1). Many statistical approaches rely on assumptions such as linearity and normality, but I cannot afford to make such assumptions.

The fundamental operation of my performance anomaly detection techniques is to quantify the dissimilarity between a distribution of response times and a set of distributions defining the normal or expected performance. Central to this operation is the issue of how a distribution, or a set of response time measurements, is represented (see Section 4.2). I have several goals in choosing a representation. First, the representation must work well even when the number and range of observations in the distribution vary widely. Second, I want a *compact* representation; that is, I want to minimize the space taken by the representation. Third, I want to minimize the amount of information lost by the representation; that is, I want to maximize the information preserved by the representation. Lastly, I want a representation for which the population of

distributions varies linearly. The last point is subtle, but the more linear the variation in some abstract space, the better my methods will work.

In the end, I settle on using the discrete quantile function (QF) representation. This representation maximizes information, requires little space, works well even when the number and range of distributions vary widely, and varies linearly in many circumstances. The QF is a b-bin vector, where b is a parameter. Each distribution forms a QF.

An important technique used by my methods is called principal components analysis, or PCA (Section 4.3). The input to PCA is a set of n b-bin QF vectors, which can be considered as n points in a b-dimensional space. Intuitively, PCA finds the directions of greatest variation among the points, and it is useful because it accounts for correlations among the bins of the QF vectors. Thus, for example, if a new distribution is consistently ten percent higher in all of its b QF bins, it is only penalized once, rather than b times.

The first k principal components (or directions of greatest variation) identified by PCA are used to form the normal basis. In addition, the $(b - k)$ remaining principal components are combined for form a single "error component". These $(k + 1)$ components form what I call the "normal basis" for a server. Intuitively, the normal basis for a server is its *profile* defining what is normal performance for the server. It also defines the extent to which normal performance varies, *e.g.* from weekday to weekend or between different seasons. Recall that the fundamental operation of my anomaly detection techniques is to determine how dissimilar a new response time distribution is compared to its past performance. This past performance is defined by the normal basis, and Section 4.4 discusses the technical details of how this comparison, called an "anomaly test" is performed. The output of the anomaly test is an anomaly score. Section 4.5 provides a tutorial of PCA and its role in the anomaly test.

One problem is that, as response time distributions are being collected to form the normal basis, there is no guarantee that all the distributions will reflect normal operation. There is the possibility that some of these distributions will be anomalous. Section 4.6 discusses this issue and offers a "bootstrapping" technique to cross-validate the input training set.

The entire set of anomaly detection methods are subject to a number of parameters. For example, how many bins should each QF have? Section 4.7 discusses the five parameters and explores the effect of each. Section 4.8 goes further, varying two parameters at a time to explore

the effects of jointly varying parameters.

So far, I have only been considering performance anomaly detection in terms of a set of discrete distributions. Another consideration is the elapsed duration of a distribution; in other words, how long should response times be aggregated to form a single distribution? Most of my methods form a distribution from a day of response time measurements. Section 4.9 discusses other potential aggregations with finer resolution, including a "rolling", window-based aggregation technique.

Section 4.10 introduces the "ordinal" analysis, used throughout Chapter 5. The idea is that, for a given server, there is a likely correlation between the first response times of connections, so it makes sense to consider all first response times as a group, all second response times as a group, *etc.*

In some cases, the traffic for a server shifts profoundly and permanently, resulting in continuous, unceasing anomalies. Section 4.11 discusses this problem and describes how the normal basis can be reset in such a case. Then, Chapter 4 closes with a summary and a discussion of the limitations of my performance anomaly detection approach.

I apply the performance anomaly detection methods to actual server performance data collected by `adudump` from the UNC network in Chapter 5. The chapter contains eight detailed case studies of performance anomalies that my methods detected. The goal is to show that the anomalies that my methods detect are legitimate anomalies reflecting real performance issues. Furthermore, I show that the data measured by `adudump` is useful in diagnosing the cause of these issues.

The first case study, in Section 5.2, discusses a teleconferencing server that routinely sees severe performance issues. The issues are always on a weekday, and the anomaly scores during the weekends are low. Based on data measured by `adudump`, I show that the anomaly score correlates well with various measures of load, including the number of connection attempts, the number of connections established, the number of ignored and rejected connections, the number of requests, and the number of open connections per hour. All of these load measurements can be inferred from the output of `adudump`. Because performance decreases so drastically for this server during periods of high load (causing severe anomalies), I conclude that the server becomes overwhelmed during the frequent periods of heavy use, and that this server should be upgraded.

The second case study, in Section 5.3, provides a compelling example of the usefulness of `adudump` data in diagnosing an issue, when combined with a knowledgeable domain expert. This server sees a sudden onset of severe performance anomalies. The anomalies last for nearly a month, and then suddenly cease. The number of rejected connections increases sharply at the end of the anomalous period, and almost all of the increase is because a single client IP address is being refused, and that client is sending connection requests very frequently. An ordinal analysis was used to discover that the second response was much longer during the anomalous period than during the normal period, and there was a strong mode at three seconds in the former. The server accepted connections on the POP3 port, and the second exchange in most POP3 dialogs is when the server makes an authentication decision. Modern Linux systems use PAM, or pluggable authentication modules, for authentication, and PAM is typically configured by default to pause for three seconds before responding that an authentication attempt failed, in order to frustrate password guessing attempts. Again, the combination of domain knowledge (which presumably the administrator of a server will have) and the data provided by `adudump` is powerful.

Section 5.4 analyzes a performance issue that was discovered by the UNC IT support group. An analysis of the structure of connections, as informed by `adudump` data, was again useful here, and it showed that the third response time in each connection was much longer during the anomaly than during the normal traffic. This case study also confirmed that the anomalies that my methods detect are actually of interest to a network manager.

The fourth case study, in Section 5.5, examines an issue that caused the most severe performance issues that I saw in the entire `adudump` data set. A campus SMTP server began having extremely poor performance. The administrators responsible for this server were able to provide information about the performance issue from there end, and they confirmed that it was severe and of interest to them. My diagnosis indicated that the fifth response time in each connection was the one causing the most problems, which in an SMTP connection typically corresponds with the email message. The administrators confirmed that the main problem involved an analysis of the email message. Thus, this case study confirmed several ways in which the diagnoses enabled by `adudump` data is useful.

The remaining case studies offer additional evidence for the usefulness of my performance

anomaly detection methods and the diagnostic power of `adudump` data, in particular the ordinal analysis. The last case study, in Section 5.9, provides an example of a limitation with the current methods related to the case in which there is a small number of response time observations, with extreme outliers possible. The discussion also proposes an augmentation to the methods that would address this limitation.

Many evaluations of anomaly (or intrusion) detection methods analyze the false positive rate, or the proportion of alarms that did not correspond to real anomalies (intrusions), and the false negative rate, or the proportion of real anomalies (intrusions) that went undiscovered by the methods. Such an analysis requires that all real anomalies (intrusions) are identified *a priori*. Unfortunately, there is no authoritative source that labels anomalies, partially because the definition of what issues are important to a network manager is an imprecise concept. However, I can still compare my performance anomaly detection methods with other approaches to discovering performance issues among servers, and that is what I do in Chapter 6.

The approach I compare my methods against is from an open-source network management tool called "Nagios". Nagios, which was deployed by the web systems group at UNC to monitor their web servers, sends a probe out to the server and measures the response. In the case of web servers, the probe is an actual HTTP request. If the response time is above a certain threshold, then Nagios sends a warning. If three successive response times are above the threshold, then Nagios sends a more urgent alert, indicating that there is a performance issue with the server.

I compare the set of performance issues identified by Nagios against the anomalies that my methods detect for the same servers. Unfortunately, the two sets do not overlap. I explain why each of the performance issues was not identified as a performance anomaly by my methods, and then I explain why each of the performance anomalies identified by my methods were not likewise identified as an issue by Nagios. I uncover several observations. First, it is very important to get the vantage point of `adudump` right, so that it is able to observe the majority of traffic for a given server, while still seeing a broad cross section of servers within a network. Second, the approach taken by Nagios is a simplistic one, resulting in many issues that have no identifiable cause. Third, I identify a situation in which the anomaly scores reported by my methods seem inordinately high. In general, the comparison study between my methods and Nagios shows that my methods are at least as good (and probably better) than some of the

tools already out there.

Chapter 2

Background and Related Work

This chapter is divided into two sections. Section 2.1 reviews the background of my work, including a discussion of the TCP sliding window mechanism and the A–B–T model. Section 2.2 reviews related research.

2.1 Background

2.1.1 The sliding window mechanism

I exploit the sliding window mechanism of the transmission control protocol (TCP) to infer application intent. Specifically, I observe the sequence and acknowledgement numbers in the header of a TCP segment to determine where the segment fits within the larger transmission. I discuss the inference algorithm in more detail in Chapter 3; this section reviews the sliding window itself.

TCP provides a mechanism to applications for reliable transport over a network, allowing them to transport data without worrying about network effects such as packet loss and reordering. The *sliding window* is the basic mechanism used to achieve this reliable transport.

The operation of TCP's sliding window is illustrated in Figure 2.1. The sender sends some data over the network. Upon receipt of the data, the receiver replies with an acknowledgement (ACK) packet indicating that it received the data. When the sender receives the ACK, it advances the sliding window. The window signifies the data that is currently in transit and unacknowledged. There are many related complexities of this scheme that I will not cover here, such as the effect of packet losses and the ability of the receiver to throttle the sender by

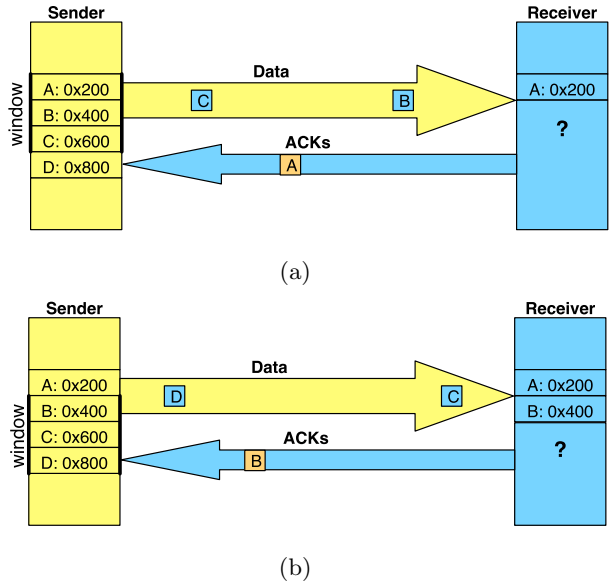


Figure 2.1: TCP steady-state operation. (a) shows the sequence numbers of the data, the packets in transit, and the state of the connection just before packets A (an acknowledgement) and B (data) are received. (b) shows the state of the network afterward; note that the sender's window has advanced.

adjusting the window size.

One requirement for correct operation of TCP is that both sides be able to determine where a particular packet belongs within the larger data stream even in the face of reordered packets and packet losses. Therefore, TCP labels the data with 32-bit *byte sequence numbers*. These sequence numbers enable the receiver to determine whether a packet contains new data (and, if so, where it belongs), and it also enables the sender to determine what data has been received (from the ACK packets).

Both hosts in a TCP connection are allowed to send data at the same time, and each hosts picks an initial sequence number (or ISN) at the beginning of the connection. These ISNs need not be different (although they almost certainly will be), because each direction (or *flow*) of traffic within the connection is independently governed by the TCP sliding window mechanism detailed above.

Thus, by observing the sequence numbers of packets as they travel across the network, one can infer how the application is using the network. For example, when a web server sends a 20 kilobyte image to a web browser, the observer will see a series of packets with sequence

numbers from the ISN to the ISN+20KB, and it will infer that the server intended to send a data object of size 20 KB. This higher-level summary of communicating application's behavior is the essence of the A–B–T model, which I introduce in the next section.

2.1.2 The A–B–T model

When considering the operation of a connection at the packet level, the complexities can be overwhelming. The *A–B–T model* [WAHC+06, HCJS07a] was created to abstract away the complexities and focus on the higher-level dialog. The model thus considers the arbitrarily large units of data that are exchanged in turn, along with the times in between such units, rather than the mechanics of how such a dialog occurs over the network. This abstraction is fundamental to my work.

Consider a spoken conversation between two people. Think of every ten word group as a unit of data. I can imagine counting the number of groups each person uttered and doing useful things with the counts. However, it would be better to group words into entire utterances, without artificially restricting the data unit size. Then, I could count the number and size of the larger utterances that make up the dialog. Furthermore, if each person politely waits for his turn to speak, I could find it useful to also measure the "think-time" between the end of one person's utterance and the beginning of the next person's utterance. In the same way, I want to remove network-level effects such as the maximum segment size (MSS), which are irrelevant to the applications using the network. I capture and measure the higher-level dialog between two hosts to unlock the many insights that are otherwise obscured by the details of network transmission.

The A–B–T model is named for its three components: A's are units of data sent from the connection initiator to the connection responder, or *requests*; B's are units of data sent in the opposite direction, or *responses*; and T's are the *think times* between the data units. The application-defined units of data in the A–B–T model are called ADUs (application data units). I will show in Chapter 3 that it is possible to build an A–B–T model for a TCP connection by passively observing the packets from an intermediate vantage point. It is possible to build models from traffic using any reliable transport protocol based on the sliding window mechanism, but TCP is the *de facto* standard and carries the vast majority of reliable, connection-oriented

traffic. It is possible to create approximate or probabilistic models from connectionless traffic or unreliable connections (*e.g.* UDP), but I cannot make strong claims about the think times using such models. Therefore, I focus on TCP.

It is important to distinguish server-side think times, or *response times*, from client-side think times. I carefully define a response time as the elapsed time following a request and preceding a response. Conversely, the less interesting client-side think time is the elapsed time following a response and preceding a request. The response time is central to my work because it is the primary metric for server performance.

The A–B–T model also accounts for connections in which one host pauses for a significant time while sending some data unit. I call such pauses *quiet times*, and they split a request (response) into multiple requests (responses). Quiet times are only reported if they exceed the intra-unit quiet time threshold, which defaults to five hundred milliseconds; otherwise, they are ignored. Five hundred milliseconds was chosen as a reasonable trade-off between a high level of abstraction and a high level of detail. Choosing a higher threshold means that important delay-based aspects of the application-level dialog will be missed; choosing a lower threshold means that I am likely to insert a quiet time in the dialog even when the application itself neither intended nor caused any delay. Note that, given my lack of definite information about application intent, I cannot get this right in every case. The threshold will be too high for some connections and too low for others. I have merely chosen a reasonable value.

Figure 2.2 shows two example *connection vectors*, or instances of an A–B–T model for two TCP connections. All aspects of the A–B–T model are included: requests (or A objects), responses (or B objects), response times, client-side think times, and quiet times. The figure shows how multiple segments may comprise a single application data unit.

The A–B–T model can model most types of TCP connections. However, there is a type of TCP connection that has a less structured dialog: both connected hosts "talk" in overlapping time intervals. Such *concurrent* connections can be easily captured by a generalization of the A–B–T model in which each flow is independently modeled as a sequence of data units and quiet times. However, independently modeling each flow loses the correlation between sequence numbers in one direction and acknowledgement numbers in the opposite direction, which is an essential aspect of inferring the application-level dialog from network traffic (since the acknowl-

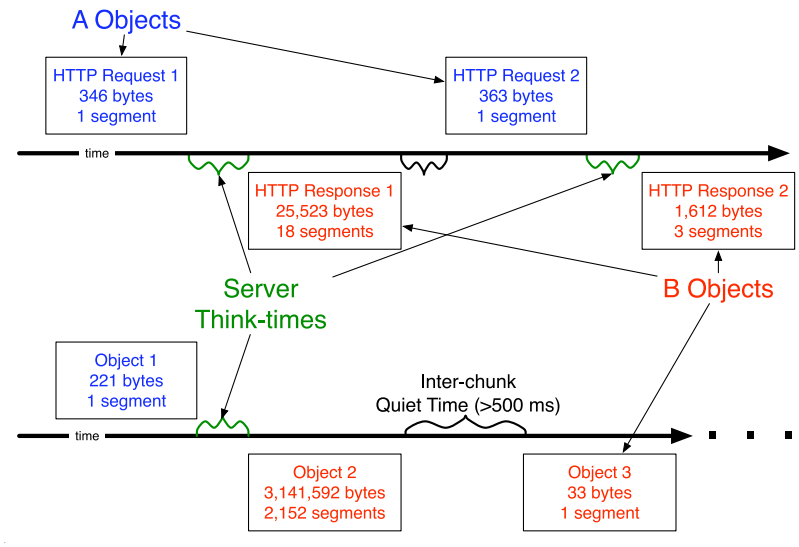


Figure 2.2: the A–B–T model, consisting of A-type objects, B-type objects, and think-times. The bottom connection shows two data units separated by an intra-unit quiet time.

edgement number increases only when the application has received the data). For this reason, I cannot make solid inferences about server performance from concurrent connections, so I focus on the think-times in non-concurrent, or *sequential*, connections.

2.2 Related Research

There are several research areas that are related to my work, and I have divided the discussion into three subsections. First, I will review research that uses passive network measurements to monitor some aspect of a network. Second, I will introduce the FCAPS model of network management, which is a taxonomy for the domain of problems related to network management. Then, I will discuss other network management tools and solutions. Lastly, I will discuss related research that models some aspect of connections.

2.2.1 Passive network measurement

One of the key aspects of my research is that the measurement is passive—that is, the measurement does not perturb the network or the services provided by the network. In this section, I review selected works that use passive measurement for similar ends, to monitor some aspect of a network.

Fraleigh *et al* [FDL+01] propose a design for a system to passively collect and store packet header traces from high-speed links within an Internet service provider's point-of-presence (PoP). The system can enable workload characterization, among other things. However, packet header traces offer a higher level of detail than I need and consume vast amounts of storage space. My work collects more insightful information in less space.

Hussain *et al* [HBP+05] describe a passive, continuous monitoring system to collect, archive, and analyze network data captured at a large ISP. They collect packet header traces as well as metadata, and they use vast disk resources to store months of data. My work is similar in principle, using a continuous, passive measurement approach to collect months of data; however, I use far fewer resources. Furthermore, the utility of my data compared to its size is much higher.

Malan and Jahanian's Windmill system [MJ98] also collects continuous passive measurements. Like my work, theirs is a *one-pass* approach to processing the traffic stream, and there is a focus on application-level effects. Unlike my work, however, they use incoming packets to trigger events in application protocol "replay" modules that they manually created for the most popular application protocols. This approach requires significant work to create each such module. Instead, I measure application-level phenomena in a generic fashion, requiring no knowledge of any particular application.

Feldmann's BLT [Fel00] is a method for capturing HTTP information, including request/response pairs. Like my work, it abstracts network and transport effects such as out-of-order delivery and retransmissions, instead focusing on the application-level data units exchanged between hosts. However, BLT focuses only on HTTP and gleans information from the TCP payload, so it is not appropriate for encrypted traffic. Furthermore, it uses a multi-pass algorithm, and so is not suitable for continuous monitoring of high-speed links, which require a one-pass approach. Another approach, based on BLT, that has similar drawbacks is Fu *et al.*'s EtE [FVCT02].

Olshefski *et al.* introduce ksniffer in [ONN04], which in turn is based on their CERTES system [ONA04]. This work is similar to mine in approach in that it passively infers application-level response times on a high-speed link (therefore using a one-pass algorithm). Their goal is to measure, at the server, the time from a client's request of a web page to the receipt of all data by the client. Like other related work, however, the focus is on HTTP, leveraging knowledge

of HTTP's functionality and requiring access to HTTP headers—which, again, is not possible in encrypted communication such as HTTPs. Furthermore, their work is not entirely passive, since the measurement occurs in the kernel of the server host and therefore steals cycles from the HTTP server application. Later work by Olshefski and Nieh on RLM [ON06] fixes this problem by performing the measurement on a separate machine; however, the machine is placed in-line with the HTTP server so that it can affect the traffic, so it also is not purely passive. My work is purely passive, generic for any sequential application protocol, and does not require access to TCP payloads, thus functioning equally well on HTTPs and HTTP (as well as any other sequential application protocol).

2.2.2 The FCAPS model of network management

The FCAPS model is the International Telecommunications Union model describing the space of network management tasks by dividing it into five layers [Udu96]: fault, configuration, accounting (or administration), performance, and security, which I will discuss in turn.

- **Fault** management is concerned with whether the individual network components—the routers, switches, and links—are up and running. If not, there is a *fault* that needs to be detected and addressed. This is typically achieved with the simple network management protocol, SNMP.

- **Configuration** management is concerned with achieving and maintaining a consistent configuration among all the devices in the network. For example, most networks will want to ensure that every side of every link is set to full-duplex (not half-duplex), and that the auto-negotiation of the link rate did not yield a lower rate than the link is capable of. Scaling these issues to a network with tens of thousands of end systems proves the need for an automatic approach to configuration management.

- **Accounting** involves measuring the traffic volume sent between routing domains. Most domains, or *autonomous systems* (AS's), have agreements with neighboring AS's about the amount and price of network traffic that the neighbor will accept for transit. For example, UNC is charged by (one of) its upstream provider(s) to the Internet according to the 95^{th} percentile of the link utilization each month, so that the more traffic UNC

sends and receives to this AS, the higher the price the AS will charge for its service. Other AS's have peering arrangements, so that transit is free as long as the service is mutual (and roughly similar in volume). In any event, accounting involves measuring the traffic volume sent to and from neighboring AS's and sometimes enforcing outgoing rate limits to keep costs down. Another area in the accounting domain is billing users (such as departments on campus) according to their usage.

- **Performance** management is concerned with the same objects as fault management, but the concern is orthogonal. Instead of asking whether the objects are up and running, performance management assumes they are running and asks whether they are running *well*. This layer describes my research.

- **Security** management is concerned with keeping the network secure from attack. For example, the routers and switches should not accept configuration changes unless the change is appropriately authenticated and authorized. Also, network users should not be able to spy on the network transmissions of other users.

These are all areas classically considered part of the domain of "network management". I argue that a natural extension of this domain should include managing the performance of authorized services in the network, such as web services and email services. This extension does not comprise an additional layer of the FCAPS model; rather, it is an extension of the fault and performance layers. The fault layer is extended to detect faults not just of the internal network devices, but also of the servers attached to the network. Likewise, the performance layer is extended to manage the performance not just of routers, switches, and links, but also the servers.

I assert that the more interesting challenges arise not in server fault management but in managing the *performance* of these servers. First, how does one measure performance? Second, how does one define what is good performance? Third, since the servers provide a range of different services, should the definition of "good" performance be variable depending on the service? Fourth, must we have access to the internal workings of the server's operating system (*e.g.* the CPU utilization) in order to measure performance, or is a "hands-off" approach satisfactory? Fifth, is there a way to automatically discover new and existing servers? (This

is an important consideration in a hierarchically organized enterprise environment such as UNC.) Each of these questions raises interesting issues with the task of server performance management, and the solution that I propose will address all of them. It is important that the solution be well-designed. It needs to work not just in small networks, but in large, enterprise-level networks such as UNC's, with tens of thousands of hosts and hundreds of servers.

2.2.3 Network management solutions

One of the more commonly used network management tools is Flowscan [Plo00]. Flowscan reads Cisco Netflow[1] records (obtained through passive measurement), which contain byte and packet counts per flow, and aggregates them into simple volume metrics of byte, packet, and flow counts per five-minute interval. The timeseries of these counts are stored in a round-robin database (using RRDtool[2]), which automatically averages older points to save space. The goal is to understand the operation of the network—the same as my goal. However, I expect my approach to unlock deeper insights. Netflow records do not take internal flow dynamics into account, and they have no notion of anything like a quiet time. Furthermore, it does not correlate the two flows in a connection, so it cannot support the idea of a think time or a response time.

Estan *et al*'s AutoFocus [ESV03] tool (and the similar Aguri [CKK01] tool by Cho *et al*) is motivated by a similar problem as my work: packet header traces provide too much information, overwhelming the network manager with often irrelevant data, so a better summary is needed. Their approach is to cleverly summarize the network traffic into a list of the most significant patterns, for some definition of "significant". For example, a flood of SYN packets to a particular host would show up as an anomaly when viewing the number of flows per destination host. Thus, the task of AutoFocus is to look at traffic from various viewpoints and carefully aggregate on one or several dimensions to come up with the list most useful to the network manager. My work is a good complement to AutoFocus because it combines insightful traffic analysis with a history and an analysis of trends per server. Also, my approach is capable of reporting in an online fashion (i.e. from a single, quick pass of a stream of packet headers), rather than

[1] http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html
[2] http://oss.oetiker.ch/rrdtool/

requiring an offline analysis like AutoFocus.

Bahl *et al*'s Sherlock system [BCG+07] attempts to localize the source of performance problems in large enterprise networks. They introduce an "inference graph" to model the dependencies in the network, providing algorithms for inferring and building these models automatically. The system is able to detect and diagnose performance problems in the network down to their root cause. However, the system requires deploying "agents" on multiple machines at multiple points in the network, so deployment is not simple. Also, the agents inject traceroutes into the network and make requests to servers, so the system is not passive.

Gilbert *et al* take a different approach [GKMS01]. They sacrifice exactness for the ability to produce meaningful summaries from high-speed traffic as it streams by. They explain: "Network engineers and researchers have three basic methods for windowing the measurement data: filtering, aggregation, and sampling. I propose a fourth method: sketching." My approach avoids the need to sacrifice exactness because it is designed to be scalable with the processing power of the monitoring hardware.

Lastly, there is a closely related network management solution called OPNET ACE[3]. ACE is similarly concerned with network service performance, and it is also interested in application response times. However, ACE is a much heavier solution. It is capable of more precisely distinguishing between network transit time and application processing time, but it requires an extensive deployment of measurement infrastructure throughout the network, on clients, servers, and points in between. Furthermore, ACE constitutes an expensive, *active* measurement approach, unlike my inexpensive, passive one.

2.2.4 Network connection modeling

The idea of user-level think-time modeling was introduced by Barford and Crovella [BC98], who used the idea used to realistically model HTTP users. Lan and Heidemann extended this idea to empirically-derived distributions of think-times in [LH02]. I extend the idea to an *application-level* think-time, which functions the same whether the think-time is due to a user thinking or an application processing some input. This idea is the basis of my server-side *response time* metric, as introduced in Chapter 3.

[3] http://www.opnet.com/solutions/application_performance/ace.html

A seminal work from Smith *et al* introduced the notion of inferring application behavior from transport-level information alone [SHCJ01]. This work eventually led to the A–B–T model and the "t-mix" traffic generator [HC06, HCJS07b, WAHC+06]. I use the A–B–T model for a different purpose, and, unlike "t-mix", I can build instances of the A–B–T model in an online fashion using a one-pass approach.

Vishwanath and Vahdat also create a general model and use it for traffic generation [VV06]. Like "t-mix" and my work, they model TCP connections as request/response pairs, with data unit sizes and "think-time" latencies. However, they take a different approach to modeling for the purpose of generating or replaying traffic, capturing information at higher "session" and request/response "exchange" layers.

Chapter 3

Real-time passive performance measurement

This chapter introduces `adudump`, the passive network measurement tool and modeler upon which the performance management methods are based. I first discuss the specific approach I have taken in Section 3.1. I then discuss the validation of `adudump` in Section 3.2 and, lastly, the data I collected in Section 3.3.

3.1 Approach

This section details the approach I have taken to solve the problem of quantifying server performance. In general, I name this problem *A–B–T inference*. The A–B–T connection vector model, comprised of ADUs (or application data units), "think times", and quiet times, was introduced in Section 2.1.2. First, Section 3.1.1 discusses a specific measurement context in terms of the UNC network context, the monitoring vantage point, and the specific setup. Section 3.1.2 gives a high-level overview of the one-pass A–B–T inference algorithm used by `adudump`, and Sections 3.1.3 and 3.1.4 detail the data structures and algorithms of `adudump`. Lastly, I discuss the limitations of a one-pass approach in Section 3.1.5.

3.1.1 Measurement setup at UNC

In order to fully understand `adudump`, it is helpful to consider the assumptions that are made about its context and measurement environment. The **monitor** is the machine on which `adudump` executes. Figure 3.1 illustrates that the monitor receives a copy of all packets that

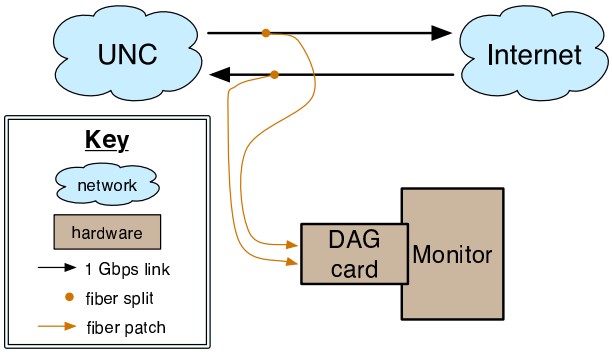


Figure 3.1: Placement of the monitor with respect to the UNC network

flow between the Internet and the UNC network.[1] The monitored links are 1 Gbps Ethernet carried over fiber-optic cables, which are tapped with optical splitters, so that the monitor gets a copy of each packet. Note that this enables passive measurement; if the monitor were inline, it would delay the delivery of each packet.

The placement of the monitor, also called its **vantage point**, provides a set of convenient labels, which we will refer to throughout this discussion. A 1 Gbps Ethernet link is carried on two fiber-optic cables, one for each direction of frame transmission on the full-duplex link. Frames transmitted on the fiber optic cable providing a path from the Internet to UNC are called **inbound**, whereas the other path is **outbound**. The destination address (port) of inbound traffic (or the source address of outbound traffic) is called a **local** address, whereas the other address (port) is called a **remote** address (port).

Connections can also be classified as either inbound or outbound, depending on the direction of the connection request (initial SYN packet). Connection requests from an external client to a local (UNC) server are inbound. I only consider inbound connections in this dissertation because I am managing the performance of UNC servers, not external servers, and I am concerned with response times for the requests of external clients.

The monitor itself is a machine running FreeBSD 5.4. It has a 1.4 GHz Xeon processor, 1.25 GB of RAM, and an Endace DAG 4.5 G4 packet capture card. The DAG card accepts fiber optic input and provides frames and precise timestamps to applications (such as `adudump`) without significant processor overhead, leaving the processor free to service the application.

[1] Actually, the monitored link is one of several that serve UNC. This one carries commodity Internet only. Traffic involving educational institutions (carried over Internet2) is not included.

The current processor hardware is quickly becoming dated, yet it provides enough power to enable `adudump` to keep up with the links that routinely carry 600 Mbps of traffic (with spikes up to 1 Gbps), with no packet loss. This is a testament to the efficiency of `adudump`, which we consider next. We expect `adudump` to scale to much heavier link loads even on a 10 Gbps link with modern faster, multicore processors.

3.1.2 Introduction to a one-pass approach to constructing connection vectors

In previous work by Hernandez-Campos and others [HCJS07a], A–B–T connection vectors were constructed in an offline fashion. First, packet header traces were captured to disk. Then, a textual representation of the traces was sorted by time within a TCP connection, so that all packets from a particular connection were contiguous and time sorted. Finally, the sorted trace was analyzed to construct the connection vectors. Because only one connection was analyzed at a time, such an approach afforded the algorithm access to the entire packet arrival history of the connection. Storing packet header traces to disk currently requires approximately 30 GB per hour, whereas my approach only uses about 1 GB per hour.

My approach, as implemented in the `adudump` tool, is similar in many respects; however, because I need a continuous monitoring approach, I cannot analyze traces offline, nor can I require that the input be sorted by connection. This has profound implications, because I must keep track of thousands or potentially millions of active connections. Therefore, the amount of state per connection must be minimized, and it must not be dependent on the number of packets seen in the connection thus far.

Before I describe the details of the one-pass inference algorithm, I will provide an example and give some higher-level intuition for its operation. Consider Figure 3.2, which shows the packets exchanged in a TCP connection and the inferences made by `adudump`. The client first sends a SYN segment to the server according to the specification of TCP. `adudump` (running on the monitor) records this event with a `SYN` record, and saves the corresponding packet arrival timestamp. The server responds with a SYN/ACK segment, at which point *adudump* prints the measured round-trip-time (from the monitor's perspective) in an `RTT` record. When the client responds with an ACK packet, the TCP three-way handshake is completed, and `adudump` reports this fact with a `SEQ` record, which means "start of a sequential connection". All of

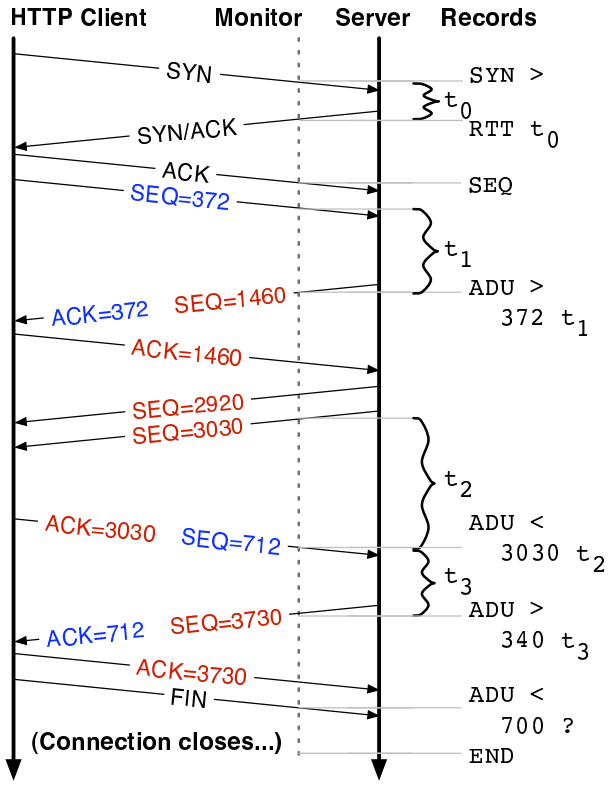


Figure 3.2: One-pass A–B–T connection vector inference example. The details of the connection close have been omitted.

these records provide contextual information and are not part of the actual A–B–T inference mechanisms.

Next, the client sends 372 bytes to the server, which we will label a request merely because it flows from the client to the server (and not because we leverage HTTP semantics). At this point, `adudump` merely keeps internal state about the size of the ADU and the time at which it was sent, opting not to report anything because it cannot determine with certainty that the client is finished sending data for now. When `adudump` sees the server respond with its own ADU, acknowledging (with the acknowledgement number of TCP) all of the request, it then reports the size of the client's request ADU along with the server's "think time". The "think time" reflects the amount of time it took the server to process the client's request and generate a suitable response. As mentioned in Section 2.1.2, I call think times that occur on the server side **response times**, and these times are the primary metric I use to quantify server performance.

The main body of the connection follows. At a high level, the operation of `adudump` for a sequential connection such as this one consists of (1) printing records as soon as possible (but no sooner), and (2) keeping track of sequence numbers, acknowledgement numbers, starting points of requests and responses, and the timestamp of the latest segment. I will define the operation of `adudump` more rigorously in the subsections that follow.

The response time is a crucial metric for measuring server performance, and it is useful to consider this metric in more detail. One weakness of the metric, as measured by `adudump` running on the monitor, is that it not only includes the amount of time taken by the server to receive, process, and respond to the request, but it also includes a component of network delay in between the server and the monitor, or the time taken to route and deliver a packet to its destination. This is unavoidable in a purely passive solution.

Furthermore, the network delay component for a response time in the UNC monitoring setup is negligible compared to any response times that might be indicative of a performance problem. Figure 3.3 shows the distribution of RTT (round-trip-time) measurements of the TCP three-way handshake for inbound connections over the course of a 22-minute trace. Over 98 percent of RTTs are less than 5 milliseconds, whereas the response times I am interested in are typically a hundred milliseconds or more. In summary, although the conflation of network delay and server processing time is an unfortunate consequence of the passive measurement

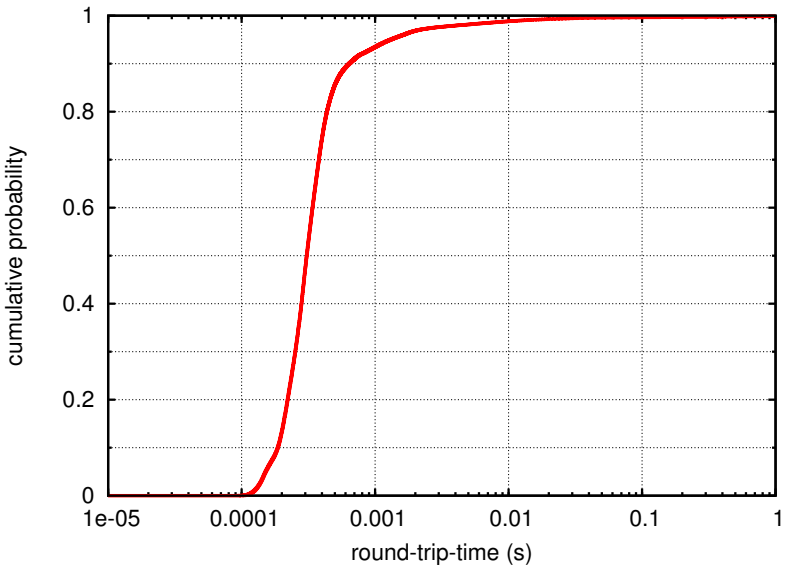


Figure 3.3: Round-trip-time measurements of the TCP handshake for inbound connections.

approach I take, it does not pose significant problems for a monitor vantage point with small network latency between it and the server.

An additional advantage of the response time metric is that it includes "top-to-bottom" processing; that is, every aspect of the server's interaction with the request is included in the response time, including the reception of packets at the network interface card, the scheduling of the server application, the execution of the server application, the packetization of outgoing packets, and the queuing delay of sending the packet out. In comparison, most approaches that measure response times from the same machine (including the server's own logging capabilities) will neglect all but the execution of the server application. Thus, the passively measured response time metric represents a more complete picture of the overall response.

3.1.3 Data structures used in A–B–T inference

Before we consider the algorithms of `adudump` in more detail, I will introduce the data structures manipulated by these algorithms. At a high level, there are two primary data structures: the current segment and the connection state table.

The current segment record, which I refer to as S, contains information from the TCP segment currently being analyzed. There is only one instance of the current segment record. The structure of the record is shown in Table 3.1. The segment record and the connection

Table 3.1: essential state for a segment record in `adudump`

Symbol	Mnemonic	Type	Description
T_S	`arrival_time`	timestamp	time at which the segment was captured
K	`connection_id`	composite "key" 4-tuple	the local and remote address and port number, as shown in Table 3.2.
N	`sequence_num`	unsigned 4-byte integer	the sequence number of the segment
L	`payload_size`	unsigned 2-byte integer	number of bytes of application data in the segment
N_L	`next_seq_num`	unsigned 4-byte integer	the last sequence number covered by the segment (*i.e.* $N_L = N + L$)
A_0	`ack_num`	unsigned 4-byte integer	the acknowledgement number of the segment
D	`direction`	enumeration	either `IN` (*i.e.* inbound) or `OUT` (*i.e.* outbound)
F	`tcp_flags`	set	contains zero or more of: `S` (SYN), `A` (ACK), `F` (FIN), and `R` (RST)

Table 3.2: connection identifier structure

Symbol	Mnemonic	Type	Description
I_L	`local_ip_addr`	Internet address	the local (i.e. UNC-side) IPv4 address (See Section 3.1.1.)
I_R	`remote_ip_addr`	Internet address	the remote (i.e. Internet-side) IPv4 address
P_L	`local_port`	unsigned 2-byte integer	the local port number
P_R	`remote_port`	unsigned 2-byte integer	the remote port number

Table 3.3: essential per-connection state in `adudump`

Symbol	Mnemonic	Type	Description
S_C	`connection_state`	enumeration	either `NEW_SYN`, `NEW_SYNACK`, `HALF_OPEN`, `MOSTLY_OPEN`, `SEQUENTIAL`, or `CONCURRENT` (See Section 3.1.4.)
K	`connection_id`	composite: "key"	the local/remote address/port number, as shown in Table 3.2.
F_I	`inbound_flow`	flow	the inbound flow state
F_O	`outbound_flow`	flow	the outbound flow state

Table 3.4: essential per-flow state in `adudump`

Symbol	Mnemonic	Type	Description
S_F	`flow_state`	enumeration	either `INACTIVE`, `ACTIVE`, or `CLOSED` (See Section 3.1.4.)
N_0	`initial_seq_num`	unsigned 4-byte integer	the initial sequence number seen in the flow (set during the 3-way handshake)
N_N	`next_seq_num`	unsigned 4-byte integer	the next expected sequence number
N_A	`adu_seq_num`	unsigned 4-byte integer	sequence number of the start of the last request or response
A_0	`initial_ack_num`	unsigned 4-byte integer	the initial acknowledgement number seen in the flow (set during the 3-way handshake)
A_H	`highest_ack_num`	unsigned 4-byte integer	the highest acknowledgement number seen so far
T_L	`last_send`	timestamp	last time a new segment was seen in this direction

record both use the connection identifier 4-tuple as the key, which is shown in Table 3.2.

The connection state table, which I refer to as T, contains a connection record for each active connection. T is implemented as a chained hash table, keyed by the connection identifier, with 250,000 buckets and three preallocated buckets per chain, and using the hashing functions provided by Bob Jenkins' "lookup3.c" library.

The structure of the records contained in T is shown in Table 3.3. Each connection record contains two flow records, F_I and F_O, for the inbound and outbound flows, respectively. (Recall that a bidirectional connection contains two unidirectional flows.) The structure of a flow record is shown in Table 3.4.

3.1.4 One-pass algorithms for A–B–T inference

The previous subsections introduced the one-pass approach to A–B–T inference and discussed the associated data structures. This section describes the algorithm of `adudump` in more detail by considering how the connection-tracking state changes over the lifetime of a connection.

The entire algorithm is driven by packet arrival events. When a TCP packet arrives, `adudump` checks the corresponding connection state (if it exists) and determines whether the segment should update the state. So, let us first consider what happens when a packet arrives. First, the DAG card receives the packet and attaches metadata to the packet, including a timestamp and the interface on which the packet was observed. The interface number is what enables `adudump` to distinguish inbound from outbound traffic, since each link is connected to a distinct DAG interface. `adudump` interfaces with the DAG card through the CoralReef API, which enables it to work equally well on other types of input, including offline traces in pcap format.[2]

Only certain types of packets are processed. The DAG card provides to `adudump` every packet received. Non-IPv4 packets are discarded, as are non-TCP packets. Only IPv4, TCP segments are considered for processing. Also, if there is no existing state for the connection of the segment, the segment is only processed if it has enough information to initialize the connection-tracking state without ambiguity—that is, if the segment is either a SYN or SYN/ACK segment. Furthermore, because I am interested only in the performance of UNC servers, and because servers by definition *accept* rather than *initiate* connections, I only consider connections that were initiated by a remote host (that is, the initial SYN segment was inbound, and the SYN/ACK segment was outbound). These and other behaviors are run-time options of `adudump`.

I will begin the discussion of `adudump`'s algorithms by considering what happens at the beginning of a connection, when the segment is a SYN segment and there is no state for the connection. See Algorithm 1, which shows how the connection and flow state is initialized. Because I am only tracking incoming connections, I only care about inbound SYN segments (the second and third preconditions). The `newConnection` function on line 1 adds a new connection with key $S.K$ to the table T of connection states. Then, the notation $T[S.K]$ in

[2]There is no interface number information in a pcap trace; `adudump` can also distinguish inbound from outbound traffic by means of a user-specified local network range, *e.g.* `192.168.10.0/24`.

Algorithm 1 First SYN segment of a connection

Require: $S.K \notin T$
Require: $S.F = \{S\}$
Require: $S.D =$ `IN`
1: `newConnection`$(T,\ S.K)$
2: $T[S.K].F_I.N_0 \leftarrow S.N$
3: $T[S.K].F_I.N_N \leftarrow S.N + 1$
4: $T[S.K].F_I.N_A \leftarrow S.N$
5: $T[S.K].F_I.A_0 \leftarrow 0$
6: $T[S.K].F_I.A_H \leftarrow 0$
7: $T[S.K].F_I.S_F \leftarrow$ `INACTIVE`
8: $T[S.K].F_I.T_L \leftarrow S.T_S$
9: $T[S.K].S_C \leftarrow$ `HALF_OPEN`
10: `reportSYN`$(S.T_S,\ S.K,\ S.D)$

lines 2–9 refers to the connection with key $S.K$ in table T. In TCP semantics, the SYN segment advances the window by one byte even though it has no application payload, so $F_I.N_N$, the next expected sequence number for the inbound flow, is initialized to one more than the segment's sequence number (line 3). The inbound flow is marked as `INACTIVE` (line 7), which means there is not currently an ADU being sent. The connection state is set to `HALF_OPEN` (line 9), which indicates that we have seen the first part (*i.e.* the SYN segment) of the TCP connection establishment protocol, called the TCP three-way handshake. Lastly, the `reportSYN` function (line 10) records the fact that `adudump` established state to track the connection. Note that $F_I.A_0$ and $F_I.A_H$ are initialized to zero (lines 5 and 6) because the initial SYN segment does not (and cannot) acknowledge any previous transmissions.

When the second part of the three-way handshake (the SYN-ACK segment) is received, Algorithm 2 is executed. Precondition 4 is there because `adudump` only tracks connections for which the server (or connection acceptor) is in the UNC network, which means that the SYN-ACK segment is outbound. Much of the state of the outbound flow is initialized as the inbound flow was in Algorithm 1 (lines 1-3, 6, 7). The connection state is advanced to `MOSTLY_OPEN` (line 8). According to the TCP protocol, the SYN-ACK segment acknowledges the initial sequence number (or ISN) set in the SYN segment. When the SYN-ACK segment is lost, the connection acceptor will sometimes receive multiple SYN segments from the connection initiator with different ISN's. For this reason, it is important to update the sequence numbers of the inbound flow (lines 9-11). Lastly, we report the RTT, or round-trip-time, as measured from the

Algorithm 2 SYN-ACK segment in a HALF_OPEN connection

Require: $S.K \in T$
Require: $T[S.K].S_C = \texttt{HALF\_OPEN}$
Require: $S.F = \{S, A\}$
Require: $S.D = \texttt{OUT}$
1: $T[S.K].F_O.N_0 \leftarrow S.N$
2: $T[S.K].F_O.N_N \leftarrow S.N + 1$
3: $T[S.K].F_O.N_A \leftarrow S.N$
4: $T[S.K].F_O.A_0 \leftarrow S.A$
5: $T[S.K].F_O.A_H \leftarrow S.A$
6: $T[S.K].F_O.S_F \leftarrow \texttt{INACTIVE}$
7: $T[S.K].F_O.T_L \leftarrow S.T_S$
8: $T[S.K].S_C \leftarrow \texttt{MOSTLY\_OPEN}$
9: $T[S.K].F_I.N_0 \leftarrow S.A - 1$
10: $T[S.K].F_I.N_N \leftarrow S.A$
11: $T[S.K].F_I.N_A \leftarrow S.A - 1$
12: $\texttt{reportRTT}(S.T_S,\ S.K,\ S.D,\ S.T_S - T[S.K].F_I.T_L)$

perspective of the monitor (line 12).

When the TCP three-way handshake is completed with the empty ACK segment, thus establishing the connection, Algorithm 3 is executed. Empty ACK segments abound in TCP connections, so preconditions 2 and 6 ensure that the connection context is correct. Just as the SYN-ACK segment acknowledges the connection initiator's choice of ISN, the ACK acknowledges the connection acceptor's ISN, so lines 3–5 update the acceptor's sequence numbers. The connection is advanced to the `SEQUENTIAL` state (line 7), and the connection establishment is reported with a `SEQ` record (line 8). Recall that there are two types of connections in the A–B–T model: sequential and concurrent. Connections in `adudump` are sequential by default, and a connection is only marked as concurrent if there is evidence for concurrency, which I discuss below.

Algorithm 4 shows what happens when a data segment is seen in the context of an `INACTIVE` flow of a sequential connection. The algorithm is for an inbound data segment only; the outbound case is similar. Data segments by definition contain some amount of application payload, hence Precondition 3. Pure ACKs (*i.e.* segments without a payload) are redundant information for the problem of A–B–T inference and can be safely ignored. Precondition 4 states that the algorithm is only executed if none of the SYN, FIN, or RST flags are set for the data segment. A data segment with a SYN flag is an error, and the FIN and RST cases are

Algorithm 3 Connection establishment in a MOSTLY_OPEN connection

Require: $S.K \in T$
Require: $T[S.K].S_C = \texttt{MOSTLY\_OPEN}$
Require: $S.F = \{A\}$
Require: $S.D = \texttt{IN}$
Require: $S.L = 0$
Require: $T[S.K].F_I.A_H = 0$
1: $T[S.K].F_I.A_0 \leftarrow S.A$
2: $T[S.K].F_I.A_H \leftarrow S.A$
3: $T[S.K].F_O.N_0 \leftarrow S.A - 1$
4: $T[S.K].F_O.N_N \leftarrow S.A$
5: $T[S.K].F_O.N_A \leftarrow S.A - 1$
6: $T[S.K].F_I.T_L \leftarrow S.T_S$
7: $T[S.K].S_C \leftarrow \texttt{SEQUENTIAL}$
8: $\texttt{reportSEQ}(S.T_S,\ S.K,\ S.D)$

Algorithm 4 Inbound data segment arrives in a `SEQUENTIAL` connection and an `INACTIVE` flow. The outbound case is similar.

Require: $S.K \in T$
Require: $T[S.K].S_C = \texttt{SEQUENTIAL}$
Require: $S.L \neq 0$
Require: $S.F \cap \{S, F, R\} = \emptyset$
Require: $T[S.K].F_I.S_F = \texttt{INACTIVE}$
Require: $S.D = \texttt{IN}$
Require: $S.N_L >_w T[S.K].F_I.N_N$
// lines 1–9 determine what to set the inbound flow's N_A to
1: **if** $T[S.K].F_I.N_N = T[S.K].F_O.A_H \wedge S.N >_w T[S.K].F_I.N_N$ **then**
2: **if** $|T[S.K].F_O.A_H - S.N|_w > \alpha \cdot MSS$ **then**
3: $T[S.K].F_I.N_A \leftarrow S.N$
4: **else**
5: $T[S.K].F_I.N_A \leftarrow T[S.K].F_O.A_H$
6: **end if**
7: **else**
8: $T[S.K].F_I.N_A \leftarrow T[S.K].F_I.N_N$
9: **end if**
10: $T[S.K].F_I.S_F \leftarrow \texttt{ACTIVE}$
11: $T[S.K].F_I.N_N \leftarrow S.N_L$
12: $T[S.K].F_I.A_H \leftarrow S.A$
13: $T[S.K].F_I.T_L \leftarrow S.T_S$
14: **if** $T[S.K].F_O.N_N >_w S.A \wedge S.N >_w T[S.K].F_O.A_H$ **then**
15: $T[S.K].S_C \leftarrow \texttt{CONCURRENT}$
16: $\texttt{reportCONC}(S.T_S, S.K)$
17: **else if** $T[S.K].F_O.S_F = \texttt{ACTIVE}$ **then**
18: $T[S.K].F_O.S_F \leftarrow \texttt{INACTIVE}$
19: $\texttt{reportADU}(S.T_S,\ S.K,\ \sim S.D,\ T[S.K].F_O.N_N -_w T[S.K].F_O.N_A,$
$T[S.K].S_C,\ S.T_S - T[S.K].F_O.T_L)$
20: **end if**

considered in Algorithms 10–13. Finally, the last precondition ensures that the segment is not entirely a retransmission. Partial retransmissions (*i.e.* segments with both already-transmitted data and new data) will be handled by the algorithm. Note that the $>_w$ operator indicates a greater-than operation that respects the boundaries of TCP's wrapped window; that is, the $>_w$ operator takes into account the case in which the 32-bit sequence number value wraps from the highest value to the lowest. Retransmissions still update the flow's T_L timestamp, however.

Now, consider the body of Algorithm 4. As the comment states, lines 1–9 are concerned with the setting of $T[S.K].F_I.N_A$, which marks the sequence number of the first byte of application data sent in the ADU started by this segment. Line 1 tests for an out-of-order segment; that is, perhaps the first segment of the ADU was lost, and the current segment is actually the second segment. In the case of an out-of-order segment (line 1), N_A is set to the highest acknowledgement number seen in the outbound flow (line 5), unless that number is more than $\alpha \cdot MSS$ away from the current segment's sequence number $S.N$ (line 2), in which case N_A is set to $S.N$ (line 3). The MSS is TCP's maximum segment size, which I assume to be 1,460 bytes. ($||_w$, like $>_w$, respects window boundaries.) This special case seems unlikely, but it is unfortunately necessary, because, in rare cases, two (broken) TCP implementations will not agree on their respective ISNs during the handshake, which can result in spurious ADU sizes. α controls the extent to which gaps in the sequence number stream are tolerated. Too small a setting will result in legitimate ADUs exhibiting a sequence number gap being ignored, and too large a setting will result in spurious (and potentially large) ADUs being reported; I chose 5 as a reasonable trade-off between these cases.

Line 10 reflects the fact that the inbound flow is now considered `ACTIVE`, and lines 11–13 are a standard state update. Line 14 tests for concurrency, which is detected if there is unacknowledged data in both flows, in which case `adudump` updates the connection state and reports the concurrency detection event. Lastly, line 17 checks for the existence of an ADU in progress on the outbound flow. For sequential connections, the outbound flow is finished with its ADU when the inbound flow begins sending data (and vice versa). In that case, the outbound flow is marked as inactive, and the now complete outbound ADU is reported, including its direction (where $\sim$ indicates the opposite direction), size (where $-_w$ also respects window boundaries), the type of connection (*i.e.* `SEQUENTIAL` or `CONCURRENT`) from which the

Algorithm 5 The `checkQuietTime`($T,\ S.K,\ \tau$) algorithm

Require: $S.K \in T$
Require: $T[S.K].S_C = \texttt{SEQUENTIAL}$
1: **if** $S.T_S - T[S.K].F_I.T_L > \tau$ **then**
2: `reportADU`($S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,$
$T[S.K].S_C,\ S.T_S - T[S.K].F_I.T_L$)
3: $T[S.K].F_I.N_A = T[S.K].F_I.N_N$
4: **end if**

Algorithm 6 The `updateSeqNum`($T,\ S.K,\ \rho$) algorithm

Require: $S.K \in T$
Require: $T[S.K].S_C = \texttt{SEQUENTIAL}$
1: **if** $S.N_L >_w T[S.K].F_I.N_N \ \wedge\ |T[S.K].F_I.N_N - S.N_L|_w \leq \rho \cdot MSS$ **then**
2: $T[S.K].F_I.N_N \leftarrow S.N_L$
3: **end if**

ADU was seen, and its "think time".

The `checkQuietTime`($T,\ S.K,\ \tau$) algorithm is listed in Algorithm 5, and it will be referred to frequently in later algorithms. The condition in line 1 is true when the current segment arrived at least τ seconds from the flow's T_L timestamp. This case is called a "quiet time", as discussed in Section 2.1.2, and is controlled by run-time parameter τ, the intra-ADU quiet time threshold parameter, which defaults to 500 milliseconds. In the case of a quiet time, the ADU in progress is reported as finished (line 2), and a new ADU is begun (line 3).

The `updateSeqNum`($T,\ S.K,\ \rho$) algorithm, listed in Algorithm 6, is another common algorithm that will be used frequently. It updates the current flow's next sequence number if the current segment is in order, within a certain tolerance. The tolerance is controlled by the parameter ρ (*e.g.* α), which is maximum size of an acceptable gap, in units of the maximum segment size, or MSS.

When an inbound data segment is seen in the context of an ACTIVE flow of a sequential connection, Algorithm 7 is executed. (The outbound case is similar.) As in Algorithm 4, the last precondition ensures that only data segments with at least some new data (even if some data is also old) are considered—although, as before, if this is the only precondition not met, the flow's T_L timestamp is still updated. Line 1 checks the quiet time condition, as discussed above. Line 2 updates the flow's N_N state variable conditional on β. I found $\beta = 15$ to be a good choice that balances the probability of missing data (when β consecutive segments are lost

Algorithm 7 Inbound data segment arrives in a `SEQUENTIAL` connection and an `ACTIVE` flow. The outbound case is similar.

Require: $S.K \in T$
Require: $T[S.K].S_C =$ `SEQUENTIAL`
Require: $S.L \neq 0$
Require: $S.F \cap \{S, F, R\} = \emptyset$
Require: $T[S.K].F_I.S_F =$ `ACTIVE`
Require: $S.D =$ `IN`
Require: $S.N_L >_w T[S.K].F_I.N_N$
1: `checkQuietTime`$(T,\ S.K,\ \tau)$
2: `updateSeqNum`$(T,\ S,\ \beta)$
3: $T[S.K].F_I.A_H \leftarrow S.A$
4: $T[S.K].F_I.T_L \leftarrow S.T_S$

Algorithm 8 Inbound data segment arrives in a `CONCURRENT` connection. The outbound case is similar.

Require: $S.K \in T$
Require: $T[S.K].S_C =$ `CONCURRENT`
Require: $S.L \neq 0$
Require: $S.F \cap \{S, F, R\} = \emptyset$
Require: $S.D =$ `IN`
Require: $S.N_L >_w T[S.K].F_I.N_N$
1: `checkQuietTime`$(T,\ S.K,\ \tau)$
2: $T[S.K].F_I.T_L \leftarrow S.T_S$
3: `updateSeqNum`$(T,\ S,\ \beta)$
4: $T[S.K].F_I.A_H \leftarrow S.A$

and the next is delivered) with the probability of a random sequence number being accidentally within the range (approximately $\beta * MSS/2^{32}$, or about one in 200,000 when β is 15). It might be acceptable to use a single parameter for both α and β, since they control similar situations. I have not done an in-depth study of this issue, but I merely changed the parameters during the debugging and validation phase as I encountered interesting connections—and in any case, this is a rare issue affecting few connections or ADUs. Lastly, lines 3 and 4 update the flow state.

Algorithm 8 shows what happens when a data segment is seen in a concurrent connection. It is substantially similar to Algorithm 7 for active sequential connections.

The `endFlow` algorithm, listed in Algorithm 9, will be used frequently in the algorithms to handle RST and FIN segments. The idea is to set the current flow state to `CLOSED`, and if the opposite flow is also closed, then end the connection. `endConn`, the dual of `newConn`, deallocates

Algorithm 9 The `endFlow` algorithm

Require: $S.K \in T$
Require: $T[S.K].S_C =$ `SEQUENTIAL`
1: $T[S.K].F_I.S_F \leftarrow$ `CLOSED`
2: **if** $T[S.K].F_O.S_F =$ `CLOSED` **then**
3: `reportClose`$(S.T_S,\ S.K)$
4: `endConn`$(T,\ S.K)$
5: **end if**

connection-tracking state.

Algorithm 10 lists what happens when a RST segment is received. By TCP semantics, the sender of the RST segment guarantees that no new data will be sent, which enables `adudump` to deduce that any seen or inferred ADU in progress by the RST sender can immediately be ended and reported. The algorithm is predicated on δ, the amount of "gap" seen between the next expected sequence number and the sequence number of the RST segment, which is computed on line 1. Note that if δ is negative, no processing occurs. When δ is 0 or 1 (lines 3–18), all the data sent in the current flow has been accounted for. If the flow is `ACTIVE`, `adudump` will report the ADU (lines 7, 16). An active ADU in the reverse direction will also need to be reported (line 11). If the RST packet contains data (which does happen in some TCP implementations), `adudump` checks the quiet time (line 5). If δ is more than $\gamma \cdot MSS$ (line 19), then the RST probably has a spurious sequence number, and `adudump` ignores it, except that it will end and report an active ADU. The last case is when δ is more than 1 but less than $\gamma \cdot MSS$ (line 24). In this case, the RST represents a sequence number gap that probably indicates missing data, and `adudump` can infer the presence of an ADU. The ordering of the next steps is very delicate. First, if the flow is inactive, update $F_I.N_A$ and $F_I.N_N$ to span the inferred ADU (lines 25–28). Next, check for concurrency (lines 29–31). Then, if the connection is not concurrent and there is an active ADU in the reverse direction, end and report the ADU (lines 32–35). Lastly, update the flow's sequence number (line 36), report the ADU (line 37), and end the flow (line 38).

Handling the arrival of a FIN segment is the most complicated algorithm. I will break up the algorithm into three cases: (1) a concurrent connection, (2) a sequential connection with an inactive or closed reverse flow, (3) a sequential connection with an active reverse flow. Algorithm 11 lists the behavior of `adudump` in the concurrent case. This case is the easiest to handle because no matter what the current flow does, it will not affect the reverse flow.

Algorithm 10 Inbound RST segment arrives in a `SEQUENTIAL` or `CONCURRENT` connection. The outbound case is similar.

Require: $S.K \in T$
Require: $T[S.K].S_C \in \{\texttt{SEQUENTIAL},\texttt{CONCURRENT}\}$
Require: $S.F \cap \{R\} \neq \emptyset$
Require: $S.D = \texttt{IN}$
1: $\delta \leftarrow S.N - T[S.K].F_I.N_N$
2: **if** $\delta = 0 \ \wedge \ \delta = 1$ **then**
3: **if** $T[S.K].F_I.S_F = \texttt{ACTIVE}$ **then**
4: **if** $S.L > 0$ **then**
5: `checkQuietTime`$(T,\ S.K,\ \tau)$
6: $T[S.K].F_I.N_N \leftarrow S.N_L$
7: `reportADU`$(S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,\ T[S.K].S_C,\ 0)$
8: **end if**
9: **else if** $T[S.K].F_I.S_F = \texttt{INACTIVE} \wedge S.L > 0$ **then**
10: **if** $T[S.K].F_O.S_F = \texttt{ACTIVE}$ **then**
11: `reportADU`$(S.T_S,\ S.K,\ \sim S.D,\ T[S.K].F_O.N_N -_w T[S.K].F_O.N_A,$
$T[S.K].S_C,\ S.T_S - T[S.K].F_O.T_L)$
12: $T[S.K].F_O.S_F \leftarrow \texttt{INACTIVE}$
13: **end if**
14: $T[S.K].F_I.N_A \leftarrow T[S.K].F_I.N_N$
15: $T[S.K].F_I.N_N \leftarrow S.N_L$
16: `reportADU`$(S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,\ T[S.K].S_C,\ 0)$
17: **end if**
18: `endFlow`$(T, S.K, F_I)$
19: **else if** $\delta > \gamma \cdot MSS$ **then**
20: **if** $T[S.K].F_I.S_F = \texttt{ACTIVE}$ **then**
21: `reportADU`$(S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,\ T[S.K].S_C,\ 0)$
22: **end if**
23: `endFlow`$(T, S.K, F_I)$
24: **else if** $\delta > 1$ **then**
25: **if** $T[S.K].F_I.S_F = \texttt{INACTIVE}$ **then**
26: $T[S.K].F_I.N_A \leftarrow T[S.K].F_I.N_N$
27: $T[S.K].F_I.N_N \leftarrow S.N$
28: **end if**
29: **if** $\texttt{A} \in S.F \ \wedge \ S.A < T[S.K].F_O.N_N$ **then**
30: $T[S.K].S_C \leftarrow \texttt{CONCURRENT}$
31: `reportCONC`$(S.T_S, S.K)$
32: **else if** $T[S.K].F_O.S_F = \texttt{ACTIVE}$ **then**
33: `reportADU`$(S.T_S,\ S.K,\ \sim S.D,\ T[S.K].F_O.N_N -_w T[S.K].F_O.N_A,\ T[S.K].S_C,\ 0)$
34: $T[S.K].F_O.S_F \leftarrow \texttt{INACTIVE}$
35: **end if**
36: $T[S.K].F_I.N_N \leftarrow S.N$
37: `reportADU`$(S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,\ T[S.K].S_C,\ 0)$
38: `endFlow`$(T, S.K, F_I)$
39: **end if**

Algorithm 11 Inbound FIN segment arrives in a `CONCURRENT` connection. The outbound case is similar.

Require: $S.K \in T$
Require: $T[S.K].S_C =$`CONCURRENT`
Require: $S.F \cap \{F\} \neq \emptyset$
Require: $S.D =$ `IN`
1: `checkQuietTime`$(T,\ S.K,\ \tau)$
2: $T[S.K].F_I.A_H \leftarrow S.A$
3: `updateSeqNum`$(T,\ S,\ \beta)$
4: **if** $T[S.K].F_I.N_A \neq T[S.K].F_I.N_N$ **then**
5: `reportADU`$(S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,\ T[S.K].S_C,\ 0)$
6: **end if**
7: `endFlow`$(T, S.K, F_I)$

Algorithm 12 Inbound FIN segment arrives in a `SEQUENTIAL` connection, and the outbound flow is either `INACTIVE` or `CLOSED`. The algorithm for an outbound FIN segment is similar.

Require: $S.K \in T$
Require: $T[S.K].S_C =$`SEQUENTIAL`
Require: $T[S.K].F_O.S_F \in \{$`INACTIVE`, `CLOSED`$\}$
Require: $S.F \cap \{F\} \neq \emptyset$
Require: $S.D =$ `IN`
1: **if** $T[S.K].F_I.S_F =$`ACTIVE` **then**
2: **if** $S.N >_w T[S.K].F_I.N_N \vee S.L > 0$ **then**
3: `checkQuietTime`$(T, S.K,\ \tau)$
4: **end if**
5: `updateSeqNum`$(T,\ S,\ \gamma)$
6: `reportADU`$(S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,\ T[S.K].S_C,\ 0)$
7: **else if** $T[S.K].F_I.S_F =$`INACTIVE` **then**
8: **if** $S.N >_w T[S.K].F_I.N_N \vee S.L > 0$ **then**
9: **if** $T[S.K].F_O.N_N >_w S.A \wedge S.N_L >_w T[S.K].F_O.A_H$ **then**
10: $T[S.K].S_C \leftarrow$`CONCURRENT`
11: `reportCONC`$(S.T_S, S.K)$
12: **end if**
13: **if** $|T[S.K].F_I.N_N - S.N_L|_w \leq \gamma \cdot MSS$ **then**
14: $T[S.K].F_I.N_A \leftarrow T[S.K].F_I.N_N$
15: $T[S.K].F_I.N_N \leftarrow S.N$
16: `reportADU`$(S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,\ T[S.K].S_C,\ 0)$
17: **end if**
18: **end if**
19: **end if**
20: `endFlow`$(T, S.K, F_I)$

Algorithm 13 Inbound FIN segment arrives in a `SEQUENTIAL` connection, and the outbound flow is `ACTIVE`. The algorithm for an outbound FIN segment is similar.

Require: $S.K \in T$
Require: $T[S.K].S_C =$`SEQUENTIAL`
Require: $T[S.K].F_O.S_F =$`ACTIVE`
Require: $S.F \cap \{F\} \neq \emptyset$
Require: $S.D =$ `IN`
1: **if** $T[S.K].F_O.N_N >_w S.A \wedge S.N_L >_w T[S.K].F_O.A_H$ **then**
2: $T[S.K].S_C \leftarrow$`CONCURRENT`
3: `reportCONC`$(S.T_S, S.K)$
4: **else if** $S.N >_w T[S.K].F_I.N_N \vee S.L > 0$ **then**
5: `reportADU`$(S.T_S,\ S.K,\ \sim S.D,\ T[S.K].F_O.N_N -_w T[S.K].F_O.N_A,$
$T[S.K].S_C,\ S.T_S - T[S.K].F_O.T_L)$
6: $T[S.K].F_O.S_F \leftarrow$`INACTIVE`
7: **end if**
8: **if** $S.N >_w T[S.K].F_I.N_N \vee S.L > 0$ **then**
9: $T[S.K].F_I.N_A \leftarrow T[S.K].F_I.N_N$
10: $T[S.K].F_I.A_H \leftarrow S.A$
11: $T[S.K].F_I.N_N \leftarrow S.N_L$
12: `reportADU`$(S.T_S,\ S.K,\ S.D,\ T[S.K].F_I.N_N -_w T[S.K].F_I.N_A,\ T[S.K].S_C,\ 0)$
13: **end if**
14: $T[S.K].F_I.S_F \leftarrow$`CLOSED`

Algorithm 12 lists the behavior of `adudump` in the second FIN case, when the connection is `SEQUENTIAL` and the reverse (in this case `OUTBOUND`) flow is **not** `ACTIVE`. In this case, we do not need to worry about ending and reporting the active ADU in the reverse direction (or the associated concurrency checks). The cases of the reverse flow's state (INACTIVE or CLOSED) are handled almost the same, except that if it's closed, we will subsequently close the connection as well.

Algorithm 13 lists the process of handling an inbound FIN segment in a sequential connection when the outbound flow is active. Note that, because the connection is sequential, the outbound flow being active implies that the inbound flow is inactive, which simplifies things. The ordering of this algorithm is delicate and not obvious. First, `adudump` checks for concurrency, because if concurrency is detected on this segment, the connection state should be changed (and the concurrency event reported) before reporting any ADUs or connection close events. Therefore, the concurrency check in lines 1–3 occurs first. If concurrency was not detected and the current segment indicates an inbound ADU (either by a sequence number gap or a payload), end and report the outbound ADU (lines 4–7) before the new inbound ADU (lines 8–13). Because the

segment is a FIN, it constitutes a promise by the host that it will not send any more data, which is why `adudump` does not wait for more data before reporting the ADU. Line 14 closes the inbound flow; `adudump` does not call `endFlow` here because the outbound flow is never closed at this point.

3.1.5 Challenges and limitations of the one-pass approach

It is relatively easy to get the A–B–T inference algorithms of `adudump` working correctly for 99% of connections. The last one percent, however, is very difficult. Not only can rare sequences of packet losses create ambiguity from the perspective of `adudump`, but incorrect TCP implementations can result in unexpected situations. This section discusses some of the difficulties faced by `adudump` and lists some of the limitations of `adudump`.

These limitations were discovered when comparing the output of `adudump` with the output from `tcp2cvec`, the original A–B–T inference tool [HC06]. As discussed in Section 3.1.2, since `tcp2cvec` operates on input sorted by time within a TCP connection, it only models a single connection at a time, so it can afford to keep extensive connection state in memory. The two tools were compared on the connections in a 32 second TCP packet header trace containing approximately 4.2 million packets, 216 thousand flows, and 80,533 connection beginnings. `tcp2cvec` makes guesses about a connection vector even when it does not see the beginning of the connection, which is not difficult since it can assume that the first packet it sees has the lowest sequence number. `adudump`, however, has a much more difficult problem and does not attempt to track connections for which it missed the beginning. Thus, only the 80,533 connections for which the trace has the SYN packet (out of 103,488 total connections) were considered.

One difference between the tools is that `adudump` ignores segments that arrive for a flow that has already been closed, whereas `tcp2cvec` does not. So, for example, `tcp2cvec` detected 7 connections that were actually concurrent, but `adudump` did not detect concurrency because one of the flows was already closed. In another 14 connections, misbehaved TCP implementations send RST segments with false acknowledgement numbers, which cause the other side to respond with a RST segment with a spurious sequence number. The γ parameter of the previous section bounds the error in this case to at most three times the MSS. These limitations are acceptable

because the very same design decisions allows `adudump` to avoid cases in which `tcp2cvec` reflects arbitrary sequence numbers in RST and FIN segments and spuriously reports large (1.2 GB in one case) ADU sizes. A falsely reported 1.2 GB ADU is a much more serious issue for my problem than a handful of small errors. Related to this, `adudump` does not attempt to infer a think time at the close of a connection because the close of a TCP connection is complicated, making a one-pass approach difficult and ambiguous.

`adudump` does not update a flow's T_L timestamp when the segment is a retransmission, whereas `tcp2cvec` does. In 26 connections, this caused `adudump` to split an ADU when `tcp2cvec` did not. Neither choice is correct in an absolute sense, and the better choice is debatable. Updating the timestamp means that accidental packet losses can mask a legitimate application quiet time, whereas not updating the timestamp means that some retransmission timeout events may be lead to an ADU being split when the application had no such intent. The fact that there are potential downsides to either option reflects a limitation of the passive measurement technique common to `adudump` and `tcp2cvec`'s to accurately know the intent of the application.

A limitation of `adudump` is that it does not estimate the round-trip-time per connection (other than measuring it once during connection establishment), nor does it incorporate such an estimate into the quiet time test. Therefore, it sometimes interprets normal network delays as quiet times, thus splitting the ADU. Related to this is that `adudump` does not incorporate the TCP behavior of a retransmission timeout (RTO) into its inference algorithms, so RTO events can sometimes be interpreted as a quiet time. Both of these events would require significant additions to `adudump` and occur rarely (only 2 of the 88,000 connections had an RTO erroneously flagged as a quiet-time). The error in such a case is small: the number of bytes sent in a flow is unchanged, as are all think times.

3.2 Validation

The heuristics that `adudump` uses to infer TCP connection structure are complex. Therefore, it is important to validate the correctness of `adudump` against a "ground truth" knowledge of application behaviors. Unfortunately, doing so would require instrumentation of application programs. As this is not feasible, I instead used a set of *synthetic applications* to generate and

send/receive ADUs with interspersed think times.

To create stress cases for exercising `adudump`, the following were randomly generated from uniform distributions (defined by run-time parameters for minimum and maximum values) each time they were used in the application: number of ADUs in a connection, ADU sizes, inter-ADU think times, socket read/write lengths, and socket inter-write delays. There was no attempt to create a "realistic" application, just one that would create a random sample of plausible application-level behaviors that would exercise the `adudump` heuristics. The generated ADU sizes and inter-ADU think times as recorded by the synthetic applications comprise the ground truth, or the *actual* data. These synthetic applications were run on both sides of a monitored network link. I captured the packets traversing the link, saving the trace as a pcap file which I then fed as input to `adudump`, producing the *measured* data. In this way, I can determine how correctly `adudump` functions.

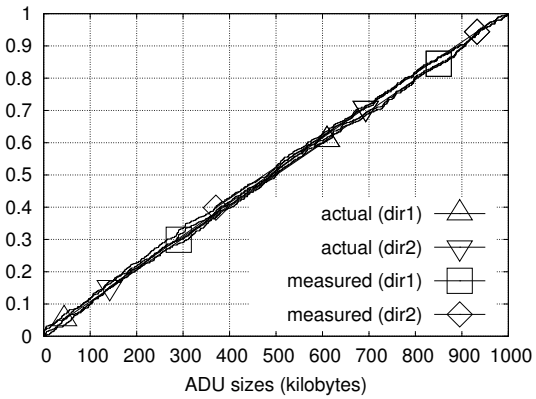
I first tested `adudump` on sequential connections only and then on concurrent connections only. I will consider each of these cases in turn.

3.2.1 Sequential Validation

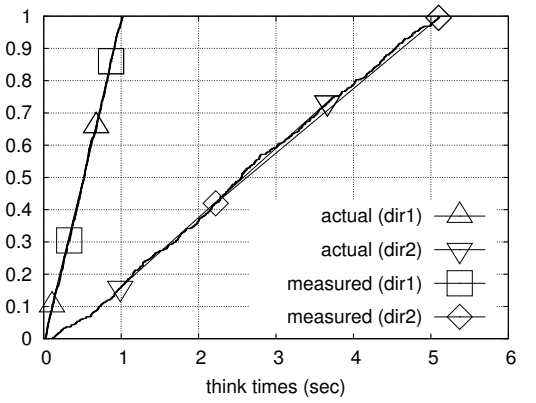
These tests produced sequential traffic because the application instances using a TCP connection take turns sending ADUs. That is, they do not send an ADU until they finish receiving the previous ADU. A random packet loss rate of 1% was introduced by FreeBSD's `dummynet` mechanism, which was also used to introduce random per-connection round-trip times. As with application behavior I use plausible randomized network path conditions to test `adudump`, but I do not claim realism.

Figure 3.4(a) plots the actual and measured per-direction ADU size distributions. The distributions are nearly identical. The slight differences are because, in the case of TCP retransmission timeouts (with sufficiently long RTT), `adudump` splits the ADUs, guessing (incorrectly in this case) that the application intended them to be distinct. As discussed in Section 3.1, the default quiet time threshold, which governs this behavior, is 500 milliseconds, so RTTs shorter than this threshold do not split the ADU. I chose 500 ms as a reasonable trade-off between structural detail and solid inference of application intent.

Similarly, Figure 3.4(b) plots the actual and measured think-time distributions. Note that

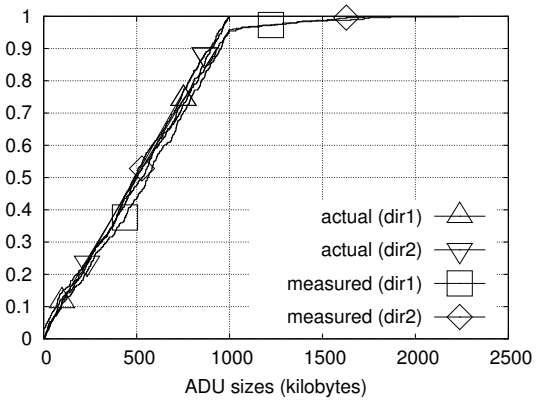


(a) CDF of actual vs. measured ADU size distributions, for either direction.

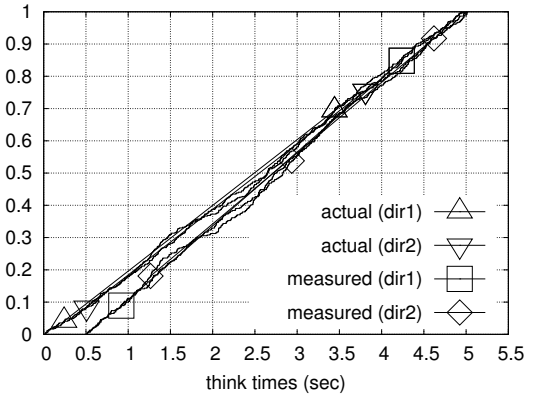


(b) CDF of actual vs. measured think-time distributions, for either direction.

Figure 3.4: Sequential validation results



(a) CDF of actual vs. measured ADU size distributions, for either direction.



(b) CDF of actual vs. measured think-time distributions, for either direction.

Figure 3.5: Concurrent validation results

the actual distributions were different for each direction. Note also that, unlike ADU sizes, `adudump` cannot exactly determine the actual time, because some non-negligible time elapses between the application's timestamp and the monitor's packet capture. Even so, `adudump`'s measurements are quite accurate.

3.2.2 Concurrent Validation

Although I do not consider the problem of inferring server performance from concurrent connections in this dissertation, it is still useful to validate that `adudump` constructs correct connection vectors in such cases. In the concurrent tests, each application instance sends multiple ADUs with interspersed think times without synchronizing on the ADUs they receive. I did not introduce packet loss in this case.

Table 3.5: the types of records output by `adudump`

Type	Information	Description
SYN	t, x, y, d	the initial SYN packet was seen at time t in direction d between host/port x and host/port y; connection-tracking state established
SAK	t, x, y, d	missed the SYN, but saw the SYN-ACK, giving us enough information to establish connection-tracking state
RTT	t, x, y, d, r	the SYN-ACK packet seen and round-trip-time measurement r
SEQ	t, x, y, d	the connection establishment
CONC	t, x, y, d	the connection has been determined to be concurrent
ADU	t, x, y, d, b, T	an application-level data unit was seen of size b bytes, and there was a think-time afterwards of T seconds. (The think-time is not always available.)
INC	t, x, y, d	report an ADU in progress (e.g. when input is exhausted)
END	t, x, y, d	the connection is closed; connection-tracking state destroyed

Figure 3.5(a) plots the actual and measured per-direction ADU size distributions. The measured data tracks the actual data well for most of the distribution, but diverges in the tail, demonstrating an important limitation of `adudump`'s passive inference abilities: if one of the applications in a concurrent connection has a genuine application-level think time between ADU transmissions that is less than the quiet-time threshold, then `adudump` will not detect it, and it combines the two ADUs into one that is larger. This is a type of inference error that is unavoidable because I do not have access to applications running on end systems. Nevertheless, this is a sacrifice I gladly make, because it enables generic access to the behavior of any server without any server instrumentation.

Figure 3.5(b) plots the actual and measured quiet-time distributions. The measured distributions track well with the actual distributions except that, as expected, there are no think times less than 500 ms, the quiet-time threshold.

3.3 Data

The `adudump` tool reports the size of the ADUs, the application-level "think time" between ADUs, and a variety of contextual information, as shown in Table 3.5. Table 3.6 shows the output of `adudump` for an example connection.

I have used `adudump` to generate a six-month data set of records of all TCP connections en-

Table 3.6: `adudump` output format for an example connection
(IP addresses (but not ports) have been anonymized.)

```
SYN: 1202706002.650917 1.2.3.4.443 < 5.6.7.8.62015
SEQ: 1202706002.681395 1.2.3.4.443 < 5.6.7.8.62015
ADU: 1202706002.688748 1.2.3.4.443 < 5.6.7.8.62015 163 SEQ 0.000542
ADU: 1202706002.733813 1.2.3.4.443 > 5.6.7.8.62015 2886 SEQ 0.045041
ADU: 1202706002.738254 1.2.3.4.443 < 5.6.7.8.62015 198 SEQ 0.004441
ADU: 1202706002.801408 1.2.3.4.443 > 5.6.7.8.62015 59 SEQ
END: 1202706002.821701 1.2.3.4.443 < 5.6.7.8.62015
```

Table 3.7: Dataset 1: `adudump` measurements from border link

begin[a]	end	duration[b]	outage	size	records	ADUs	conns
Fr 03/14[c] 22:25	Th 04/17 03:50	33:05:25[d]	1:14:11	813 GB	11.8 B	8.8 B	820 M
Fr 04/18 18:01	We 04/23 07:39	4:13:37	0:03:35	106 GB	1.6 B	1.1 B	116 M
We 04/23 11:14	Th 04/24 03:00	0:15:46	0:07:38	16 GB	234 M	161 M	19 M
Th 04/24 10:38	Fr 05/16 11:19	22:00:41	0:07:04	530 GB	7.7 B	5.7 B	532 M
Fr 05/16 18:23	Fr 05/23 00:06	6:05:43	5:16:20	108 GB	1.6 B	1.07 B	148 M
We 05/28 16:26	Mo 06/30 16:45	33:00:19	2:20:57	482 GB	7.3 B	4.7 B	686 M
Th 07/03 13:42	Fr 08/01 07:07	28:17:25	0:00:10	361 GB	5.7 B	3.5 B	563 M
Fr 08/01 07:17	Tu 08/19 13:12	18:05:55	1:02:05	273 GB	4.1 B	2.7 B	346 M
We 08/20 15:17	Mo 09/01 22:36	12:07:19	0:21:15	242 GB	3.6 B	2.5 B	271 M
Tu 09/02 19:51	We 10/01 21:25	29:01:34	n/a	629 GB	9.2 B	6.5 B	697 M
		188:01:44	7:04:55	3.53 TB	52.8 B	36.7 B	4.2 B

[a] all times local (EDT); all dates are 2008
[b] durations are listed as days:hours:minutes
[c] days are in MM/DD format
[d] data for Monday, March 17, was lost

Table 3.8: Dataset 2: `adudump` measurements from border link

begin[a]	end	duration[b]	outage	size	records	ADUs	conns
Tu 12/09[c] 15:00	Th 01/01 09:41	22:18:41	1:02:11	284 GB	4.3 B	2.7 B	383 M
Fr 01/02 11:52	Th 01/15 07:10	12:19:18	1:04:57	188 GB	2.8 B	1.8 B	258 M
Fr 01/16 12:07	Mo 01/19 11:22	2:23:45	2:03:55	54 GB	773 M	569 M	49 M
We 01/21 15:17	We 01/21 15:27	0:00:10	0:20:03	116 MB	1.7 M	950 k	146 k
Th 01/22 11:30	Fr 01/30 23:55	8:12:25	0:10:57	141 GB	2.1 B	1.4 B	138 M
Sa 01/31 10:52	>Tu 03/10 00:00[d]	>37:13:08	n/a	>636 GB	>9.3 B	>6.2 B	>634 M
		>82:15:27	5:18:03	>1.3 TB	>19.3 B	>12.7 B	>1.6 B

[a] all times local (EDT); all dates are Dec. 2008 – Mar. 2009
[b] durations are listed as days:hours:minutes
[c] days are in MM/DD format
[d] As of April, 2009, data collection is still ongoing, hence the > symbol

tering the UNC campus from the Internet. It is this data set that will be used for the remainder of this paper. In this data, ADUs from concurrent connections constitute approximately 5% of the connections, 25% of the ADUs seen, and 30% of the size in bytes. Overall, I collected nearly five terabytes of data, modeling about nearly 6 billion connections. Tables 3.7 and 3.8 list the individual collections, which were punctuated by measurement faults such as power outages and full disks. The duration of an outage was simply as long as it took for me to notice the outage and restart the collection process. To preserve the privacy of UNC hosts (a requirement for capturing UNC traffic), only inbound connections (*i.e.* those for which the initial SYN was sent to the UNC network) were captured.

Extracting response times

The central metric for server performance is the server response time, the time between a request and the subsequent response, also called the server-side think-time. Most of Chapters 4 and 5 will be concerned with response times. The response time is a special case of the think-time. Think-times and quiet-times are reported by `adudump` in `ADU` records. Unfortunately, the current output of `adudump` is not optimized for determining whether the time reported in an `ADU` record is a quiet-time, client-side think-time, or server-side think-time. To identify which it is, the subsequent `ADU` record for that connection must be considered. If (and only if) the current `ADU` record reports an incoming (*i.e.* client-to-server) ADU, and the subsequent `ADU` record reports an outgoing ADU, then the current ADU contains a response time. Thus, I must save a copy of the current record for each connection at all times.

Identifying response times from `adudump` output is an essential operation of any server performance analysis, and it is one of the first steps performed on the data from `adudump`. This operation is currently done offline for testing and implementation purposes. Because this operation is important yet requires a potentially large amount of memory, it would be desirable to change the output format of `adudump` and add logic to `adudump` to identify response times. For example, instead of a generic `ADU` record, I could report `REQ` records for request ADUs, `RESP` records for response ADUs, and QADU records for the ADUs separated by quiet-times. This is an area for future work.

3.4 Summary and contributions

This chapter introduced the measurement approach used in this dissertation, centered on the novel `adudump` tool. Section 3.1 gave the details of the approach and showed how the approach met each of the constraints. Section 3.2 shows that `adudump` accurately measures application behavior. Lastly, Section 3.3 introduced the data set that will be used for the remainder of the dissertation.

The contributions of this chapter are as follows:

- a **passive** measurement infrastructure for measuring server performance
- a **pervasive** measurement infrastructure for measuring the performance of many servers in a network
- a **generic** measurement infrastructure capable of measuring the performance of many *types* of servers, running different application protocols
- a **responsive** measurement infrastructure that reports server performance measurements immediately after observing them
- an **inexpensive** measurement infrastructure with a straightforward deployment

Chapter 4

Performance Anomaly Detection

This chapter is about detecting anomalies in the performance of a server. These methods will be applied to a large set of servers in Chapter 5. First, I motivate these methods in terms of the overall goals in Section 4.1. Then, I discuss various ways of representing distributions and the requirements we have for the representation in Section 4.2. Section 4.3 introduces principal components analysis (PCA), which is used in the anomaly test introduced in Section 4.4. Section 4.5 gives an extended example of the discrimination methods to build intuition. Section 4.6 discusses how to decide on a basis for normality. Sections 4.7 and 4.8 explore the setting of various parameters. Section 4.9 introduces a range of different time-scales for the analysis, and Section 4.10 introduces "ordinal" analysis, used in Chapter 5. Section 4.11 demonstrates the ability to reset the normal basis. Finally, Section 4.12 discusses some limitations of the anomaly detection approach, and Section 4.13 concludes the chapter.

4.1 Motivation

One of the biggest challenges in this research is to come up with a method of detecting performance anomalies for any sort of server. The requirement of extreme generality of our approach means that there are very few assumptions we can make *a priori* about the data. Many statistical approaches rely on assumptions such as linearity and normality. We will discuss such assumptions and their applicability to our data, and we will see that we cannot make these assumptions in all cases. Therefore, our approach needs to work despite the lack of meaningful assumptions.

At a high level, we want a generic anomaly test that can tell us when any performance anomaly occurred for any server, regardless of its function. We evaluate this test by applying it for all of the servers in UNC's network.

4.2 Distribution representations

Fundamentally, the requirement is to compare sets of measurements. For example, I often want to compare the set of response times for the past 24 hours with the set of response times over the previous month or two. In particular, note that these sets will almost never have the same number of elements. A very important issue is how the sets should be represented. Different representations lose different amounts of information, and some lend themselves more naturally to comparisons.

A univariate probability distribution is one way of representing a set of measurements. It gives, for each value in the set, a likelihood of a random variable having that value. Thus, the probability density function (PDF) of the random variable X is given by $f(X)$, which has the constraints:

$$f(x) \geq 0,\ x \in X$$

$$\int_{x \in X} f(x)dx = 1.$$

That is, no value in X can have negative probability, and the total probability of all values must sum to 1. This distribution can be equivalently described by a cumulative distribution function (CDF) $F(X)$:

$$F(x) = \int_{-\infty}^{x} f(u)du.$$

Intuitively, the CDF is the probability of the random variable being less than or equal to a given value. The CDF is a nondecreasing function. Another way of describing the distribution is by its quantile function (QF), which is also nondecreasing. The QF is the inverse of the CDF: for a given probability p, it yields the corresponding value x such that a random variable would

be less than or equal to x with probability p. One complication is that there may be multiple values of x satisfying this condition; the minimum of them is chosen. Thus, the QF $Q(p)$ is defined as:

$$Q(p) = \inf\{u : F(u) \geq p\}$$

One way of representing a set of measurements is with a *parametric* distribution, or a closed-form, analytical distribution. Many such parametric distributions exist, *e.g.* the familiar, bell-shaped Gaussian distribution, which has parameters μ (mean) and σ^2 (variance). Comparing parametric distributions—whether it is merely the parameters that might be different or whether it is the distributions themselves that might be different—is typically straightforward. However, not all data sets are well described by a parametric distribution. Indeed, because I need a generic solution, I cannot assume that it will always be possible to find a parametric solution for a data set. Therefore, I will use non-parametric, or *empirical* distributions, which partition the domain of a PDF, CDF, or QF, and assign a value to each partition.

I have several goals in choosing a representation. First, I must be able to compare distributions from the same server, even when the number and range of observations might vary widely. Second, I want a *compact* representation. Third, I want to maximize the amount of information preserved by the representation; said another way, I want to minimize the amount of information lost by summarizing the distribution into a representation. Lastly, I want a representation for which the population of distributions varies linearly. I will discuss each of these goals in turn.

The first and most essential goal of choosing a representation is that the representations of different distributions can be compared in a natural, general way. For example, consider two distributions of response times from a particular server, one from last Sunday and one from last Monday. These distributions probably have a different size (*i.e.* the number of observations), and perhaps significantly different. Furthermore, they might have a significantly different range—especially if, say, there was anomalous behavior on Monday. We need a representation that lends itself to comparisons between distributions, even with such differences.

Second, I want a *compact* representation. This means that using a full empirical distribution

(*i.e.* the entire set of observations) is not acceptable. Furthermore, as we will see, some representations can require additional information to be passed along with the representation itself, such as bin boundaries for discrete PDFs. Although bin boundaries would not preclude a representation, it would make that representation less desirable.

Any representation (other than the full empirical distribution) will involve some amount of *information loss*, because the representation is essentially a summary, or estimation, of the distribution. The third goal in choosing a representation is that I want to minimize the amount of information loss. The information in a distribution can be quantified with Shannon's information-theoretic notion of entropy, which we will introduce and use below.

Lastly, I want the set of distributions to vary *linearly*. Each distribution (*e.g.* the set of response times for a particular server and a particular day) will be summarized with a representation. We will see that these representations can be thought of as points in a high-dimensional space. I want the set of these points to form a linear space within this high-dimensional space. For example, I desire that the points might form a line (one inherent dimension), or a plane (two inherent dimensions), or a hyper-plane (three or more inherent dimensions), in this high-dimensional space. I do not want the points to form a curved space such as a curved line or a curved surface, because the analysis of such spaces is much more difficult and computationally intensive.

Given that we cannot use parametric representations (in which we could simply compare the parameters among distributions), we must find another way of allowing comparisons to be made between distributions. The typical way of doing this is to use a "binned" representation. In other words, the range of the empirical distribution is partitioned into a fixed number of bins, and the value of each bin is a function over the observations within that bin. Note that the number of bins will not change. It is a parameter to my methods that I will call `num_bins`, and its space is explored in Section 4.7.4.

Perhaps the most common empirical representation is a *histogram*, which is a discrete PDF. The range of the distribution X is partitioned into subsets p_i, forming b "bins". Each observation falls into exactly one bin. Let $x_i, i = 1 \ldots N$ be the distribution of N observations, and let $[a_i, b_i)$ be the boundaries of partition p_i, with $a_1 = -\inf andb_n = \inf$. Then a histogram $\underline{h}$, a vector, is defined as the count of observations per bin:

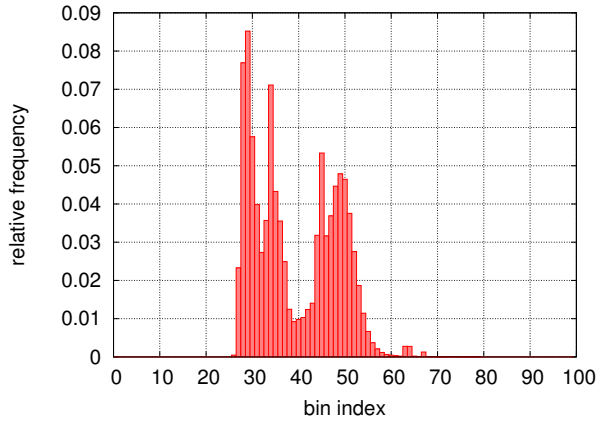


Figure 4.1: Example histogram for an example server, with bins divided equally according to a logarithmic scale.

$$h_i = |\{x_j : a_i \leq x_j < b_i\}|\,,\ i = 1 \ldots b.$$

Note that the constraints of this discrete PDF are parallel to the constrains of a continuous PDF: $\sum_{i=1}^{b} h_i = 1$ and $0 \leq h_i \leq 1, i = 1 \ldots b$. Similarly, a cumulative histogram $\underline{H}$ is the discrete version of a CDF:

$$H_i = |\{x_j : x_j < b_i\}|\,,\ i = 1 \ldots b.$$

where $0 \leq H_1 \leq \cdots \leq H_b \leq 1$.

In many applications, discrete PDFs and CDFs use bins of equal size. (That is to say, the "width" of bins, not the count of observations per bin, are typically equal.) However, this approach works poorly for most sets of server response times (and request/response sizes). Since the data varies over several orders of magnitude, a better choice for bins is to equalize their widths on a logarithmic scale. For example, Figure 4.1 shows a PDF of response times for an example server with 100 bins, where the 99 bin boundaries are equally spaced between the logarithms of the minimum and maximum response times in the distribution. Unfortunately, even this scheme yields a histogram with many near-zero values. This effect is undesirable because the informational content of the histogram is then reduced. A histogram with b bins

is essentially a vector of length b, which we can think of as a point in b-dimensional space. If one of the bins is always zero, then we actually have a $(b-1)$-dimensional space embedded in the b-dimensional space; in other words, we are wasting a dimension. This would be like using three dimensions to describe a set of points, even though all the points fall on the x-y plane.

The informational content of a histogram is defined using Shannon's entropy [Sha48], $H(\underline{h})$, where $\underline{h}$ is a histogram as defined above, with b bins. Additionally, let n be the total number of observations. Then the entropy is:

$$H(\underline{h}) = -\sum_{i=1}^{b} \frac{h_i}{n} \log_2 \left(\frac{h_i}{n} \right)$$

Intuitively, the entropy is at its maximum value of $\log_2(b)$ when each bin of the histogram has the same count (*i.e.* the histogram is "flat"), and the entropy is at a minimum value of 0 when all of the observations fall into a single bin (*i.e.* the histogram is a single "spike"). Recall that we want to minimize the amount of information lost when summarizing the original distribution into a representation. This means that we want to find a representation of the distribution with a large entropy. A simple approach to determining the bin boundaries is not acceptable because it will often result in many empty bins, and thus significant information loss. Therefore, we need to pick the bin boundaries more carefully.

However, because we need to compare distributions, we need a single set of bin boundaries to use for all the distributions of a particular server. Even a careful partitioning of the range of the distribution will result in a poor choice of bin boundaries for an anomalous distribution, because the range can vary significantly. Furthermore, a more carefully tuned scheme for selecting bin boundaries will result in a less compact distribution: instead of specifying bin boundaries with two values (*e.g.* "the bin boundaries are equally spaced between x and y"), we now need to give the entire set of $(b-1)$ bin boundaries.

Instead, consider a discrete version of the QF. The crucial difference between a QF and a PDF or CDF is that the domain of a QF is fixed. A PDF or CDF will have a domain defined by the range of observations seen in a distribution, whereas a QF's domain is always $[0, 1]$, the set of valid quantiles. Therefore, a QF works equally well, regardless of the range of a distribution. Furthermore, dividing the domain of a QF evenly results in a set of bins

that contain an *equal* number of observations (ignoring discretization effects), thus yielding maximum entropy. However, a simple count of the values in each bin would be rather boring, because it would always be flat, so instead, we average the values in each bin, thus preserving the effect of outliers. Let $\underline{Q}$ (a vector) be the discrete quantile function of a set X:

$$Q_i = b \int_{\frac{i-1}{b}}^{\frac{i}{b}} Q(x)dx,\ i = 1, \ldots, b,$$

For example, say we want the second value of the 10-bin discrete QF of the distribution X, or Q_2. The second bin corresponds to quantiles in the range of $[0.10, 0.20)$, *i.e.* the 10th through 19th percentiles. If the 0.10 quantile corresponds to a value of j (meaning 10 percent of values in X are less than or equal to j), and the 0.20 quantile corresponds to k, then let c be the sum of all observations in X within the range $[j, k)$, and d be the count of the observations. Then Q_2 is simply c/d. Note that, when computing a 10-bin discrete QF of multiple distributions, the bin boundaries of *e.g.* $[0.10, 0.20)$ will never change, even though j, k, c, d, and Q_i will change among the distributions.

The discrete QF does not need to worry about finding appropriate bin boundaries, and it preserves more information than either a discrete PDF or a discrete CDF with poorly chosen bin boundaries. Clearly, it is the best choice of a representation so far. However, there is another option. We could let the quantile function over some "training" distribution (*e.g.* the first two weeks of response times for a server) define the permanent bin boundaries of PDFs/CDFs of corresponding measurements for that server. I will call these *adaptive* PDFs and CDFs. The aforementioned drawbacks to PDFs and CDFs still apply to adaptive PDFs/CDFs (or aPDFs/aCDFs): these bin boundaries would not adapt well to anomalous distributions with significantly different ranges, and they are not as compact because the bin boundaries must be saved. However, they have the advantage of minimizing the information loss for the PDF/CDF representations. We will compare these three representations (QFs, aPDFs, and aCDFs) in terms of the final goal: linearity among distributions.

Recall that, as I summarize a distribution into a b-bin PDF or CDF or QF, I can consider that representation (which is actually a vector of length b) as a point in b-dimensional space. The fourth and final goal of choosing a representation is stated in terms of the "spread" or

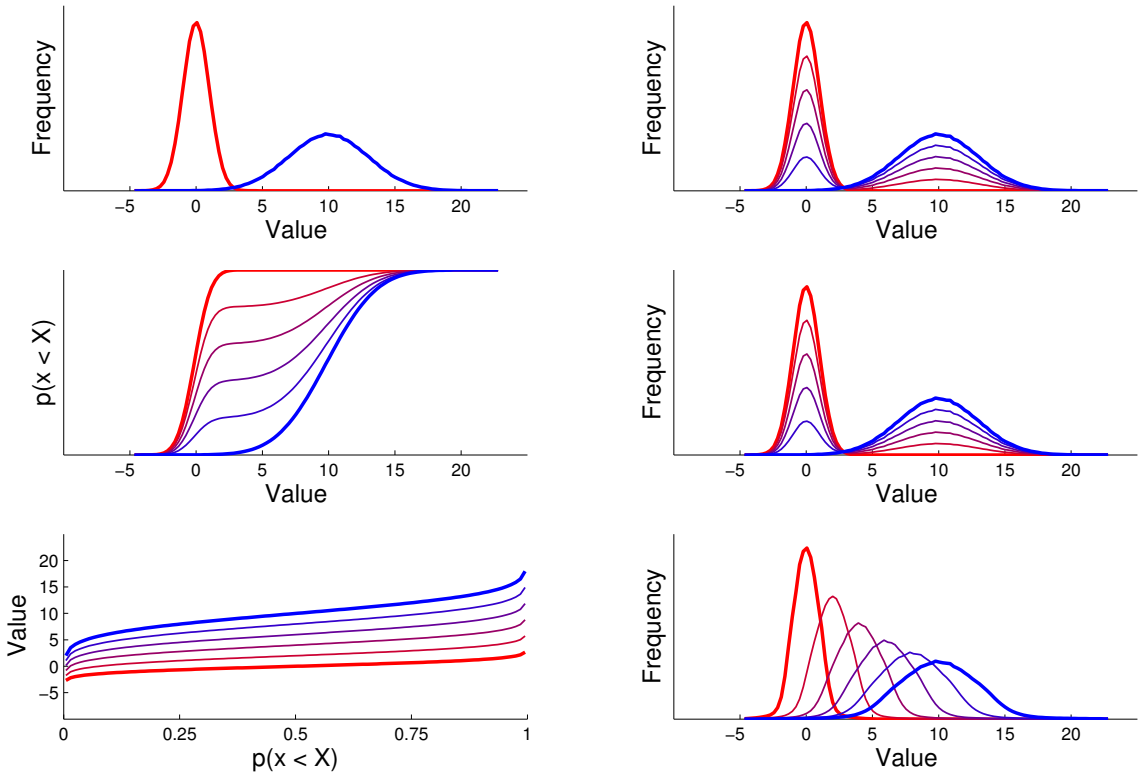
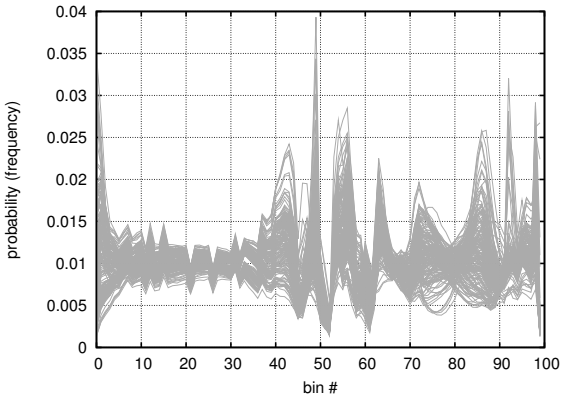


Figure 4.2: Interpolations between two Gaussian distributions (one red, one blue), shown upper-left. PDF interpolation, upper-right. CDF interpolation and the corresponding PDFs (middle row). QF interpolation and the corresponding PDFs (bottom row). From [Bro08].

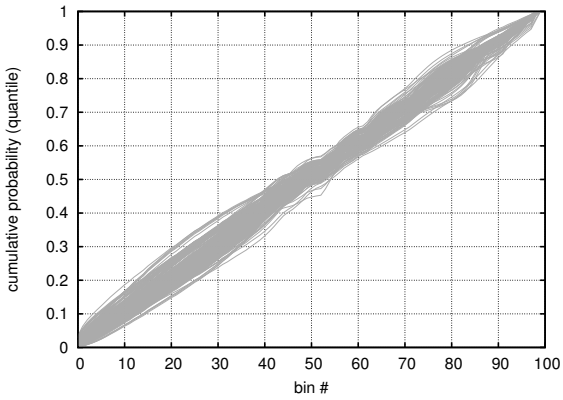
variation of a population of such points in the b-dimensional space: I want the spread to be linear, forming a plane or hyper-plane in the space, rather than a nonlinear variation, which forms a curved manifold. Perhaps surprisingly, the choice of representation does affect the linearity of the variation. Broadhurst [Bro08] showed how different types of variation among distributions can have linear variation using certain representations. For example, if all of the distributions in a population are parametric (*e.g.* Gaussian), and the distributions vary in terms of the parameters, then the QF representation will have linear variation. Figure 4.2 illustrates this effect. Consider two Gaussian distributions. Each can be represented as a PDF or a CDF, and each of these representations can be considered a point in a multidimensional space (where the number of bins determines the number of dimensions). An *interpolation* is a point in this space that lies on the line connecting the two distributions. An interpolation in the PDF (top) or CDF (middle) space yields a mixture of two Gaussian distributions, whereas an interpolation in the QF space (bottom) yields a pure Gaussian distribution. On the other hand, if the distributions are mixtures of parametric distributions, and they vary in terms of the weights of the mixture (*e.g.* one distribution is forty percent Gaussian and sixty percent Poisson, whereas another is fifty/fifty), then the PDF and CDF will have linear variation.

As mentioned above, however, I rarely see distributions in my data that conform to a well-known parametric representation. In such cases, Broadhurst uses PCA, or principal components analysis, to determine the extent to which a particular representation varies linearly among the population, relative to the other representations. I will formally define PCA in the next section, but intuitively, if the data forms *e.g.* a two-dimensional plane in b-dimensional space, then PCA will compute the perpendicular vectors defining the plane, even if the plane is skewed from the input coordinate system. In such a case, the first two *singular values* returned by PCA, which are sorted descendingly, will be nonzero, and all others will be zero. The same is true if the data *inherently* varies linearly in any d-dimensional subspace, $d < b$: only the first d singular values (which, again, are sorted) will be nonzero. With real data (and computers with limited precision), the singular values are rarely zero, but can often be close enough to zero as to be negligible.

I briefly explore this issue below. I looked at 179 day-long distributions of response times for the top ten most popular servers in my data, thus forming ten populations, each of size

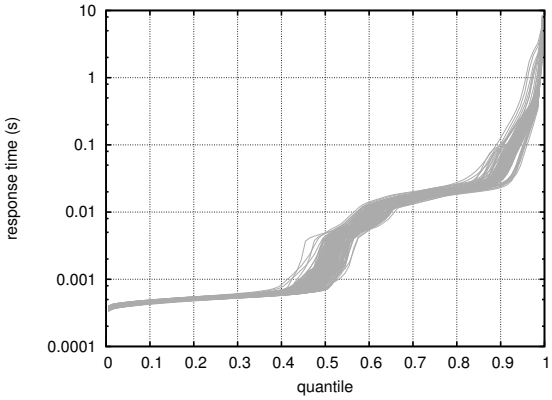


(a) Adaptive PDF representation

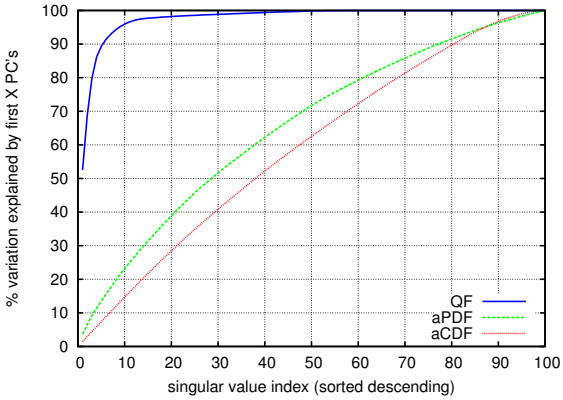


(b) Adaptive CDF representation

Figure 4.3: Distributions of response time measurements using aPDF and aCDF representations.



(a) Quantile function representation



(b) Cumulative normalized singular values for QF, aPDF, and aCDF representations.

Figure 4.4: Quantile function representation and singular values of PCA using various representations.

179 distributions (or points). These distributions were represented as 100-bin aPDFs (Figure 4.3(a)), 100-bin aCDF (Figure 4.3(b)), and 100-bin QF (Figure 4.4(a)). For the aPDF and aCDF representations, I chose bin boundaries according to the QF of the entire population, so any widely varying ranges of response times were accounted for before the bin boundaries were chosen.

Figure 4.4(b) shows the cumulative normalized singular values from PCA over each of the representations, for one of the ten servers. The quicker the curve converges to $y = 1$, the better PCA is able to capture the variation of the population using a particular representation, and hence the more linear the variation using that representation. We see that the QF representation

yields the most linear variation for this server. Indeed, the same is true for all ten servers, though I omit the plots here. Therefore, I will only use the QF representation throughout this dissertation, unless noted otherwise.

4.3 Principal Components Analysis

Here I introduce PCA, providing intuition of the process and significance of PCA along with a formal mathematical definition.

Principal components analysis, or **PCA**, finds the *principal components*, or the directions of greatest variation, in multivariate data. Because multivariate data is often correlated, the information contained in the data can often be represented in fewer dimensions. For example, data points in three dimensions (*i.e.* tri-variate data) might be planar, forming a two-dimensional object in three-dimensional space. PCA will find the vectors along which the data vary, and the extent to which the data varies along each of these vectors. So, in the example of planar data in three dimensions, PCA gives us three vectors, with the third being perpendicular to the data plane and having a corresponding weight of zero.

Say we have an $n \times b$ data matrix $\mathbf{X} = \{X_{ij}\}$. The n rows of $\mathbf{X}$ are discrete quantile functions, vectors of b elements each. The b columns, vectors $\overline{X_j}$, are the measurements corresponding to a particular dimension (in this case a bin of the QF); for example, $\overline{X_1}$ is the first bin from each quantile function. Thus, $\mathbf{X}$ has dimensions $n \times b$. To compute the principal components, we first compute the mean vector μ, or the row vector containing each of the column means:

$$\mu_j = \frac{1}{n} \sum_{i=1}^{n} X_{ij},\ j = 1 \ldots b$$

We then center the data at the origin by subtracting out μ to get a *centered* data matrix $\mathbf{Z}$:

$$Z_{ij} = X_{ij} - \mu_j,\ i = 1 \ldots n, j = 1 \ldots b$$

Now, the mean of each column of $\mathbf{Z}$ is zero. Now we compute $\mathbf{C}$, the covariance matrix of $\mathbf{Z}$. The covariance of two vectors $\overline{X_i}$ and $\overline{X_j}$ is given by:

$$\mathsf{cov}(\overline{X_i}, \overline{X_j}) = E[(\overline{X_i} - E[\overline{X_i}])(\overline{X_j} - E[\overline{X_j}])],$$

Where E is the expectation operator: $E[\overline{X}] = \frac{1}{|\overline{X}|} \sum_{i=1}^{|\overline{X}|} X_i$. Note that because we already centered the data, $E[\overline{Z_j}] = 0, j = 1 \ldots b$, where $\overline{Z_j}$ is the j^{th} column of $\mathbf{Z}$. The covariance matrix $\mathbf{C}$ then gives the covariance of the *columns* (or dimensions) of $\mathbf{Z}$:

$$C_{ij} = \mathsf{cov}(\overline{Z_i}, \overline{Z_j}) = E[(\overline{Z_i})(\overline{Z_j})] = \frac{1}{n} \sum_{k=1}^{n} Z_{ik} Z_{jk}$$

At this point, PCA is simply the eigenvalue decomposition of $\mathbf{C}$; that is, representing $\mathbf{C}$ in terms of its eigenvectors and eigenvalues. The eigenvectors are sorted by their corresponding eigenvalue so that the most significant eigenvector is first. The first eigenvector (or *principal component*), V_1, represents the vector of greatest variation among the original data. The second eigenvector, V_2, is orthogonal to the first, and represents the greatest *remaining* variation after ignoring the first eigenvector. This process repeats for all b eigenvectors, each of them linearly independent from all the previous ones, thus forming a $b \times b$ orthonormal basis $\mathbf{V}$, which is also called a rotation matrix. The eigenvalues $\{\lambda_i\}$ specify the standard deviation of the data along each of these eigenvectors. In some cases, primarily when $n < b$, the rank of the covariance matrix will be less than b, and there will not be b legitimate eigenvectors. In such a case, the corresponding eigenvalues will be zero, and PCA will choose arbitrary vectors to complete the basis $\mathbf{V}$, obeying the constraint that all vectors are mutually orthogonal.

4.4 Discrimination

The basic operation of my methodology is to decide whether a given distribution of *e.g.* response times for a particular server is well described by the historical population of distributions of response times for the same server. If it is, the new distribution is considered "normal", or "non-anomalous". Otherwise, the distribution is considered anomalous.

This is a difficult operation because of the requirement that it work equally well on a wide variety of input data. The population of distributions might be normally distributed in the

quantile-function space—but typically it is not. The population might have a wide "spread", or it might not. The shape of the distributions is completely arbitrary. In general, there are very few assumptions that I can make globally for all servers, and my methods must work well in all cases.

Let's assume we have a set of n distributions, each a b-bin quantile function, as the input data. We represent this dataset as a $n \times b$ data matrix, $\mathbf{X}$. As detailed in the previous section, this data matrix is first zero-centered by subtracting the mean vector, and the means of each column are saved in the vector μ. The result is then analyzed according to PCA to determine the primary (orthogonal) components of variation (the eigenvectors or principal components, V_i), and the amount of variation accounted for by each component (the eigenvalues or singular values λ_i).

I then form a *basis*, which defines the sort of distribution that is considered "normal", or typical, based on the population seen so far. First, let L be the total variation of all the principal components, or $\sum_{i=1}^{b} \lambda_i$. Then let R_i be the amount of variation remaining after principal component i:

$$R_i = \sum_{j=i+1}^{b} \left(\frac{\lambda_j}{L} \right), \; i = 1 \ldots b$$

Note that R_i is always less than one, R_b is zero, and the sequence of R_i is nonincreasing. The "normal" basis is formed by including principal components until R_i is less than some threshold value, which I call `basis_error`. In other words, if there are k principal components in the basis, then R_k is the last value of R that is greater than `basis_error`. Intuitively, the `basis_error` parameter controls the specificity of the basis: the lower the value, the more specific to the input the basis will be, and the higher the value, the looser it will be. This parameter is discussed in more detail in Section 4.7.1.

The basis B is a $b \times k$ matrix, where k is the number of principal components included. This basis represents a k-dimensional subspace of the original b dimensions, and it is the *inherent* dimensionality of the data at a tolerance of `basis_error`. Each of the input points (which are the original distributions) have a certain *projection error* onto this basis; that is, there is some distance between the original distribution and its projection onto the k-dimensional subspace.

If Z is a (b-dimensional) point and Z' is its (k-dimensional) projection onto the basis, then let Z'' be the projection expressed in terms of the original b dimensions, or $Z'B^T$. Then the projection error of Z is defined as the Euclidean distance from Z to Z'':

$$PE(Z) = \left(\sum_{i=1}^{b} (Z_i - Z_i'')^2 \right)^{\frac{1}{2}}$$

I call the mean of the projection errors μ_p, and the variance of the projection errors λ_p. In this way, I essentially "roll up" all of the leftover dimensions into a single dimension, which I call the "error" dimension. Thus, the number of dimensions in the basis is controlled by the inherent dimensionality of the data, rather than an arbitrary choice of b.

I now describe the basic operation of the discrimination process, which I call the *anomaly test*. The idea is to compare a distribution to be tested for being anomalous against the basis formed by a population of distributions, and to quantify the extent to which the tested distribution is *not* described by the basis. First, the tested distribution (which is, in its QF representation, a b-dimensional point) is zero-centered by subtracting the mean vector, μ, to get a (row) vector I will call T. T is then multiplied by the basis B to get T', the projection of T onto the basis. One can think of this operation as changing the coordinate system of T from the original coordinates (the quantile bin values from $1 \ldots b$) to the PCA-defined coordinates (the distance along principal components $1 \ldots k$). Then, I compute $PE(T) - \mu_p$, or the zero-centered projection error of T, and call it p.

At this point, I have T', the zero-centered coordinates of the test distribution in terms of the principal components, λ, the variances of the original data along each of the principal components, p, the zero-centered projection error of the test distribution, and λ_p, the variance of the original projection errors. In other words, I have $k + 1$ zero-centered values along with the variance of each of these values as defined by the original population. I can now compute the *standardized distance* from the basis to the tested distribution. The standardized distance is a special case of the Mahalanobis distance in which the covariance matrix is diagonal. Our covariance matrix is diagonal because PCA returns a set of components that are mutually orthogonal, thus removing correlations among the original components. The standardized distance between two d-dimensional points X and Y is defined as:

$$SD(X,Y) = \sum_{i=1}^{d} \left(\frac{(X_i - Y_i)^2}{\sigma_i^2} \right),$$

...where σ_i^2 is the variance along dimension i. In my application, I am measuring the distance of the test distribution to the origin to compute the output of the anomaly test, which I call the *anomaly score*:

$$AS(T) = \sum_{i=1}^{k} \left(\frac{(T_i')^2}{\lambda_i} \right) + \frac{p^2}{\lambda_p}$$

Intuitively, the anomaly score measures the extent to which the test distribution is *not* described by the basis. If the test distribution is well described by the basis, then its anomaly score will be near zero; otherwise, it will be large. Note that, because we are dealing with a *standardized* distance metric, the anomaly score is unaffected by the scale and the spread of the input population. In other words, an anomaly score of twenty is comparable, even between servers with drastically different variances. An anomaly score less than a parameter `anom_thres` is considered non-anomalous; otherwise, it is classified as an anomaly of some severity. Section 4.7.5 explores the `anom_thres` setting in more detail.

If the test distribution is deemed to be normal (or non-anomalous), then it is added to the set of distributions used to define the normal basis. This is done to make the process adaptive to gradual shifts in the behavior of a server, whether because of seasonal changes in the demand placed on the server, the gradual aging of the server, or other reasons. At a high level, this is a "windowing" operation: the oldest day in the input set is dropped out of the window as the newest normal day is incorporated. The size of the window, `window_size`, is a parameter, which we will explore in Section 4.7.2. Once the input set is established, the normal basis is recomputed using PCA, as detailed above. On the other hand, if the test distribution is considered anomalous, it is not added to the normal set, and the current basis is reused.

An astute observer might ask: why must you use PCA? Why not just compute the Mahalanobis distance on the original b dimensional input data and test distribution? The (generalized) Mahalanobis distance incorporates the correlation among dimensions, so that the test

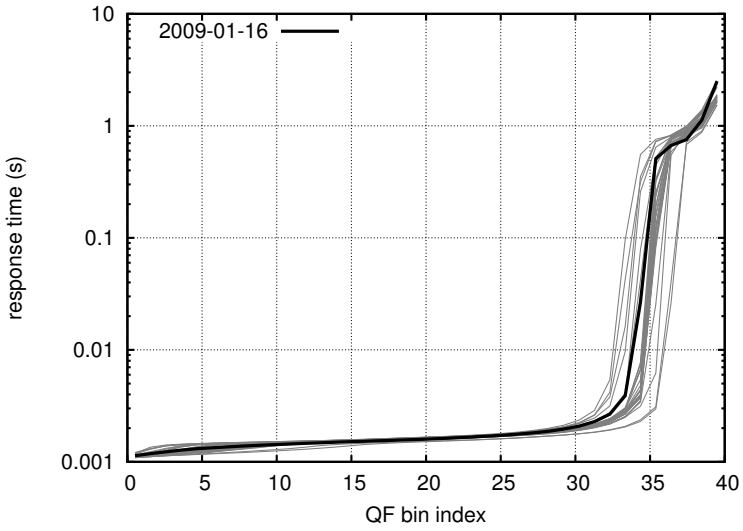


Figure 4.5: Quantile functions for example server, plotted on a logarithmic scale, with the anomalous QF highlighted

distribution is not penalized multiple times for being on the edge of some envelope. However, the Mahalanobis distance uses the covariance matrix, which will be singular if the number of samples is less than the number of bins—the so-called "high-dimension low-sample-size (HDLSS) situation". The HDLSS situation is common in my application, and a singular covariance matrix will yield infinite or otherwise nonsensical Mahalanobis distances, so we use PCA as an approach more robust to the HDLSS situation.

4.5 Tutorial

To build intuition, I will now give an extended example of applying the discrimination methods of the previous section. Even though the heart of the discrimination method is actually implemented as a set of matrix operations, I will consider the method as though it were an iterative process. In this tutorial, I will walk through each step of an anomaly test that discovers a non-intuitive anomaly. This tutorial will help build intuition for PCA, how I use it in my methods, and what sort of anomalies PCA is designed to find.

The anomalous distribution (represented as a QF) is plotted in Figure 4.5 along with the input "normal" distributions with which it is to be compared. It is not obvious why this distribution might be considered anomalous, because it falls entirely within the envelope of the

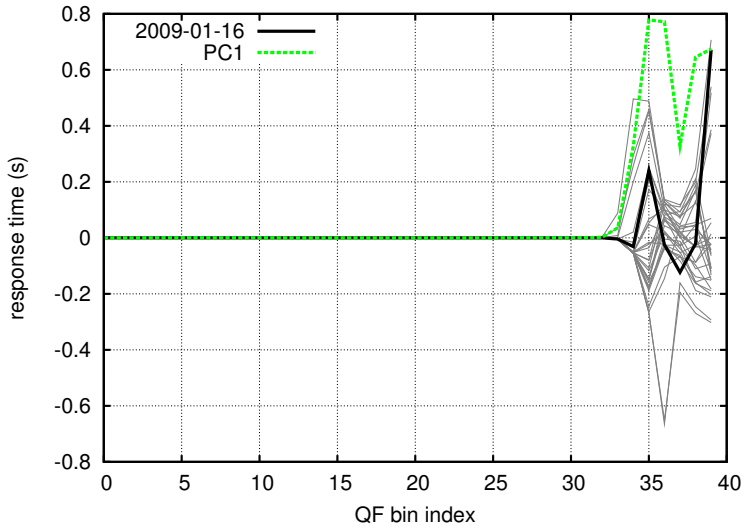


Figure 4.6: The zero-centered quantile functions, with principal component 1

normal distributions (even in the last bin). As we will discover, this distribution is nevertheless considered anomalous because, intuitively, PCA is more concerned with matters of shape than matters of envelope coverage.

Figure 4.6 plots the zero-centered normal QFs; that is, the QFs with the mean of each bin subtracted, which are the input to PCA. The same mean vector is subtracted from the anomalous QF, even though it was not used when computing the mean vector. Finally, the first principal component (PC 1) is shown as well. This principal component was computed as the *shape* that explains the greatest amount of deviation from the center-line. This is exactly equivalent to the explanation offered in the previous section: if the 40-bin QFs are considered points in 40-dimensional space, then PC 1 is the vector describing the direction in which the points are the most spread out. Note that the scale (or "size") of PC 1 as it is plotted here is irrelevant; only the shape matters. The actual size of PC 1 is 1, because all principal components have a size of 1 (in terms of the L_2 norm); I have enlarged it in this plot merely so its shape could be seen. In fact, flipping it about the center-line (by multiplying it by -1) would not change the outcome, because the shape remains the same (or, if you prefer, the vector still describes the same direction in 40-dimensional space). One other observation: the fact that PC 1 does not cross the center-line indicates that QFs typically do not cross the center line, either.

Figure 4.7 shows the *projection* of each QF onto PC 1. Clearly, each projection has the

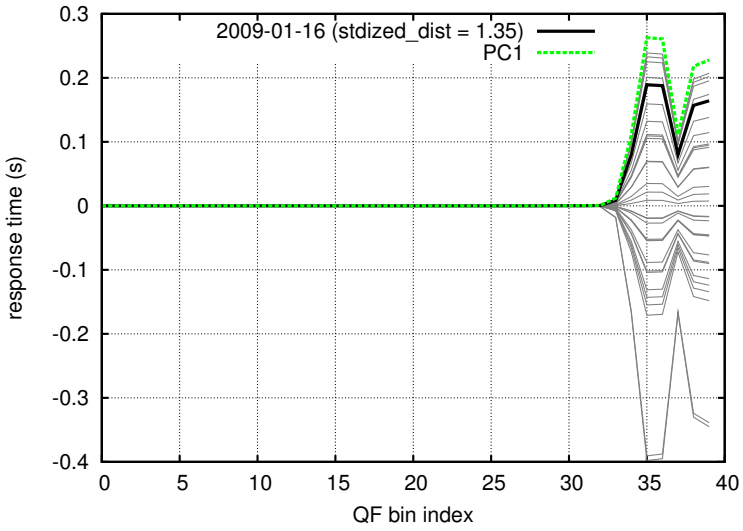


Figure 4.7: The projection of each QF onto principal component 1

same shape as PC 1. (Again, PC 1 is scaled here by an arbitrary amount, merely to make it visible on the plot.) The only variable is the scale of each projection, which is actually the extent to which PC 1 describes the QF. If, for example, a QF has the exact same shape as PC 1, except for a single bin that is the opposite of what it should be, then the resultant scale is zero, and the projection of the QF onto PC 1 would yield the zero line. Spatially, if we consider PC 1 as a line (though it is actually a vector), then the projection of a QF (a point) is the perpendicular projection of the point on the line—just as point is projected onto the X-axis in a Cartesian coordinate system to get the X-coordinate. Thus, spatially speaking, each curve plotted here is a 40-dimensional point somewhere on the axis described by PC 1, and the only variable is the distance along the axis from the origin, which describes the scale of each curve.

This scale, or the coordinate along PC 1, is used to compute the first part of the anomaly score. Recall from the previous section that the anomaly score is the standardized distance of the test point from the origin, or the sum-of-squares of the number of standard deviations from the test point to the origin along each axis. The standard deviation along the first axis is the standard deviation of the set of coordinates of the input QFs, the square of which has already been computed as the eigenvalue corresponding to PC 1. Therefore, to get the contribution of this axis to the anomaly score, I square the coordinate of the test point and divide it by the eigenvalue. The result, as indicated in the figure, is 1.35—far short of the anomaly threshold

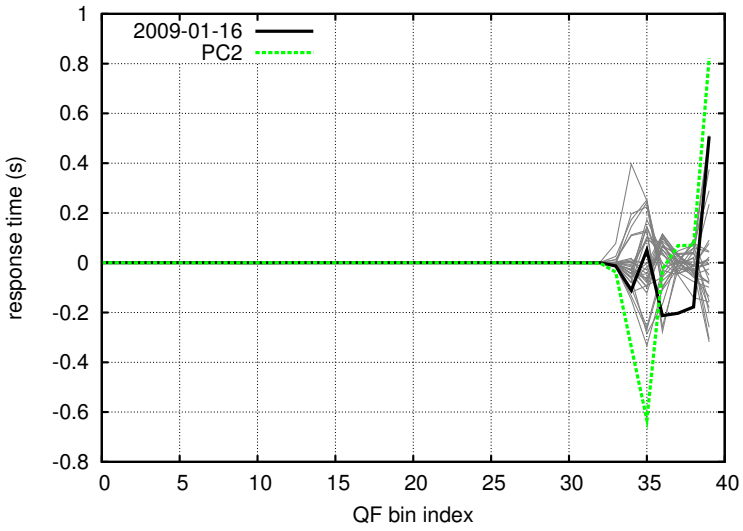


Figure 4.8: Residual 1: the leftover of each QF, after subtracting its respective projection onto PC 1 from the QF itself

of 50 (the threshold I use for most of my results, as discussed in Section 4.7).

Next, we subtract the projection of the QF from the QF itself to get the first *residual*, shown in Figure 4.8. The entire process essentially iterates on this set of distributions. The shape describing the most variation (or the direction of greatest spread in 40-dimensional space) is computed and shown as PC 2. Note that PC 2 is guaranteed to be orthogonal to PC 1, because PC 1 has already been subtracted from each QF. The figure shows that PC 2 crosses over the zero line, which indicates that there is a negative correlation between bins 34 and 35 and bin 39 of the first residual.

Then, the projection of residual 1 onto PC 2 is shown in Figure 4.9. Again, all of the projections have the same shape; the only difference is the scale. PC 2 does a fairly good job of describing the first residual of the test distribution, and the contribution of PC 2 to the anomaly score is 1.84. That brings the sum up to 3.19—still far short of the anomaly threshold of 50.

Figure 4.10 shows residual 2, the leftover information after subtracting the first *and* second projections from each QF. The third principal component is, again, the most representative shape of these curves.

Figure 4.11 shows the projection of residual 2 onto PC 3. PC 3 does a good job of describing the shape of the second residual of the test point, so the contribution of PC 3 to the anomaly

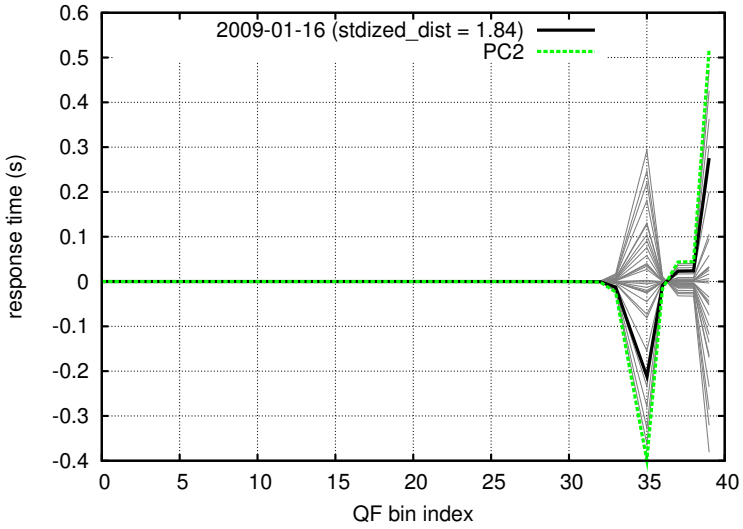


Figure 4.9: The projection of each residual 1 onto PC 2

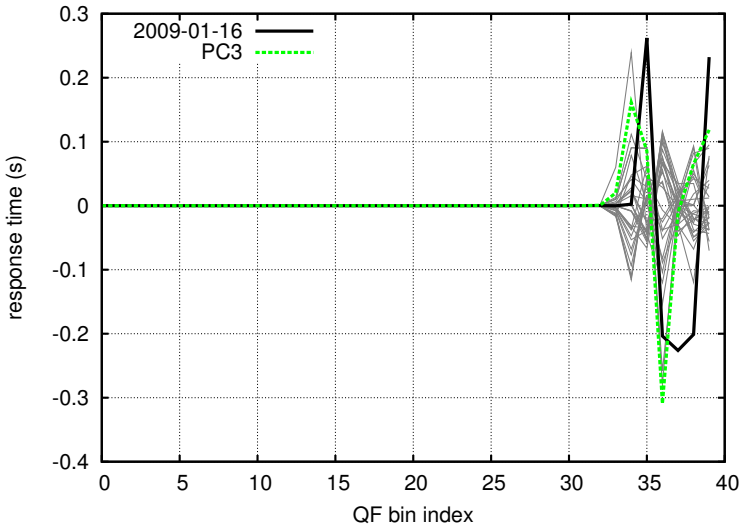


Figure 4.10: Residual 2: the leftover after subtracting the first and second projections from each QF

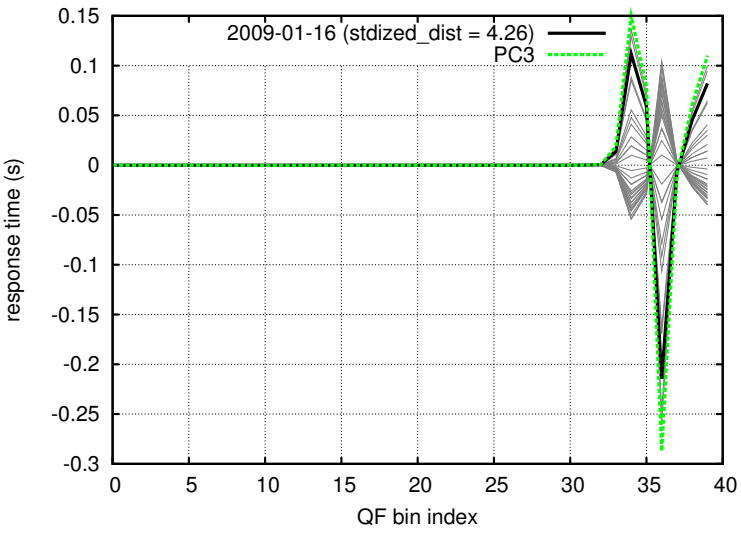


Figure 4.11: The projection of each residual 2 onto PC 3

score is 4.26. Recall that this is the *square* of the number of standard deviations the test point is from the origin along PC 3.

Figure 4.12 shows the third residual, and Figure 4.13 shows the projection of this residual onto PC 4. Again, PC 4 does a decent job of explaining the shape of the residual for the test point. The fourth projection adds 3.37 to the anomaly score.

Figure 4.14 shows the fourth residual. Notice that, at this point, the test distribution clearly has more energy than any of the "normal" distributions. Thus far, the principal components have explained less of the overall variation of the test point than they do of the normal distributions. This is a clue that the test point will end up being flagged as anomalous, because that energy has to go somewhere. If it isn't accounted for by the principal components, then it will be counted as projection error. Also, note how the fifth principal component seems to do a very good job of describing the shape of the test residual.

Indeed, it does such a good job of describing the test residual that the standardized distance component of PC 5 is 60.50, as shown in Figure 4.15. This means that the projection of the test residual along PC 5 is $\sqrt{60.50} = 7.78$ standard deviations away from the origin. This is easily enough to put the anomaly score over the threshold of 50.

In this case, there are five principal components in the normal basis, so we are done iterating. However, we also need to augment the anomaly score by incorporating the projection error, or

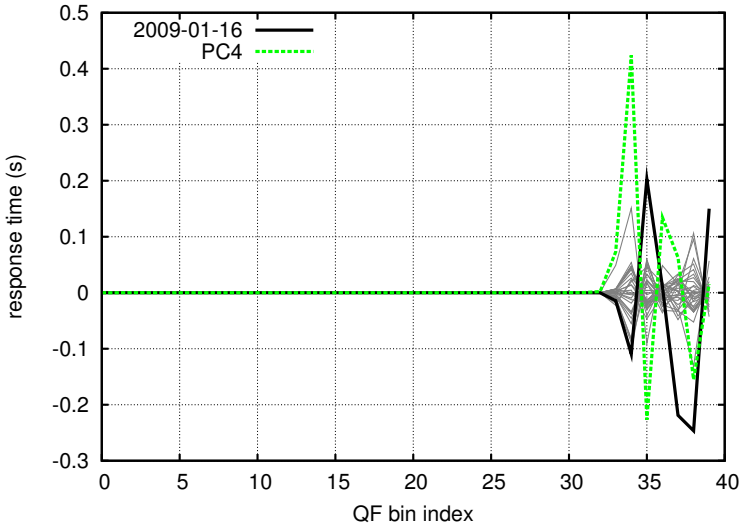


Figure 4.12: Residual 3: the leftover after subtracting the first through the third projections from each QF

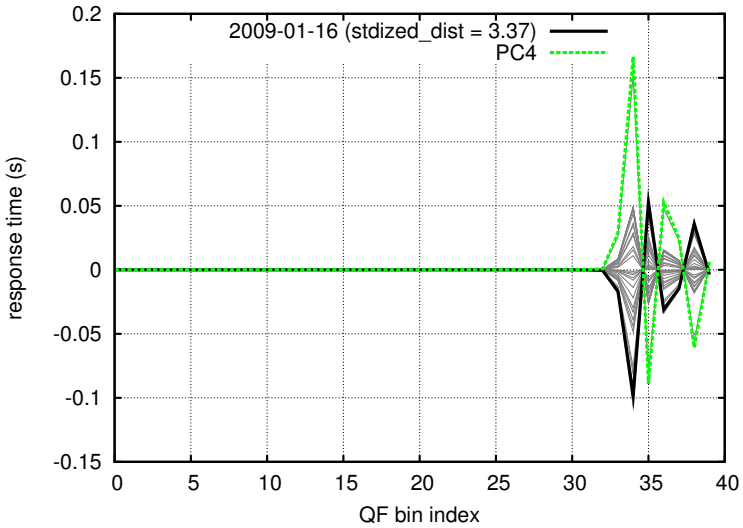


Figure 4.13: The projection of residual 3 onto PC 4

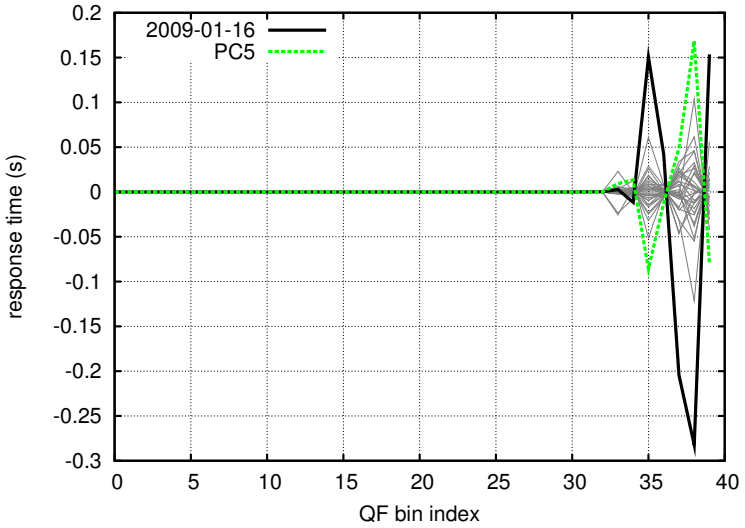


Figure 4.14: Residual 4: the leftover after subtracting projections 1–4

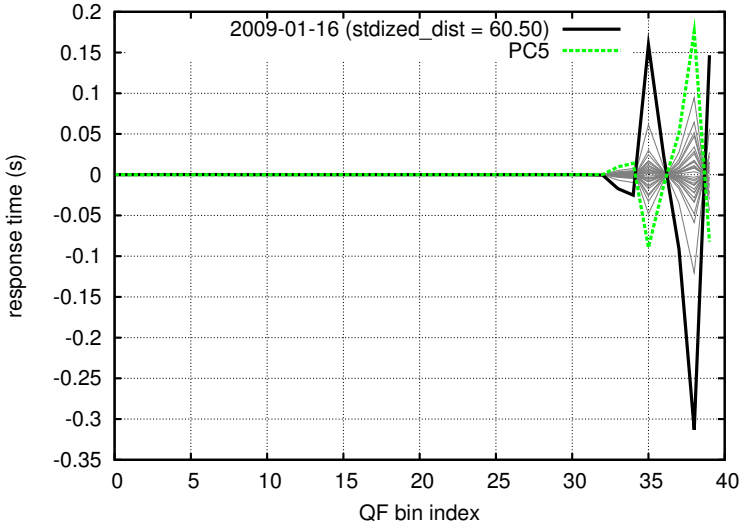


Figure 4.15: The projection of residual 4 onto PC 5

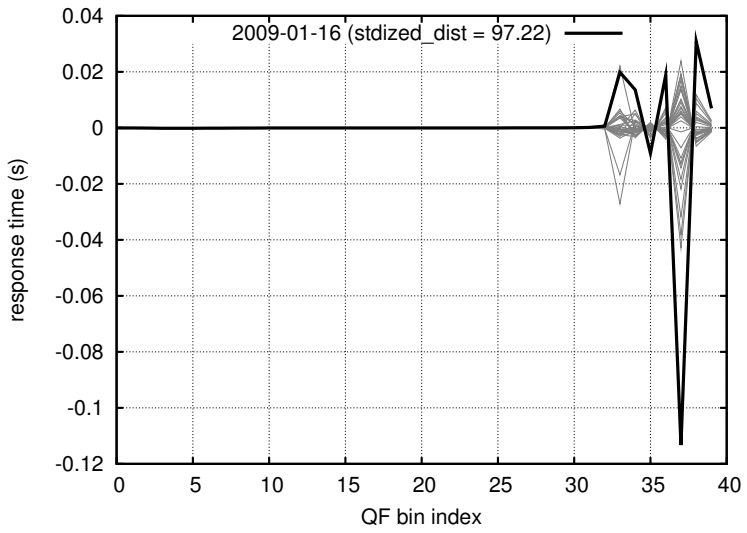


Figure 4.16: Residual 5: the final residual after subtracting the projection of each QF into the PCA-defined normal subspace

the extent to which the test point is *not* described by the normal basis. Figure 4.16 shows the fifth residual, which is the difference between the original point (or QF) and the projection of the point (or QF) onto the 5-dimensional normal basis. The projection error is the Euclidean distance from the original point to its normal projection—in other words, the square root of the sum of squares of the bins of residual 5. Note that even the normal points have a fifth residual, because the normal basis only explains the distributions to a tolerance of `basis_error` (in this case 0.01). The projection errors of the normal points are considered as a set. The mean and variance of this set is computed. In a parallel way to the previous anomaly component calculations, the mean is then subtracted from the projection error of the test point, and the result is squared and divided by the variance to get the final component of the anomaly score, which in this case is 97.22. The total anomaly score is 168.54, which is more than the anomaly threshold of 50, so the test point is indeed considered anomalous.

In essence, the distribution was considered anomalous because, at a key point towards the last QF bin, the distribution did not increase as fast as the normal distributions, and then, in the last two bins it increased faster (refer back to Figure 4.5). In other words, it was anti-correlated with the normal basis. This meant that the principal components derived from the normal basis poorly explained the test distribution, which led to the high anomaly score.

4.6 Bootstrapping

Bootstrapping is the process of selecting a set of input data to use as the definition of normal behavior for a server. The output of bootstrapping is referred to as the "initial basis" of normality for the server. As mentioned in the previous section, this basis will change over time as distributions deemed to be normal are added to it and others are dropped. Bootstrapping is done as early as possible after measurements are collected, and it is complicated by the potential presence of abnormal values. The bootstrapping process, also called cross-validation, is intended to be as automatic as possible, but it is impossible to get it "right" (a subjective determination) in all cases. For example, the input set to be validated might have half or more abnormal distributions. Therefore, while automatic methods are used by default, not only are the parameters adjustable by the network manager, but the network manager can also select the initial basis manually.

First, a number of distributions are accumulated. The number is a parameter that I call `tset_size`. I must choose `tset_size` as a reasonable trade-off between the delay of making anomaly decisions and the stability of the training set. Too small a value of `tset_size` means that the basis can fail to account for the natural, non-anomalous variation of the server's operation. Section 4.7.3 discusses the choice of this parameter in more detail.

The bootstrapping process, shown in pseudo-code in Algorithm 14, is composed of k iterations (where k is a parameter), for i from k down to 1 (lines 2–10). At each iteration, all combinations of i distributions among the training set are left out, *i.e.* $C(n, i)$ total combinations, where n is the number of distributions and C is the combination function $\frac{n!}{((n-i)!i!)}$. For each combination, the i left-out samples are individually tested (line 8) against the basis defined by the remaining $n - i$ samples (line 7), and the average anomaly score for that combination is saved (line 9) as a measure of how dissimilar the left-out samples are from the rest. If the maximum for all the metrics (line 11) is greater than the anomaly score threshold (the same one used in the previous section for determining whether a single point is anomalous), then the bootstrapping process ends, defining the initial basis as that formed by the input set minus the maximally dissimilar combination (lines 13–15). Otherwise, if no combination yields an average score above the threshold, the process repeats for $(i - 1)$. If none of the k iterations gives any

Algorithm 14 The bootstrapping process, where n is the number of distributions, k is the maximum number of days to exclude, and X is the array of n distributions. `getAllCombinations` returns all combinations of k items selected from n total items as an array of arrays.

```
1: bootstrap(n, k, X) :
2: for i ← k downto 1:
3:     C[][] ← getAllCombinations(n, k)
4:     for j ← 1 to length(C)
5:         Z ← {X_C[j][1], X_C[j][2], ...} // let Z be the distributions excluded from X
6:         Y ← {X − Z} // let Y be everything but the excluded distributions
7:         N ← formNormalBasis(Y)
8:         S[] ← anomScores(N, Z) // test each item of Z against the normal basis N
9:         A[j] ← mean(S)
10:    endfor
11:    m ← indexOfMaximum(A)
12:    if A[m] > anom_thres:
13:        Z ← {X_C[m][1], X_C[j][2], ...}
14:        Y ← {X − Z}
15:        return formNormalBasis(Y)
16:    endif
17: endfor
18: return formNormalBasis(X)
```

value above the threshold, then all n samples are substantially similar (or else more than k samples are dissimilar from the rest), and the initial basis is formed by all n samples (line 18).

The parameter k was chosen to be 2 by default, so that behavior that was consistently different on a particular day of the week would be included. Otherwise, if *e.g.* Saturdays were consistently different for a server, then they might be flagged as anomalous every week, resulting in an annoyingly high anomaly detection rate.

As k increases, the overall computation required for bootstrapping increases exponentially, because we must test all possible combinations of k. This is another reason to keep k small. However, there are probably optimizations that we could make, *e.g.* based on the overall distance of a point from the mean. Finding an optimized bootstrapping process that yields the same (or substantially similar) results is a topic for future work.

4.7 Parameters

This section describes the various parameters used in the anomaly detection approach, as introduced in Section 4.4. I provide intuition for each of these parameters, and I show the

results of various parameter settings on a corpus of UNC server data. While there is no "correct" setting for any of the parameters, each of them affects the ability of my methods to identify anomalies of interest to a network manager.

The corpus is the `adudump` data from the Dataset 2 for a set of "heavy-hitter" server address/port pairs. I must consider the port because a single system with a single IP address can serve multiple applications with different server processes operating on different ports. The address/port pairs were selected based on the data in the dataset through February 18^{th}. Address/port pairs were included in the corpus if there was an average of at least twenty ADUs per minute over the course of a day, for at least half of the days since the beginning of the dataset (December 9^{th}), or 34 days. There were 229 such address/port pairs on 202 unique server addresses.

To show the effect of each parameter, I ran the anomaly detection analysis of Section 4.4 on every server in the corpus. The parameter settings were chosen according to the default settings given here, except when they are varied to demonstrate their effect. An initial training set of `tset_size` = 21 days was chosen. As discussed in Section 4.6, the training set was bootstrapped, identifying at most two days to exclude when creating the normal basis. The basis is formed with a `basis_error` setting of 0.01, and then the anomaly score was computed. If the anomaly score was less than `anom_thres` = 50, then the test day is added to the window (the maximum size of which is `window_size` = 60), and the new basis is computed at the same setting of `basis_error`. Lastly, I use `num_bins` = 40.

4.7.1 Basis Error

Recall from Section 4.4 that the `basis_error` is the maximum amount of relative variation explained by the "error" dimension. In other words, principal components (which, remember, are sorted by importance) are added to the normality-defining basis until there is only `basis_error`$*100$ percent of the overall variance remaining.

Intuitively, the `basis_error` parameter controls the "tightness" of the fit of the basis. The closer `basis_error` is to zero, the more fully the basis will describe the data upon which it is based, and the lower the expected error. When another distribution is tested against the basis, it must also have a low projection error, or else it will be flagged as anomalous.

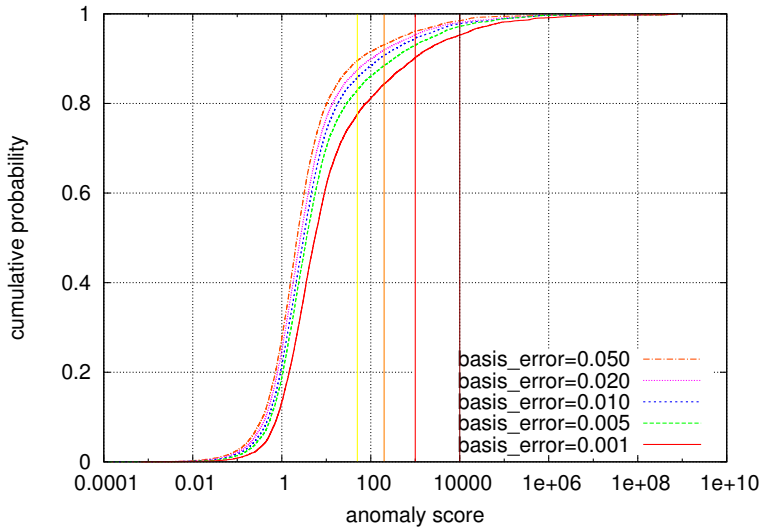


Figure 4.17: Anomaly scores as `basis_error` varies, for all the servers in the corpus.

Figure 4.17 shows a CDF of all of the anomaly scores for all servers, at various settings of `basis_error`. Various anomaly thresholds (which we will use in later results) are shown as vertical lines. In general, the higher the setting of `basis_error`, the more tolerant the basis is, and the lower the anomaly score.

4.7.2 Window Size

The `window_size` parameter defines the maximum number of days to use when defining a new basis. After a new distribution is determined by the anomaly test to be non-anomalous, it is added to the "window" of most recent non-anomalous days from which to define a normal basis. If the addition of this distribution pushes the window over the limit defined by `window_size`, then the oldest distribution in the window is dropped.

Intuitively, the `window_size` setting is the amount of "memory" of the system. The larger the setting of `window_size`, the more tolerant the basis is to different distributions, because a larger variety of distributions comprises the normal basis. If there is a natural "lull" during certain periods (such as the winter break at UNC), the window should be large enough that the distributions before the lull are not forgotten.

Figure 4.18 shows the distribution of anomaly scores for the server address/port pairs in the corpus at various settings of `window_size`. Note that the window starts out, before the first

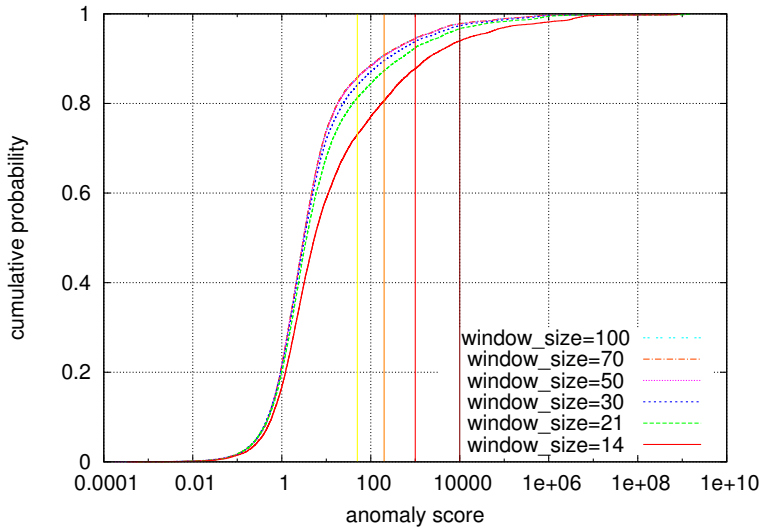


Figure 4.18: Anomaly scores as `window_size` varies, for all the servers in the corpus.

anomaly test, at a size of `tset_size` minus whatever days were excluded by the bootstrapped (at most two). I am only testing the corpus through March 9^{th}, which is 81 days or about 12 weeks from the beginning of the corpus. This is a major reason why a `window_size` of 100 gives almost identical results to a `window_size` of 70: there are a maximum of 21 days that the basis could possibly be different, and probably fewer because of the existence of anomalies (which do not grow the window).

4.7.3 Training Set Size

The `tset_size` parameter defines the number of distributions required for a training set, which will then be bootstrapped (see Section 4.6) to determine whether, out of the first `tset_size` distributions, some are unusual and should be excluded when computing the initial basis of normality. We would like this parameter to be as small as possible, because it determines the number of days we must wait after starting data collection before making any real anomaly decisions.

Figure 4.19 shows the distribution of the anomaly scores at various settings of `tset_size`. In general, the larger the setting of `tset_size`, the lower the anomaly score. Clearly, a setting of `tset_size` = 5 is too small, because the anomaly scores are unreasonable. One might argue that a setting of 50 is too large: since at most two distributions can be excluded, some unusual

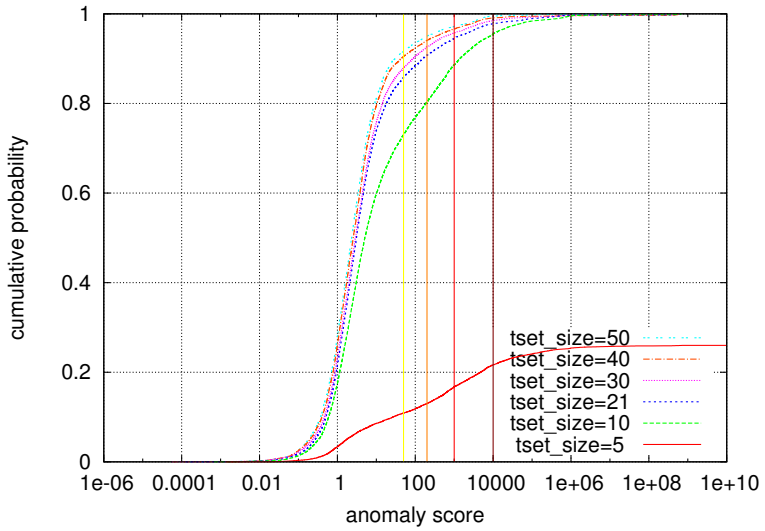


Figure 4.19: Anomaly scores as `tset_size` varies, for all the servers in the corpus.

distributions might be included in the initial basis of normality, thus reducing the anomaly score of legitimate anomalies.

I have not explored the effect of changing the maximum number of days that bootstrapping can exclude from the training set. With a `tset_size` = 21, I argue that 2 is a reasonable value for this parameter, so that, if one day per week is consistently unusual, then it will include that day in what is considered normal. Furthermore, increasing this value incurs exponential computational cost unless a more clever approach is invented. But regardless of algorithmic efficiency, it is important to keep in mind that bootstrapping is inherently a very subjective task; we cannot know *a priori* what distributions, out of the first few, will prove to be normal. For this reason, I have provided a way for the network manager to reset the basis, even manually deciding which distributions should be considered normal, as discussed in Section 4.11.

4.7.4 Number of Bins

The `num_bins` parameter controls how a distribution of *e.g.* response times is represented as a quantile function (QF). The more bins, the greater the granularity of the QF, the higher dimensional space each QF resides in, and the more information is preserved when summarizing the distribution as a QF.

Figure 4.20 shows the distribution of anomaly scores at various settings of `num_bins`. In

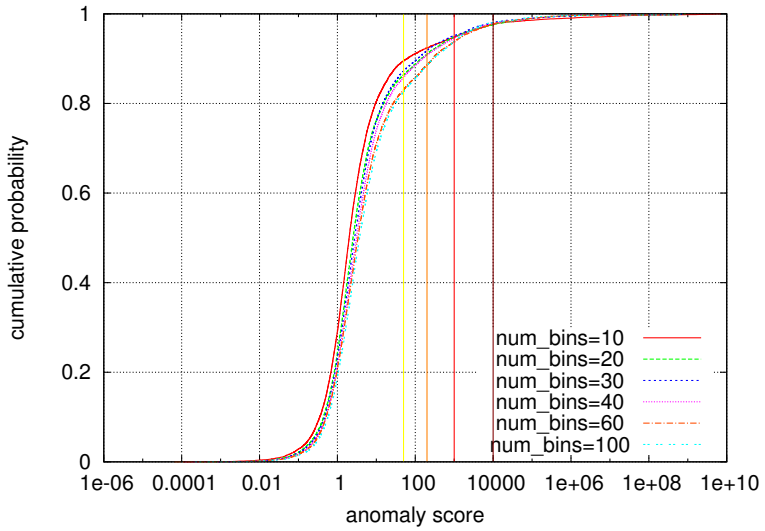


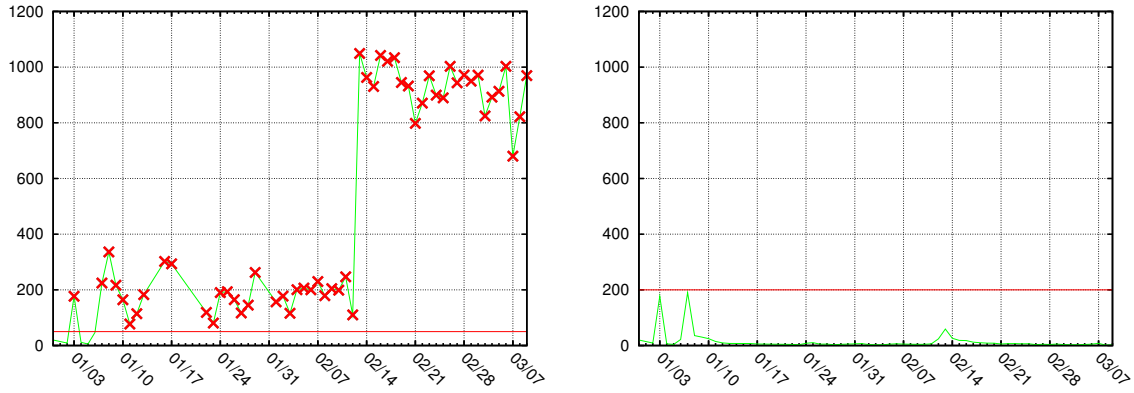
Figure 4.20: Anomaly scores as `num_bins` varies, for all the servers in the corpus.

general, the fewer bins, the lower the anomaly score. This occurs for two reasons. First, recall that the number of summands in the anomaly score is equal to the number of dimensions in the basis, plus one for the projection error. Therefore, as the number of dimensions increases, the anomaly score tends to increase as well. When there are fewer bins overall, the number of dimensions in the normal basis tends to be smaller, hence the generally lower anomaly score. The other reason is that, if the population of distributions varies in a nonlinear fashion in the b-dimensional space, the effects of the nonlinear variation will increase as b (or `num_bins`) increases. However, this is not a reason to choose a lower setting for the `num_bins` parameter, because the fewer bins, the less information is captured in the QF.

4.7.5 Anomaly Threshold

The `anom_thres` setting controls two things: the threshold of the anomaly score by which a distribution is judged as anomalous, and the decision of whether to include that distribution in the window of distributions used to define a normal basis. Actually, as we will see in the rest of this chapter, we will consider several different thresholds for the anomaly score that define a range of severities—so the latter effect is more significant.

Intuitively, the `anom_thres` parameter defines the "stability" or even "stubbornness" of the basis: the lower the `anom_thres` setting, the more resistant the basis is to change, because too



(a) `anom_thres` = 50 (b) `anom_thres` = 200

Figure 4.21: Anomaly scores for an example server at different settings of `anom_thres`

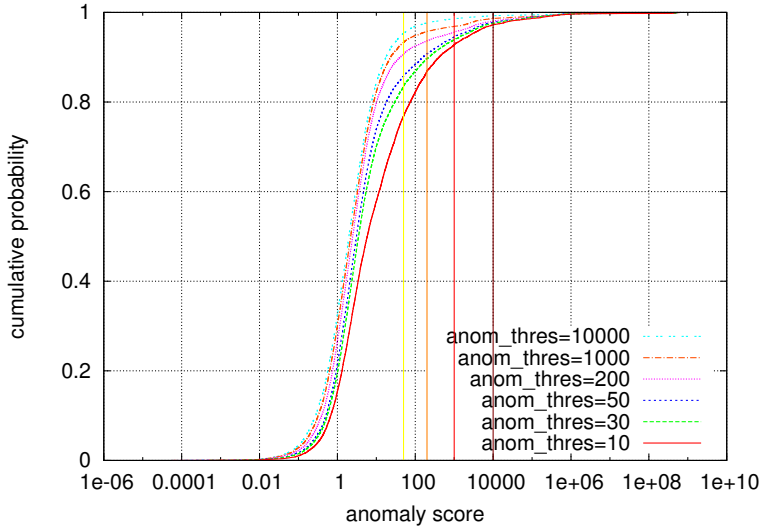


Figure 4.22: Anomaly scores as `anom_thres` varies, for all the servers in the corpus.

abrupt a change will not be added to the window used to compute the basis. If we desire to track only very gradual changes in the operation of a server, then we should set this parameter lower. If, on the other hand, we want to permit a relatively wide range of distributions to be considered non-anomalous, and only flag the egregious distributions as anomalous, then we should set this parameter higher. Figure 4.21 demonstrates this intuition with an example. With an anomaly threshold of 50, most of the distributions after a shift on January 7^{th} are considered anomalous, whereas an anomaly threshold of 200 allows the distribution from January 7^{th} to be considered normal, which causes all of the following distributions also to be considered normal.

Figure 4.22 shows the distribution of anomaly scores at various settings of `anom_thres`. In

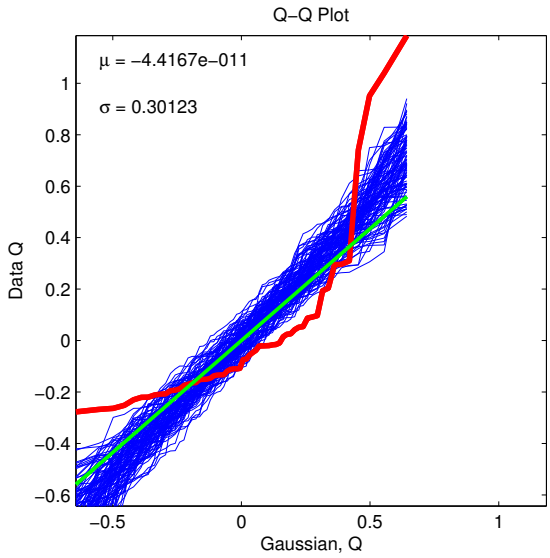


Figure 4.23: Quantile-Quantile plot of projection along first principal component for an example server. The projection is quite non-Gaussian.

general, the lower the setting, the higher the anomaly score. An example will help explain why this is the case. Consider two successive days of response times for a particular server that have the exact same shape. Let's say that the distribution of the first day yields an anomaly score of 150. If that is above the setting of `anom_thres`, then the day will not be added to the window, and the second day will yield the same score. On the other hand, if the anomaly score of 150 is below the threshold, then the first distribution will be added to the window, and the distribution will be part of the set of days used to define the normal basis, and the second distribution will yield a much lower anomaly score (although not necessarily zero). Thus, the higher the setting of `anom_thres`, the lower the anomaly score.

Under certain assumptions, we do not need to guess at a good setting of `anom_thres`. Recall that the distributions, represented as QFs, can be considered points in some high-dimensional space. The points are translated into the PCA-computed normal basis, which we will say uses u components to define normality. Thus the translated points now reside in u-dimensional space—plus one dimension for the error component (see Section 4.4). The mean of each coordinate in this $(u+1)$-dimensional space is zero, because we initially zero-centered the input data before doing PCA by subtracting the mean. Recall that the anomaly score is the sum-of-squares distance of the zero-centered test point in this space, in units of the standard deviation along

each component. An equivalent way of measuring the "standardized distance" of the test point is to first divide the i^{th} coordinate of each zero-centered point by σ_i, where σ_i is the standard deviation along coordinate i, for $i = 1 \ldots u$. Considering each coordinate independently, each of them now has zero mean and a standard deviation of one, and the anomaly score is simply the square of the Euclidean distance to the test point.

If the QFs are distributed according to a $(u+1)$-variate Gaussian (or normal) distribution—in other words, if the distribution of values along each coordinate conforms to a Gaussian distribution—then the anomaly score corresponds exactly to the definition of a χ^2 distribution with $(u+1)$ degrees of freedom. Thus, if this assumption holds, we can then assign a *likelihood* of seeing the test point, where the likelihood ranges from zero to one, zero being impossible and one being certain. This would give us a nicer approach to the anomaly threshold, because it would be independent of u, whereas the current anomaly score tends to be larger as u is larger.

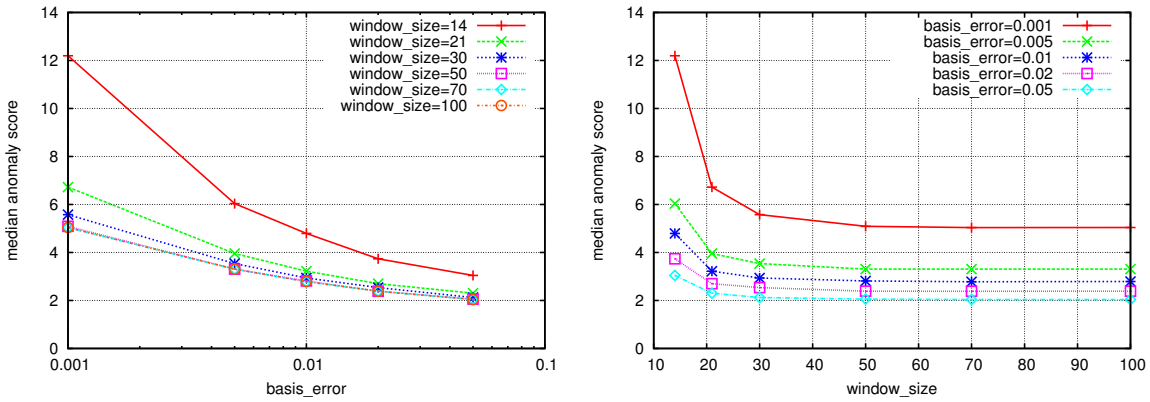
Unfortunately, this assumption frequently does not hold for our data. For example, see Figure 4.23, which is a QQ (or quantile-quantile) plot of the first coordinate for a server. The QQ plot shows the quantile of the input distribution against the quantile of a Gaussian distribution with the same μ and σ^2 as the input distribution, to determine whether the input distribution is well described by a Gaussian distribution. If it is, the line will appear roughly linear, and it will appear closer to linear as the number of points in the distribution increases. In this QQ plot, there are 100 blue lines of random Gaussian distributions, each with a sample size equal to the number of samples in the input distribution. In this way, the blue lines provide an intuitive sense of the typical range of variation when given an input distribution that truly is Gaussian. The red line is the QQ line of the input distribution (*i.e.* the first coordinate of the QF points as translated into the normal basis). It is clearly very different, falling well outside the envelope of the blue lines; thus, the input data was not Gaussian. Hence, we cannot use the χ^2 distribution when deciding whether a point was anomalous, and we must continue to use the more general anomaly score.

4.8 Joint Parameter Exploration

The previous section described the five parameters used in the anomaly detection approach and explored the effect of varying each parameter by itself. We saw a set of default settings used throughout the discussion, and at most one parameter differed from the default setting at a time. Although a full analysis of the entire five-dimensional parameter space is infeasible, this section explores the parameter space more fully by jointly varying two parameters at a time. Each of the ten subsections presents results from varying two of the five parameters together, using the same methodology and corpus as seen in the previous section.

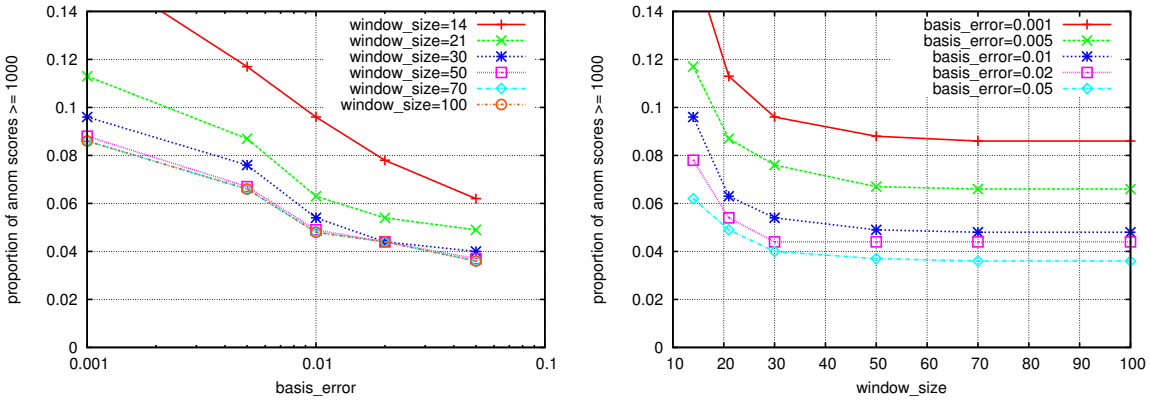
Note two things about the analysis in this section. First, it was done *after* choosing settings for each of the parameters, so these results did not guide the choice of parameter settings. Second, this analysis is not intended to be a comprehensive explanation of the parameters; rather, like the previous section, this section merely provides intuition about the interaction between parameters.

It is useful to recall the chief trends that the previous section uncovered. As `basis_error` increases, the normal basis is less strict, and anomaly scores are generally lowered. As `window_size` increases, the number and variety of distributions forming the normal basis increases, so it is more likely that a new distribution is similar to at least one of the input distributions, and thus anomaly scores are lowered. As `tset_size` increases, the more likely it is that one or more dissimilar distributions will be added to the normal basis (since, at present, at most two distributions are excluded), thus increasing the variation in the normal basis and making anomalies less likely. As `num_bins` increases, the dimensionality of the input basis tends to increase, which tends to increase anomaly scores. As `anom_thres` increases, the less likely an anomaly score will be flagged as anomalous, and thus the more likely it will be added to the normal basis, which in turn gives future similar distributions a lower score. In summary, increasing `num_bins` tends to *increase* anomaly scores, whereas increasing `basis_error`, `window_size`, `tset_size`, or `anom_thres` tends to *decrease* anomaly scores.



(a) by `basis_error` (b) by `window_size`

Figure 4.24: Median anomaly score as `basis_error` and `window_size` vary



(a) by `basis_error` (b) by `window_size`

Figure 4.25: Proportion of anomaly scores greater than or equal to 1000, as `basis_error` and `window_size` vary

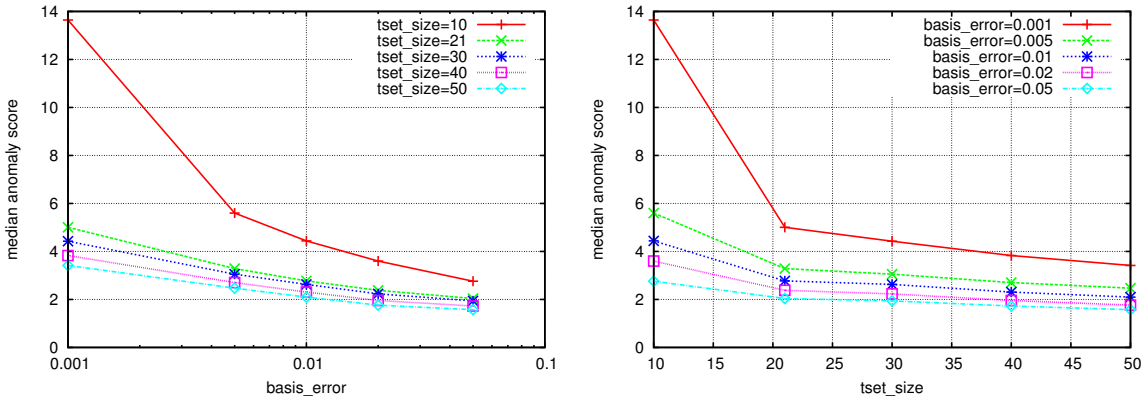
4.8.1 Basis error and window size

First, I explored the effect of jointly varying `basis_error` and `window_size` from the default values. This is effectively exploring a two-dimensional plane of settings within the five-dimensional parameter space. I chose `basis_error` settings of 0.001, 0.005, 0.01, 0.02, and 0.05. I chose `window_size` settings of 14 (or two weeks), 21 (three weeks), 30, 50, 70, and 100. I will use these settings throughout this section.

The median anomaly score among the entire corpus (*i.e.* the median of all the scores for all servers on all days) is plotted in Figure 4.24. The two subplots show the same data but varied by alternate parameters. Each point represents the median anomaly score over the entire corpus. For example, the upper left-most point, plotted in red in both plots, is a result from running the anomaly detection method with `basis_error` = 0.001 and `window_size` = 14 over each of the 229 servers in the corpus (each with over 80 day-long distributions), and then taking the median of over 18,000 anomaly scores. Likewise, the next point along the red line in the left plot is the median over the entire corpus when `basis_error` = 0.005 and `window_size` = 14.

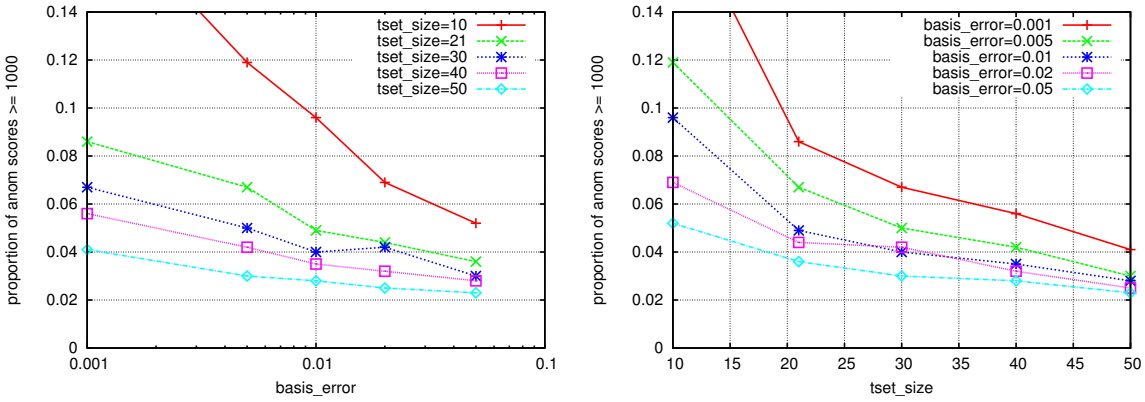
The overall trends are clearly seen: both `basis_error` and `window_size` yield lower lower anomaly scores as they are increased, as discussed in the previous section. In addition, I observe that `window_size` has a diminished effect as `basis_error` is increased, whereas `basis_error` has a relatively constant effect for `window_size` greater than 20. Note also that, between a `window_size` of 70 and `window_size` of 100, most of the individual anomaly tests comprising a point in these results are the same; that is, the input to each normal basis is the same, all other parameters holding constant. The two do not diverge until the seventy-first distribution is added to the normal basis, at which point the former setting requires that the oldest distribution roll out of the window but the latter setting includes the oldest distribution. Therefore, it is no surprise that the results for these two settings are largely similar.

I also plotted in Figure 4.25 the proportion of anomaly scores in the corpus that were greater than 1,000. Recall that the default anomaly threshold I use is actually 50, so 1,000 represents a large anomaly score. It will be the third of four anomaly thresholds that I use in the next section, used to define so-called "error" anomalies. The overall trends are the same as in the



(a) by `basis_error` (b) by `tset_size`

Figure 4.26: Median anomaly score as `basis_error` and `tset_size` vary



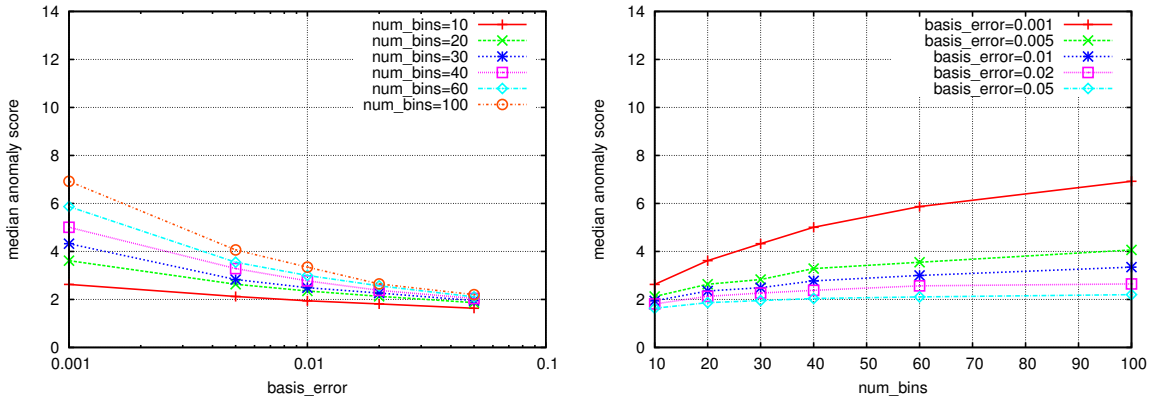
(a) by `basis_error` (b) by `tset_size`

Figure 4.27: Proportion of anomaly scores greater than or equal to 1000, as `basis_error` and `tset_size` vary

previous figure. It is also notable that, for many settings, a high proportion of anomaly scores are above the threshold.

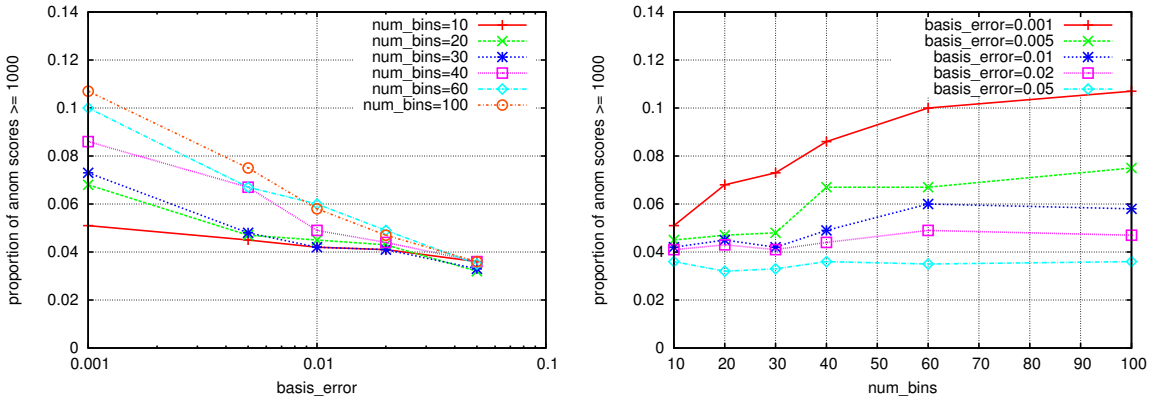
4.8.2 Basis error and training set size

Next, I jointly varied `basis_error` and `tset_size`. I chose `tset_size` settings of 10, 21 (or three weeks), 30, 40, and 50. I also generated results for `tset_size` equal to 5, but these results suffered from an as yet undiagnosed bug in which the anomaly scores often exceed 10^{30}, so I have not included these results here. See Figure 4.26. I again observe that increasing either setting yields lower anomaly scores. Also, both parameters are more sensitive when the other setting is lower. These observations remain true for the proportion of anomaly scores greater



(a) by `basis_error` (b) by `num_bins`

Figure 4.28: Median anomaly score as `basis_error` and `num_bins` vary



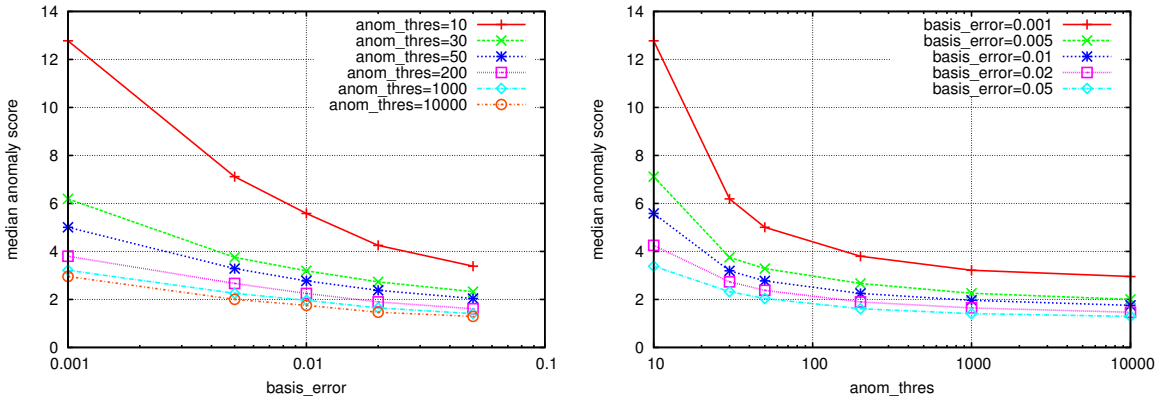
(a) by `basis_error` (b) by `num_bins`

Figure 4.29: Proportion of anomaly scores greater than or equal to 1000, as `basis_error` and `num_bins` vary

than 1,000, as shown in Figure 4.27.

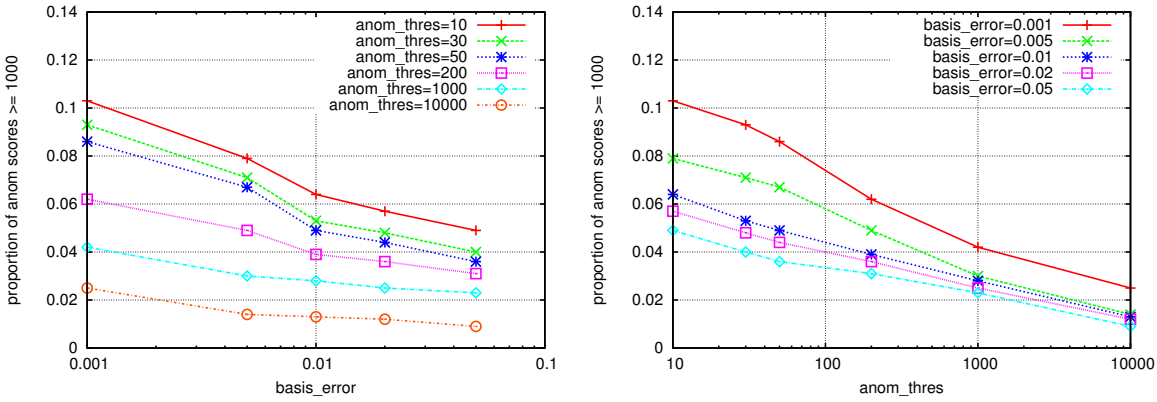
4.8.3 Basis error and number of bins

Next, I varied `basis_error` together with `num_bins`. I chose `num_bins` settings of 10, 20, 30, 40, 50, and 100. The median anomaly score is plotted in Figure 4.28. The observations related to `basis_error` are consistent with previous observations: as `basis_error` increases, the sensitivity of the other parameter decreases, and the median anomaly score also decreases. Conversely, these effects occur when `num_bins` decreases. Figure 4.29 shows the proportion of anomaly scores greater than 1,000, which concurs with these observations (although its curves are not as smooth).



(a) by `basis_error` (b) by `anom_thres`

Figure 4.30: Median anomaly score as `basis_error` and `anom_thres` vary



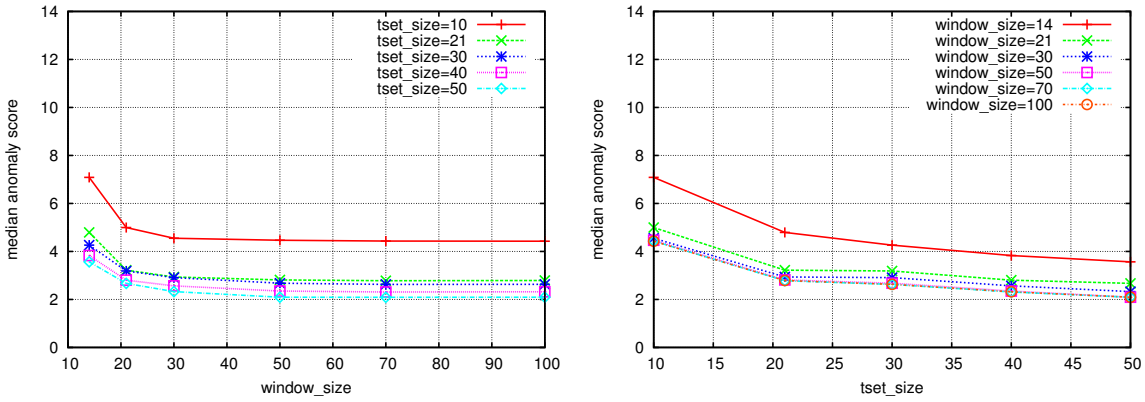
(a) by `basis_error` (b) by `anom_thres`

Figure 4.31: Proportion of anomaly scores greater than or equal to 1000, as `basis_error` and `anom_thres` vary

4.8.4 Basis error and anomaly threshold

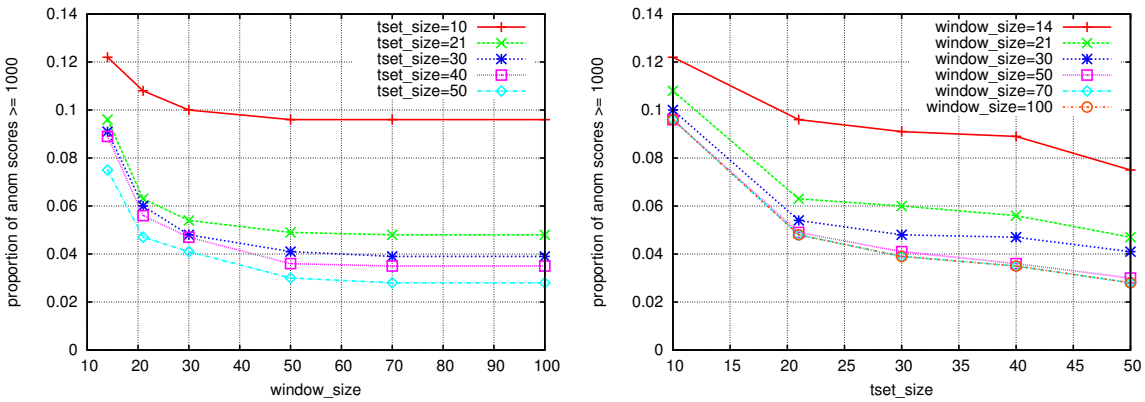
Next, I jointly varied `basis_error` and `anom_thres`, and the results are shown in Figure 4.30. I chose `anom_thres` settings of 10, 30, 50, 200, 1000, and 10000. As in Section 4.8.2, increasing either setting yields lower anomaly scores, and both parameters are more sensitive when the other settings is lower.

Figure 4.31 shows the proportion of anomaly scores exceeding 1,000 for each parameter setting. These results exhibit the widest variation seen thus far. The proportion seems to be quite sensitive to `anom_thres`. Again, the reason for this is that, as `anom_thres` increases, the anomaly test is more willing to admit unusual distributions into the normal basis. As the normal basis incorporates these unusual distributions, its variation increases, and future anomaly tests



(a) by `window_size` (b) by `tset_size`

Figure 4.32: Median anomaly score as `window_size` and `tset_size` vary



(a) by `window_size` (b) by `tset_size`

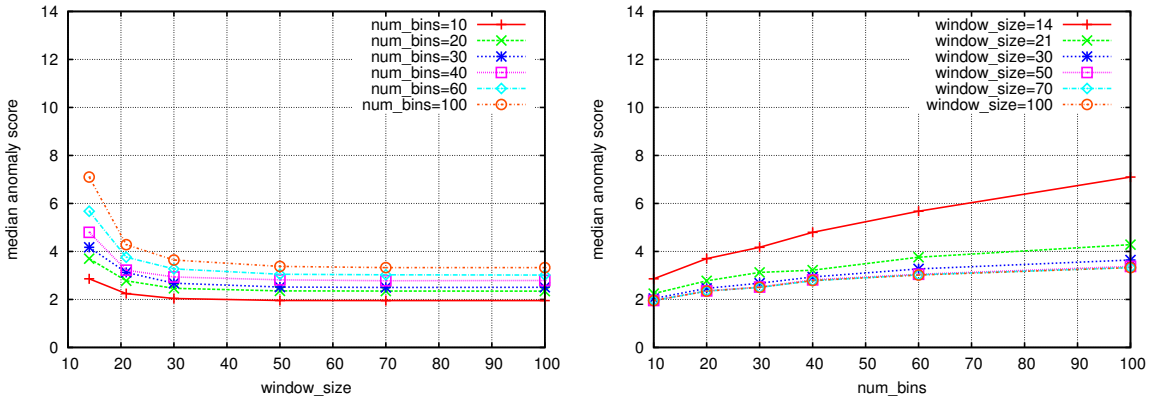
Figure 4.33: Proportion of anomaly scores greater than or equal to 1000, as `window_size` and `tset_size` vary

thus result in lower anomaly scores.

4.8.5 Window size and training set size

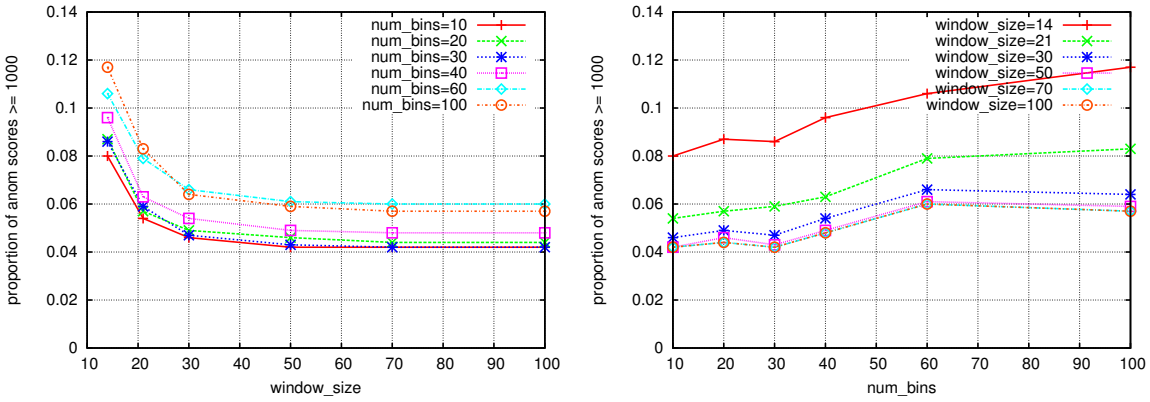
Next, I jointly varied `window_size` and `tset_size`, and the results are shown in Figure 4.32. This time, although both parameters yield lower anomaly scores as they are increased, the effect of each parameter appears to be relatively independent of the other parameter. I also note that, as in Section 4.8.1, `window_size` settings of 50, 70, and 100 are relatively constant. This is because the corpus has 81 days, so the normal bases and anomaly tests in such cases are similar—especially between settings of 70 and 100.

Figure 4.33 shows the proportion of anomaly scores exceeding 1,000. By this metric, the



(a) by `window_size` (b) by `num_bins`

Figure 4.34: Median anomaly score as `window_size` and `num_bins` vary



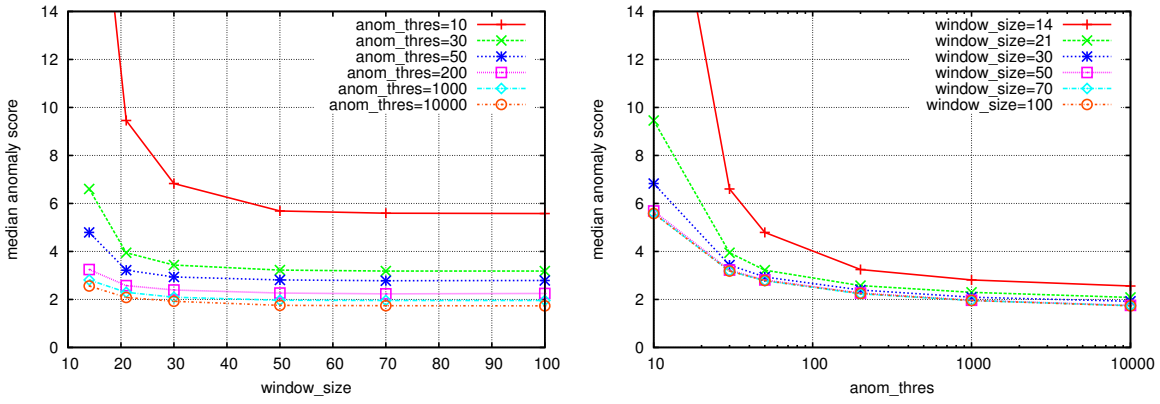
(a) by `window_size` (b) by `num_bins`

Figure 4.35: Proportion of anomaly scores greater than or equal to 1000, as `window_size` and `num_bins` vary

results for `tset_size` = 10 and for `window_size` = 14 are quite different from their neighboring settings.

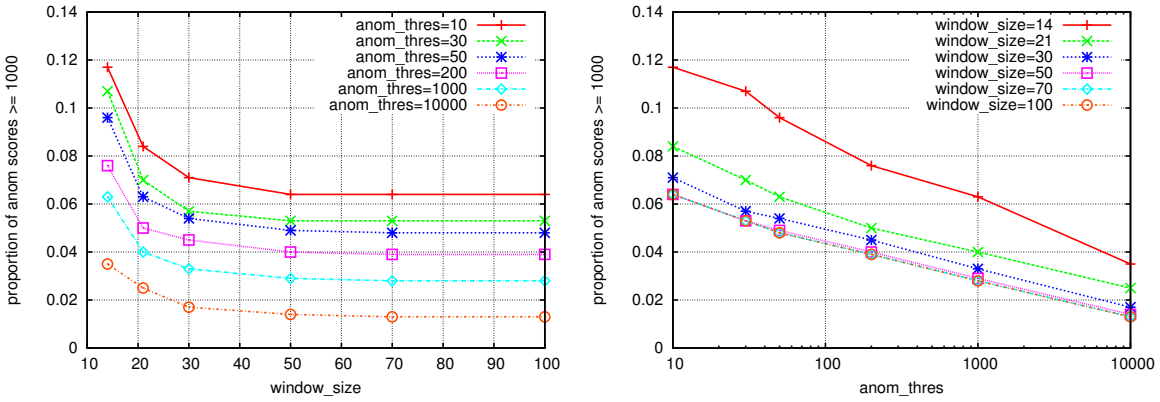
4.8.6 Window size and number of bins

Next, I examined the joint effects of `window_size` and `num_bins`. The median anomaly scores are shown in Figure 4.34 and 4.35. The observations from previous sections still hold here, and there are no surprises. The anomaly scores decrease as `window_size` increases and as `num_bins` decreases. The effect of `num_bins` is relatively constant for most settings of `window_size`, and the `window_size` is more sensitive as `num_bins` increases.



(a) by `window_size` (b) by `anom_thres`

Figure 4.36: Median anomaly score as `window_size` and `anom_thres` vary



(a) by `window_size` (b) by `anom_thres`

Figure 4.37: Proportion of anomaly scores greater than or equal to 1000, as `window_size` and `anom_thres` vary

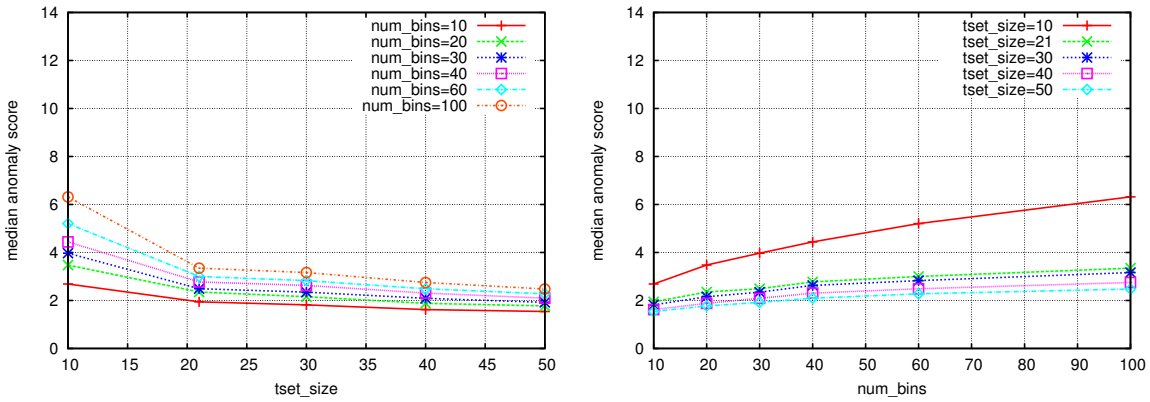
4.8.7 Window size and anomaly threshold

Next, I jointly varied `window_size` and `anom_thres`, and the median anomaly score is shown in Figure 4.36. There is a large outlier when `window_size` is 14 and `anom_thres` is 10, but other than that, the results are as we would expect based on previous observations.

The proportion of anomaly scores greater than 1,000 is shown in Figure 4.37. Again, judging by this metric, the `anom_thres` has the largest effect of any parameter.

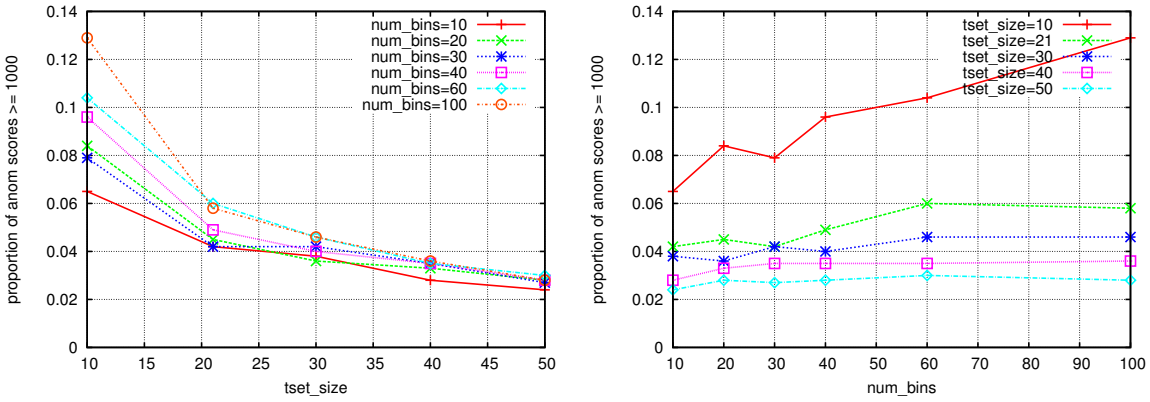
4.8.8 Training set size and number of bins

Next, I jointly varied `tset_size` and `num_bins`, and the results are shown in Figure 4.38. Again, there are no surprises here. However, the proportion of anomaly scores exceeding 1,000,



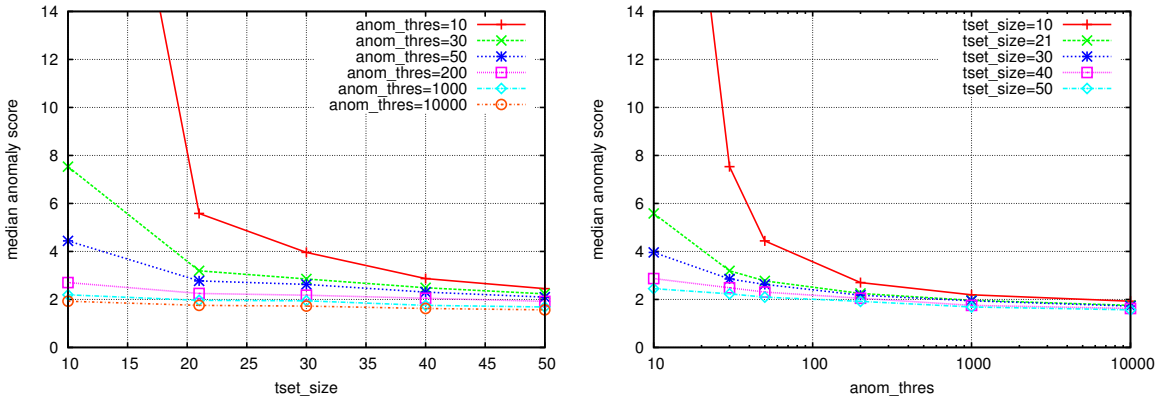
(a) by `tset_size` (b) by `num_bins`

Figure 4.38: Median anomaly score as `tset_size` and `num_bins` vary



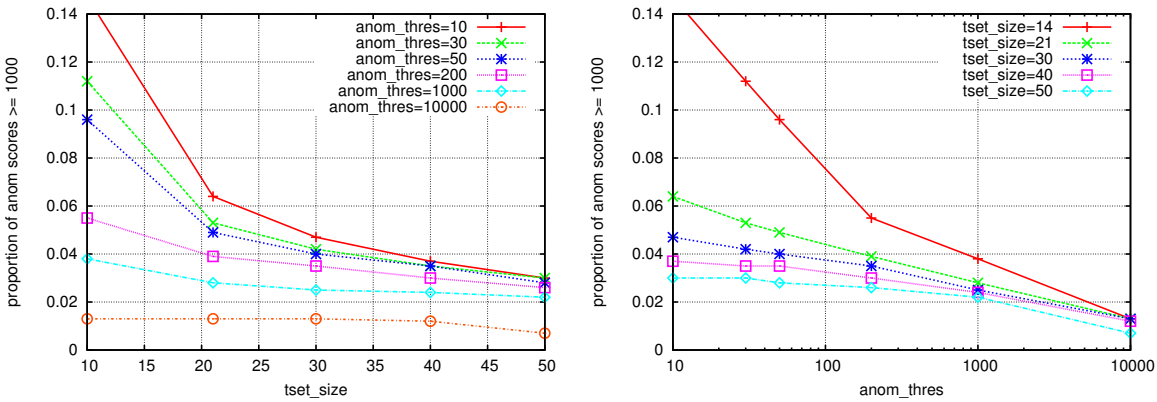
(a) by `tset_size` (b) by `num_bins`

Figure 4.39: Proportion of anomaly scores greater than or equal to 1000, as `tset_size` and `num_bins` vary



(a) by `tset_size` (b) by `anom_thres`

Figure 4.40: Median anomaly score as `tset_size` and `anom_thres` vary



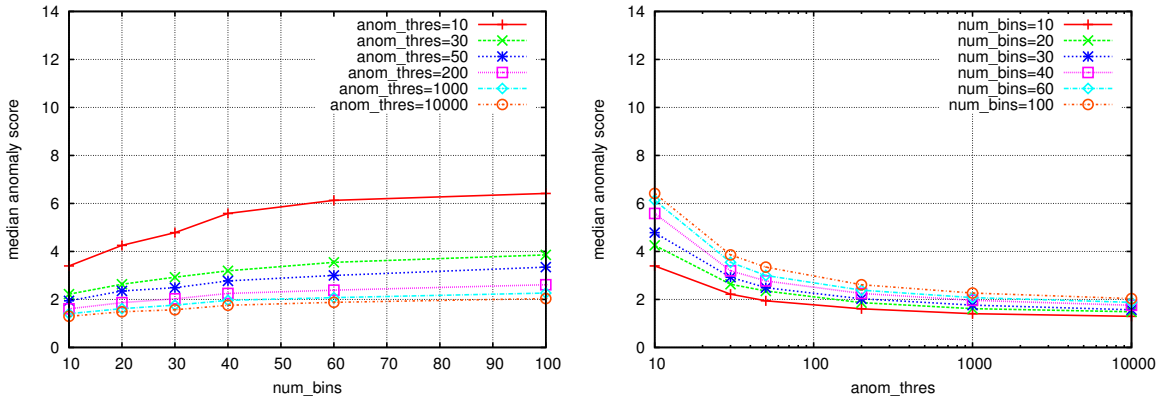
(a) by `tset_size` (b) by `anom_thres`

Figure 4.41: Proportion of anomaly scores greater than or equal to 1000, as `tset_size` and `anom_thres` vary

as shown in Figure 4.39 is somewhat surprising in that, in most cases, the proportion for `num_bins` = 30 is outside the bounds established by its neighboring settings. I do not know why.

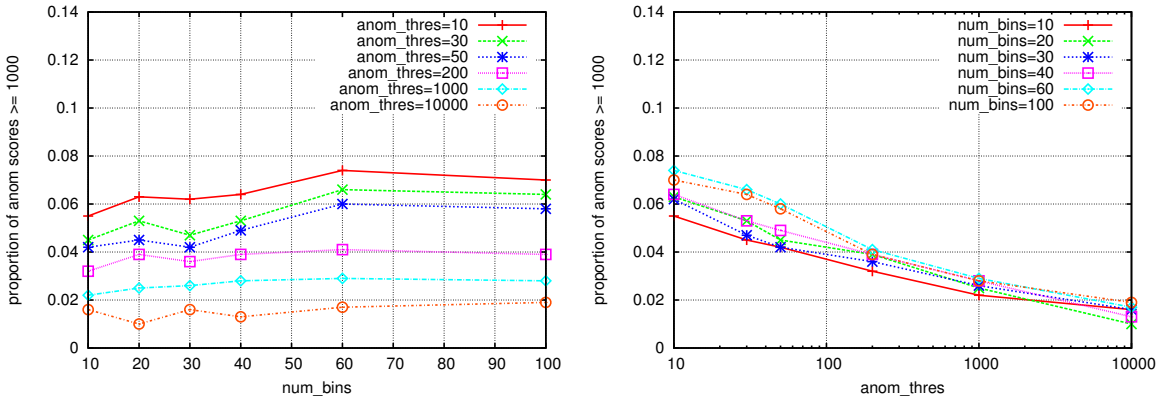
4.8.9 Training set size and anomaly threshold

Next, I jointly varied `tset_size` and `anom_thres`. See Figure 4.40. Note the extent of the outlier that occurs when `tset_size` is 10 and `anom_thres` is 10. Also, it seems that `tset_size` hardly matters when `anom_thres` = 10,000. Figure 4.41 shows similar results for the proportion of anomaly scores exceeding 1,000 per joint setting.



(a) by `num_bins` (b) by `anom_thres`

Figure 4.42: Median anomaly score as `num_bins` and `anom_thres` vary



(a) by `num_bins` (b) by `anom_thres`

Figure 4.43: Proportion of anomaly scores greater than or equal to 1000, as `num_bins` and `anom_thres` vary

4.8.10 Number of bins and anomaly threshold

Lastly, I varied `num_bins` together with `anom_thres`. Figure 4.42 shows the median anomaly score. There are no surprises here, either. Figure 4.43 is surprising because of the extent to which the lines cross each other, when plotted by `anom_thres` setting. I do not know why this occurs.

4.8.11 Conclusions

In general, I have shown in this section that anomaly scores tend to increase as `basis_error`, `window_size`, `tset_size`, and `anom_thres` decrease and as `num_bins` increases. Typically, increasing one parameter meant that other parameters had a decreased effect, although `num_bins`

is again the exception; increasing `num_bins` increased the effect of other parameters. The `window_size` parameter had relatively little effect beyond 30, so a setting of 30 seems reasonable for it. The `basis_error` parameter gives reasonably stable results for settings beyond 0.005. `tset_size` should be as small as possible so that a normal basis can be constructed as soon as possible, and a setting of 21 seems reasonable. `num_bins` gives sometimes unpredictable results, but a setting of 40 bins seems as reasonable as any other. `anom_thres` effects the methods the most strongly, and I will use various settings (including 50 and 1000) in this dissertation.

Determining the "right" settings for a particular network will be an ongoing exercise. In general, any discussion of a "correct" setting will require an authoritative labeling of certain distributions as normal or anomalous. Then, the parameters can be tuned to attempt to meet as many of the authoritative labels as possible.

4.9 Timescales

Thus far, I have only discussed the anomaly detection methods at a generic level, not specifying the time-scale of the input distributions. All of the examples discussed so far have been at a time scale of one day (*i.e.* distributions of response times over a 24-hour period). Because a QF can be computed, in principle, for any distribution, my methods are in fact agnostic to the time-scale, as long as there are enough samples in each distribution (a caveat I will discuss later). This section shows how to use the anomaly detection methods at different time-scales. The goal here is to *quickly* detect performance issues as soon as possible after they occur.

To illustrate this issue, I will examine at different time-scales a performance issue that occurred in January of 2009 with one of the web servers in the UNC network. This particular web server is involved with registration-related tasks. In particular, this web server provides an interface to a course search engine, which searches for courses matching certain criteria (*e.g.* subject=COMP and availability=OPEN), and lists the results in a web page.

Figure 4.44 shows the anomaly scores resulting from the same day-by-day analysis that we saw in the previous section, in which response time distributions from a full, 24-hour day comprise the individual QFs upon which the anomaly detection methods operate. The figure

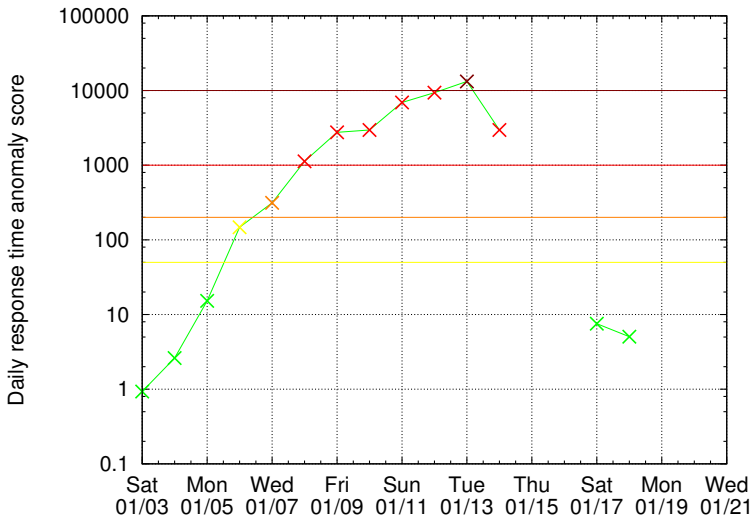


Figure 4.44: Anomaly score for web server. Outages resulting in incomplete days are represented by breaks in the line.

shows that the anomaly score steadily increases from Saturday, January 3^{rd} until it reaches a `critical` anomaly on Tuesday, January 13^{th}.

Instead of grouping 24 hours of response times into a distribution, I have aggregated only 6 hours, for a "quarter-day" time-scale. In this analysis, the response times occurring from midnight to 6 AM were considered group A, 6 AM to noon is group B, noon to 6 PM is group C, and 6 PM to midnight is group D. For a given day, each group was compared only with distributions from the same group, comprising four distinct anomaly detection tests per day (each with its own basis). In other words, all non-anomalous group A distributions formed a group A basis used in the anomaly test for the group A distribution from the current day, and likewise for groups B, C, and D. I used this approach because the response times, along with many other traffic characteristics, are not stable, but have diurnal patterns, fluctuating consistently through the course of a day, so it would therefore be unwise to compare *e.g.* noon traffic to midnight traffic.

The results of the quarter-day analysis are shown in Figure 4.45. Each quarter-day block is color-coded according to its anomaly score, as shown in the legend. Quarter-day blocks are colored black if there was a data collection outage for at least part of the block (*i.e.* the `adudump` process running on the monitor failed for some reason). The figure gives us greater resolution when viewing the overall issue. For example, we can now see that the early morning hours

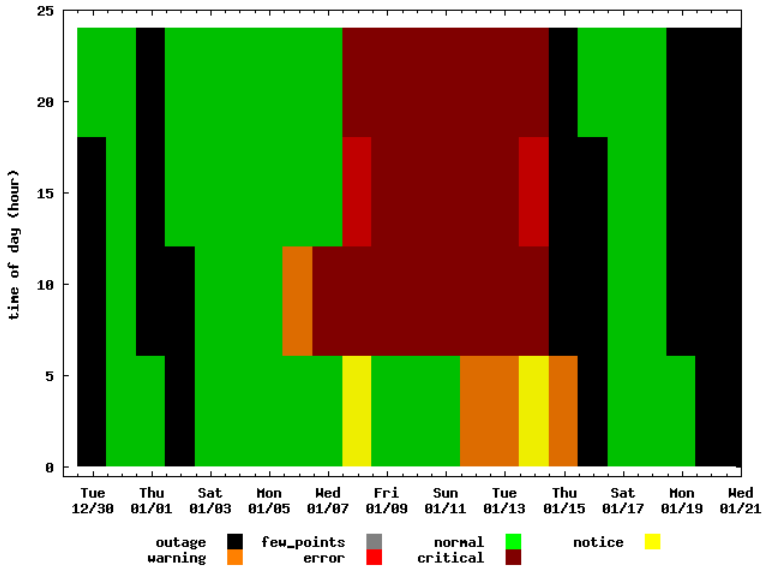


Figure 4.45: Time-scale analysis for web server on quarter-day distributions.

are often non-anomalous (and, in general, less anomalous) during the issue than are the other hours. Furthermore, the anomaly scores are in the critical area for the entire day except the early morning hours.

For even greater resolution, see Figure 4.46. The idea is the same as the previous figure, except that instead of grouping response times by quarter-day, I group them by one hour periods. There is one additional feature of this figure: there are two blocks colored grey, which signifies that there were too few response times in the distribution to perform the anomaly detection. This situation occurs when there are fewer samples (or response times) than bins in the QF, and I discuss this limitation in Section 4.12.

The hourly time-scale results in greater resolution, but it does so at a price. Figure 4.46 shows that it also is a *noisy* analysis: there are many hours flagged as relatively severe anomalies whose neighbors are not anomalous, and there does not seem to be much stability as to whether each time of day is anomalous. Furthermore, many anomalies appear at the hourly time-scale for which the quarter-day analysis gives no indication. One could imagine a set of heuristics based on the hourly analysis to decide when to actually alert a human; for example, alert if five of the last eight hours were at least a `error` anomaly, or if seven of the last eight were at least a `warning` anomaly. However, these would add needless complexity and parameters to the anomaly detection system.

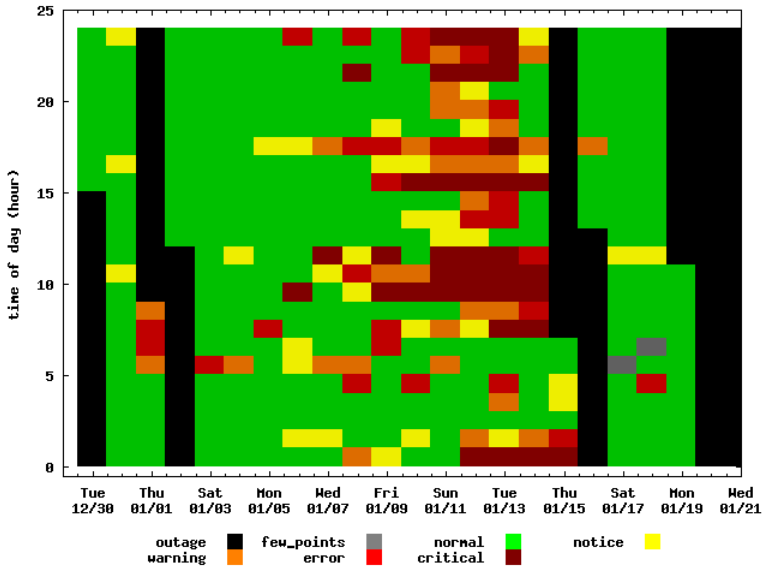


Figure 4.46: Time-scale analysis for web server on hourly distributions.

The hourly time-scale is noisy because there are relatively few response times in each distribution. This increases the effects of sampling error and results in an analysis that we have less confidence in. Ideally, we would like a method that reacts quickly to new response times yet does not suffer from sampling error.

I have made steps towards this ideal by implementing a sort of rolling-window time-scale. I aggregate response times over a 24-hour period, yet I advance this 24-hour window every 15 minutes. This solution achieves both robustness to sampling error as well as a quick reaction to new response times. Figure 4.47 shows the results of this "rolling-day" analysis. The anomaly scores increase and decrease more gradually, and in general the time-series is more stable. At the same time, however, long response times quickly influence the anomaly score. Note that, as the grey blocks indicate, I do not test a rolling-day distribution unless it is entirely unaffected by outages.

The rolling-day analysis can potentially introduce more computational overhead than could keep up with the incoming data. A naive approach would involve joining 96 quarter-hour distributions, every 15 minutes. I will outline a better approach, and the one I actually used, here. Assign each quarter-hour distribution its respective number: the midnight until 12:15 AM distribution gets the number 0, the 12:15–12:30 AM distribution gets the number 1, and so on. Furthermore, consider these numeric labels in binary. Every second quarter-hour distribution

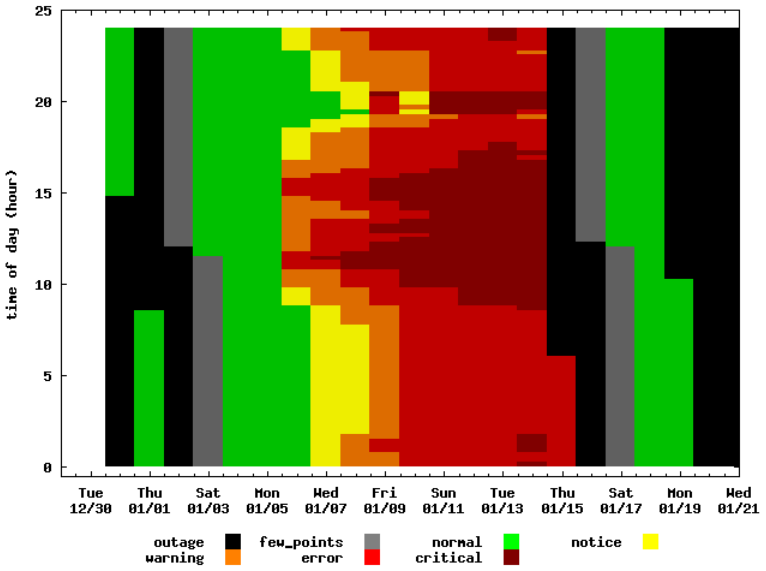


Figure 4.47: Time-scale analysis for web server on rolling-day distributions.

(*i.e.* those whose binary numeric labels end in "1"), combine the distribution with the previous distribution to form a half-hour distribution, and save the result. Every fourth distribution (*i.e.* those whose binary labels end in "11"), combine and save the half-hour distribution as before, and also combine and save the hour distribution. At every eighth distribution, do the same with half-hour, hour, and two-hour distributions. Continuing on in this fashion up to the 16-hour distribution, I form a hierarchy of saved combinations in the form of a binary tree. I can then exploit this tree to compute the rolling-day analysis by combining $\lceil log_2(96) \rceil = 7$ distributions (6 combinations), rather than 96 distributions (95 combinations), thus achieving the same results with a fifteen-fold reduction in the number of combinations and a large speed increase.

This section looked at a performance issue at several time-scales and introduced a "rolling-day" approach that achieves both robustness to sampling error and responsiveness to performance issues. I will use this rolling-day time-scale extensively in Chapter 5.

4.10 Ordinal Analysis

This section introduces what I call *ordinal* analysis. In an ordinal analysis, the emphasis is on the *order* of a request, response, or response time within its connection. For example,

consider all the response times for a particular server on a particular day. Instead of treating the entire set as a single distribution, I partition the set by the ordinal of the response time within its connection, so that the set of *e.g.* first response times from each connection comprises the distribution for ordinal 1, the set of second response times comprises the distribution for ordinal 2, *etc.*

In this way, I can account for the commonalities among connections imposed by the application protocol. For example, an SMTP dialog has some number of smaller requests and responses, followed by a large request that contains an email message. There are also commonalities imposed by a particular instance of a server. For example, some web servers, such as those hosting a portal or content management system, might dedicate the first one or two exchanges of each connection to authentication and session management, saving the real work for later exchanges, thus making it likely that the first one or two response times are much shorter than the third. In any event, an ordinal analysis gives visibility into the statistically "typical" connection, which can unlock deeper insights into the differences between the connections comprising normal versus anomalous traffic.

I use ordinal analysis several different ways in Chapter 5. First, I can simply compare the distributions of *e.g.* first response times (or second response times, or first request sizes, *et cetera*) between normal and anomalous traffic—assuming, as always, that there are a sufficient number of measurements. However, I can also do a full discrimination analysis on a particular response time ordinal. For example, perhaps the regular anomaly detection test (involving all the response times for a server) reveals an anomaly, but the first and third response times are not anomalous. Then we might conclude that the second response time is the cause of the problems. In general, the ordinal analysis can provide insight into the nature of performance issues, as we will see in Chapter 5.

4.11 Resetting the Basis

Sometimes, the traffic for a server shifts profoundly and permanently, resulting in continuous, unceasing anomalies. For example, see Figure 4.48, which shows that an SMTP server on campus began having `critical` anomalies on Tuesday, February 3^{rd}, 2009, and did not stop

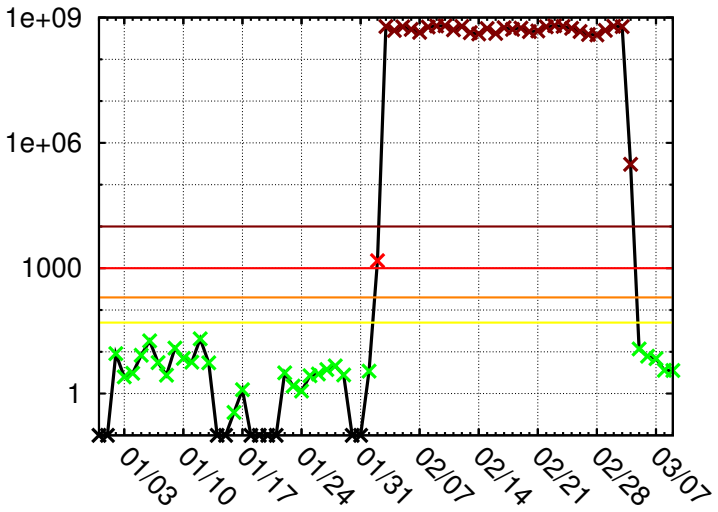


Figure 4.48: Anomaly score for an SMTP server.

for weeks. (I will analyze this server in more detail in Section 5.5.) In such cases, the network manager might wish to change the definition of normality for the server, essentially saying that what is normal for this server has changed. My methods allow for such a redefinition, which I call "resetting the basis".

Recall from Section 4.6 that the definition of normal is computed by doing PCA on a set of training distributions. Recall also that, although this definition of normal evolves over time, distributions deemed to be anomalous are not included. Typically, the set of training distributions will be bootstrapped (or cross-validated) to eliminate dissimilar distributions from the set. However, I also allow the network manager the ability to manually override the cross-validation step, giving him manual control over the selection of training distributions. If he simply includes some of the previously anomalous distribution in the set of distributions used as input to the PCA-based normal basis definition, other distributions that are anomalous in a similar way will then be considered normal. I will give an example of manually resetting the basis in Section 5.5.

4.12 Limitations

So far, we've only discussed doing the PCA-based discrimination scheme on response time distributions. However, there is no fundamental reason why the same scheme cannot be applied

to other types of data, such as request sizes, response sizes, or even the dialog length of a connection. Anything for which there are multiple observations per time period can conceivably be represented as a population of QFs.

However, distributions of request sizes, response sizes, and especially dialog lengths often have one characteristic not seen in the response time distributions: strong modality. This characteristic has two important effects worth considering. First, it provides a further reason for using QFs instead of PDFs or CDFs to represent the distributions. With PDFs and CDFs, the informational content (in terms of entropy) is inherently lower for strongly modal distributions, because, regardless of the choice of bin boundaries, there will be at least one bin with a "spike", or larger than average count. QFs, on the other hand, work just as well on strongly modal distributions: multiple quantiles having the same value does not interfere with the partitioning of the quantile domain, and the end result is still evenly spread among bins, even if the mode must be split among them.

The second effect of strongly modal distributions is that they tend to vary less linearly in the QF space. Recall from Broadhurst [Bro08] that the variation of parameters for parametric distributions results in variation that is linear in the QF space, whereas varying the weights of a mixture of multiple distributions results in a variation that is linear in the PDF space. A strongly modal distribution with one mode can be thought of as a mixture of a delta distribution (with δ equal to the mode) and an empirical distribution. The leftover empirical distributions vary among themselves, according to whatever unseen parameters govern their behavior, but the mixture of the empirical and the delta distribution varies too. The result is a population of distributions that probably varies in a nonlinear way in both QF and in PDF space.

The impact of nonlinear variation on the anomaly test method has not been fully explored. Indeed, while I showed in Section 4.2 that the QF representation varies *more* linearly than the other representations, it might still have significantly nonlinear variation. Practically speaking, this is not a huge concern; it merely means that the anomaly score will typically be higher for some servers. However, this is an area for future exploration.

Another fundamental limitation of my methods occurs when there are too few data points. This is simply sampling error, and would be a problem regardless of the methods I choose. However, the rate of observations does affect my method. The amount of time over which to

aggregate observations into a distribution must be large enough that a sufficient number of observations are collected. Determining what is a "sufficient number" of observations is an open issue of my methods. There certainly must be more observations in a distribution than bins in a QF. In general, however, the QF degenerates well (and much better than the PDF and CDF) to the case in which there are fewer observations than bins; however, such QFs typically have an anomalous shape. So, I will not compute QFs when there are fewer points than bins, and I will skip the anomaly test over any interval without a QF.

4.13 Chapter Summary

This chapter introduced a generic method to detect anomalies in server performance data. First, we discussed the advantages and disadvantages of various representations of distributions in Section 4.2, and we decided that the QF, or quantile function, was the best representation for our purposes. Then, we introduced PCA, or principal components analysis, and used it to construct a method for anomaly detection in Section 4.4. We saw that this method was generic enough to work with any set of server performance data, regardless of the shape and size. It is even generic enough to work with other types of data, such as request size and response size—subject to the limitations mentioned in Section 4.12. We also discussed how to "bootstrap" a definition of normality in Section 4.6.

Chapter 5

Case Studies

The previous chapter described methods for detecting server performance anomalies using only the abstract data made available by `adudump`. In this chapter, I want to show not only that these methods can detect anomalies, but also that these methods can detect *legitimate* anomalies, *i.e.* those that a network manager would actually care about. I also want to show how a human can interact with *adudump* data to plumb the depths of information and insight that it provides. The overall purpose of this chapter is to show the richness of diagnostic data that can be obtained merely by passive analysis of TCP/IP protocol headers in a continuous monitoring system.

This chapter starts by selecting a set of servers to explore in detail in Section 5.1. The seven servers selected are each explored in turn in Sections 5.2–5.8. Section 5.9 shows a shortcoming of the current anomaly detection approach with regard to strong outliers. Lastly, Section 5.10 discusses limitations of the results, and then I conclude.

5.1 Selection of servers

Section 4.7 discusses how the set of "heavy-hitter" server sockets was selected. I give the daily anomaly score timeseries for each of these server sockets in Appendix A. Of these servers, I visually selected the servers that had the most severe anomalies for a closer study. I excluded servers that either did not have an associated name (via a reverse-DNS lookup) or whose name included "dhcp". The latter group includes, for example, student machines in dorm rooms, and their performance is not generally of interest to a network manager. In the end, I selected

seven servers with either sustained anomaly scores above ten thousand or else multiple anomaly scores above 500,000. This list includes server 12 (Section 5.3), server 53 (Section 5.6), server 71 (Section 5.7), server 82 (Section 5.8), server 92 (Section 5.5), and server 104 (Section 5.2). In addition, Section 5.9 refers to server 109, and Section 5.4 refers to a portal server from Dataset 1 that has since been decommissioned.

Throughout this chapter, I will use the following shorthand classification system. Anomaly scores less than the `anom_thres` setting of 50 will be called "normal" and colored green. Anomaly scores between 50 and 200 will be called `notice` anomalies and colored yellow. Anomaly scores between 200 and 1000 will be called `warning` anomalies and colored orange. Anomaly scores between 1000 and 10000 will be called `error` anomalies and colored red. Lastly, anomaly scores greater than 10000 will be called `critical` anomalies and colored scarlet.

5.2 Case study: teleconferencing server

The first of the servers with severe anomalies is a web-based teleconferencing system. The anomaly timeseries for the server for the day time-scale is shown in Figure 5.1. Although most of the other servers we will investigate had ongoing performance issues lasting for weeks at a time, this server has more scattered anomalies. Starting the week of January 25^{th}, there was one `critical` performance issue each week lasting between one and three days. The rolling-day time-scale plot, shown in Figure 5.2, tells the same story.

Figure 5.3 plots distributions for the basis set and for some of the `critical` anomalies. The distributions had to be plotted on a logarithmic scale because they are very different from the basis set (normal distributions). Note that the most visually anomalous distribution, occurring on February 20, is also the date with the highest anomaly score.

Figure 5.1 suggests that the anomaly scores are correlated with the day of the week. The labeled days are Saturdays, and there is only one anomaly on those days. Also, most of the worst anomalies seem to occur in the middle of the week. Figure 5.4 makes this observation explicit. Every Tuesday that is not during an outage is at least a `notice` anomaly. Most of the other anomalies occur on Wednesday or Thursday, with the worst anomaly occurring on a Friday.

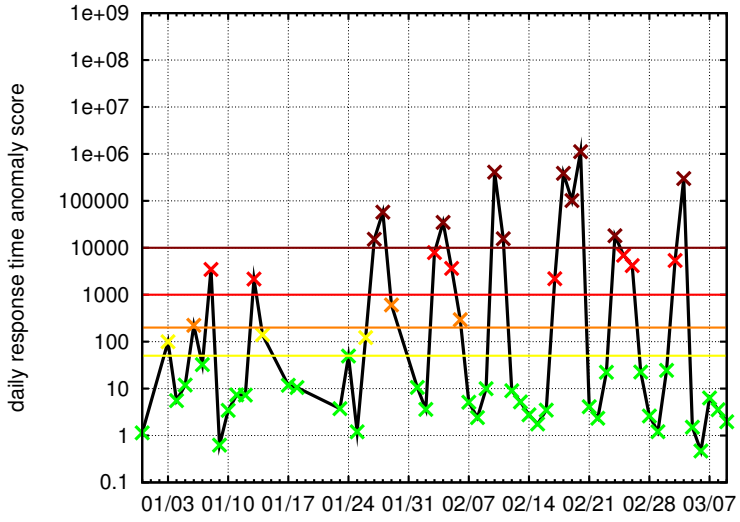


Figure 5.1: Anomaly score for teleconferencing server on a logarithmically-scaled y-axis. The labeled days on the x-axis are Saturdays.

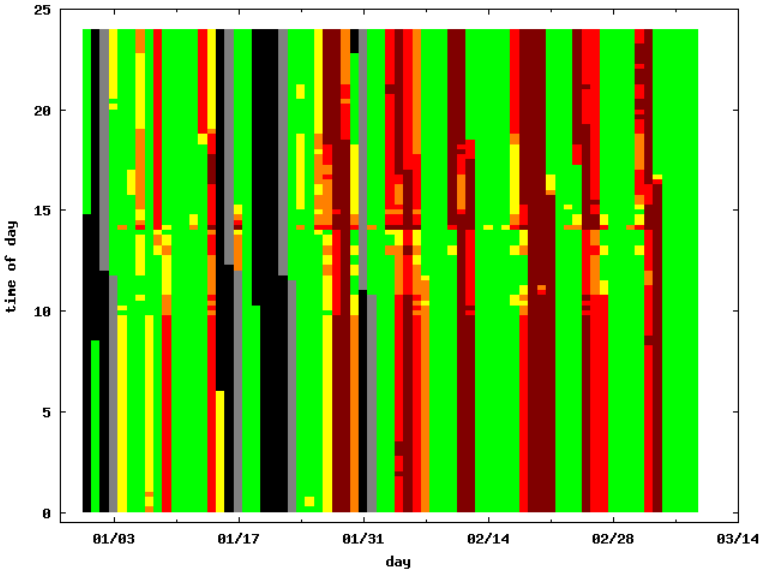


Figure 5.2: The time-scale analysis for the teleconferencing server with a rolling-day basis.

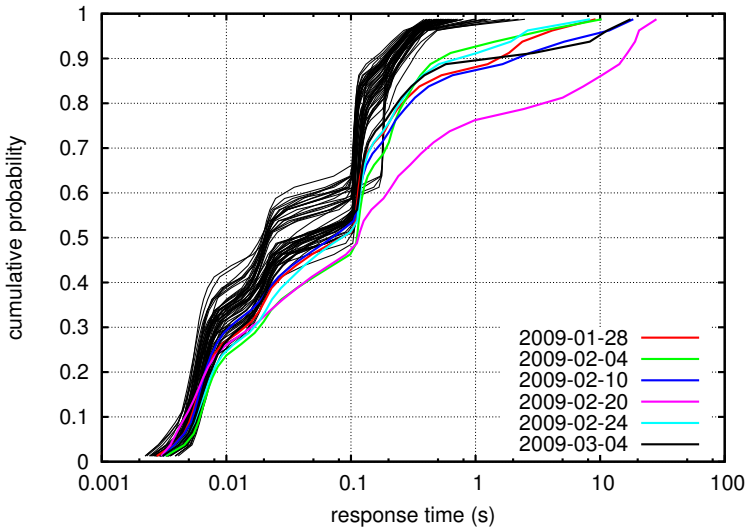


Figure 5.3: Selected distributions of `critical` anomalies for the teleconferencing server, plotted on a logarithmic x-axis.

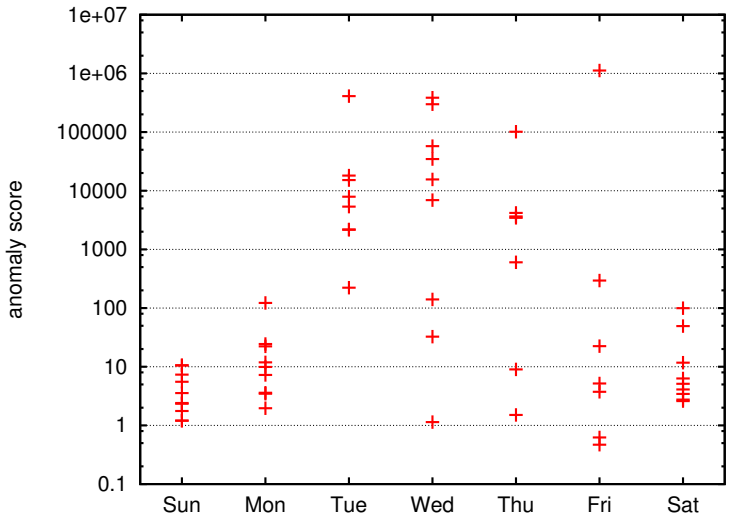
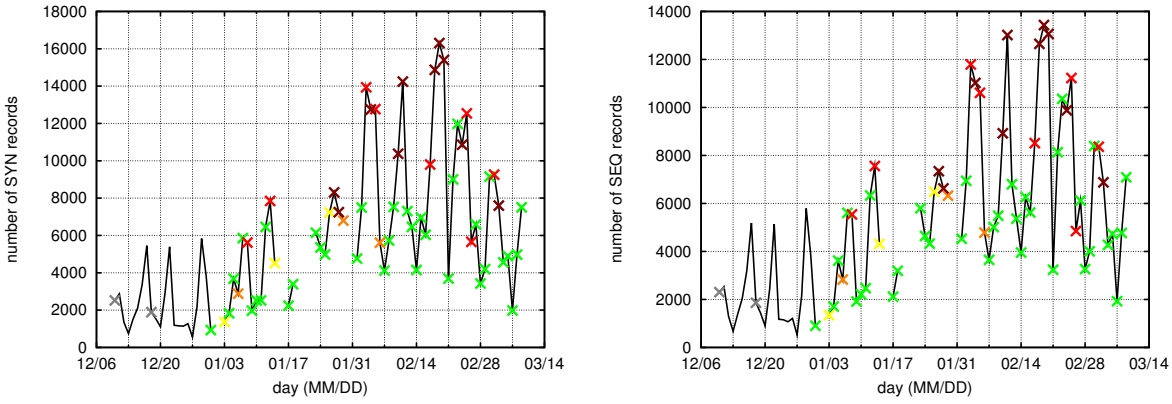


Figure 5.4: Anomaly score by weekday for the teleconferencing server, plotted on a logarithmic y-axis.



(a) SYN records

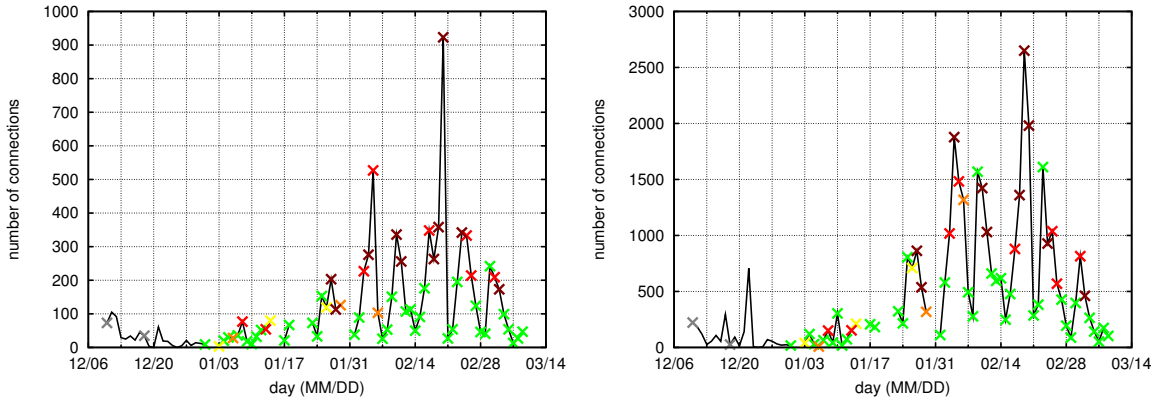
(b) SEQ records

Figure 5.5: The number of `SYN` (SEQ) records, or connection attempts (connection establishments), reported by `adudump` per day for the teleconferencing server. The color of the X indicates the anomaly classification of that day. Days without an X are from the training set, and the grey X's mark the days rejected by the bootstrapping process. Breaks in the line represent outages.

The data that `adudump` provides is also useful for providing insight into the characteristics of anomalous traffic. In general, I will call the process of searching for such insight the *diagnosis* process, although I note here that, because I choose a passive approach, I will never be able to diagnose the root cause of an issue with certainty. Instead, I merely attempt to list as many ways as possible that the traffic during a performance issue is different than the traffic during normal operation.

Perhaps the most obvious reason a server might have a performance anomaly is because it is overloaded; that is, it cannot keep up with the demand for its service. The data provided by `adudump` offers several ways to measure the demand. One such measure is the number of connection attempts. Figure 5.5(a) shows the number of `SYN` records reported by `adudump` for each day, with the plotted point (x) marking the number of `SYN` records color coded for the anomaly classification of that day. The anomalies occur when the demand is higher. Note that this is not simply the number of SYN segments sent to the server because retransmissions are not counted. Thus, keeping with the philosophy of abstracting away TCP effects, `adudump` reports the number of connection attempts regardless of whether packets were lost, retransmitted, *etc.*

Another measure of the demand on a server is the number of connections established involving the server. `adudump` reports a `SEQ` record whenever a connection is established. Figure 5.5(b)



(a) ignored connections

(b) rejected connections

Figure 5.6: The number of ignored (or dropped) connections and the number of rejected connections for the teleconferencing server, per day.

shows the number of `SEQ` records per day for the teleconferencing server. This measure of demand is also correlated with the anomaly score.

The difference between the number of `SEQ` and `SYN` records is the number of *failed* connection attempts. Connection attempts can fail either because they are ignored (or lost) or because they are rejected. Overloaded servers sometimes drop incoming SYN segments, and sometimes they reject the connection attempt by replying with a RST segment. The number of failed connections per day is shown in Figure 5.6. Like the measures of demand, the number of failed connections seems to be correlated with the anomaly score.

Yet another measure of the demand on a server—and one that `adudump` is particularly well-suited to measure—is the number of requests sent to the server. Connection-oriented metrics such as the number of attempts and establishments are useful data, but they are ubiquitous measurements available from many network measurement sources, such as Cisco's Netflow, Bro [Pax99], and others. While I do not claim that the number of requests is original to my research, it is less common, harder to measure, and at least as useful a measure. Figure 5.7 shows the number of requests per day. This measure of demand also seems to be correlated with the anomaly score.

To explore the correlation of demand *versus* anomaly score, I have plotted scatter plots for each of the measures of load introduced here. Figure 5.8 shows the anomaly score as a function of the demand, for our three measures of demand. Because the point cloud trends from the

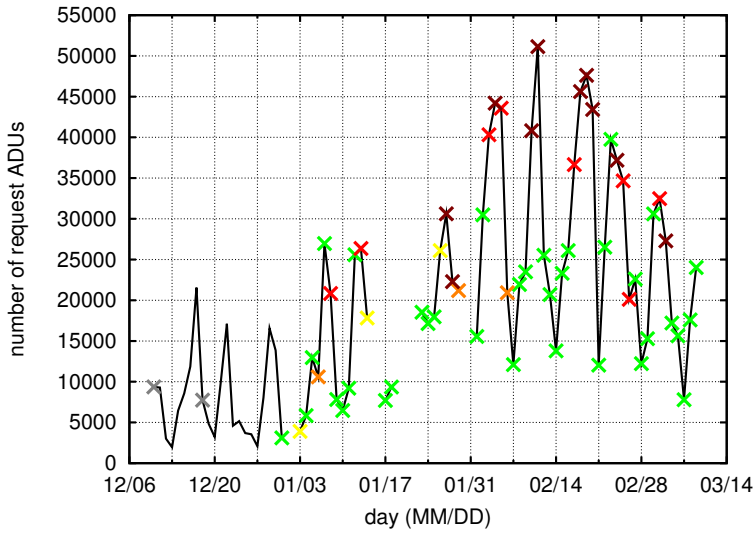
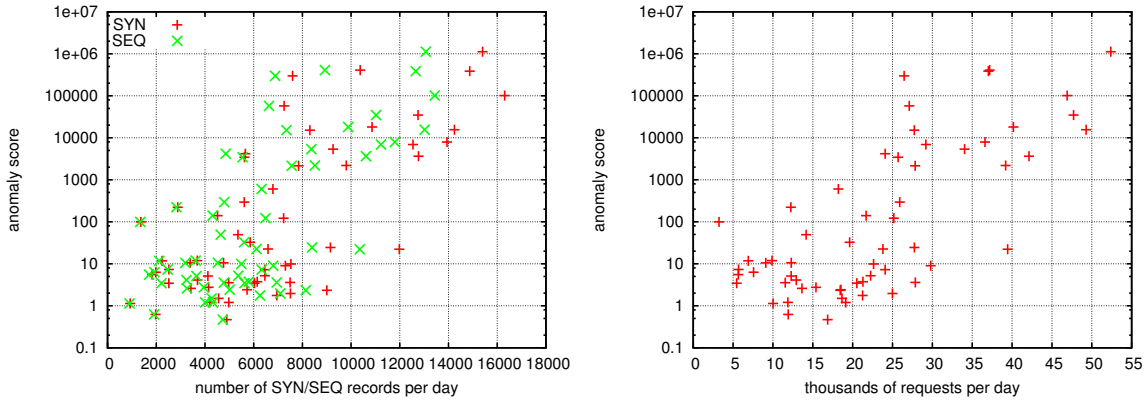


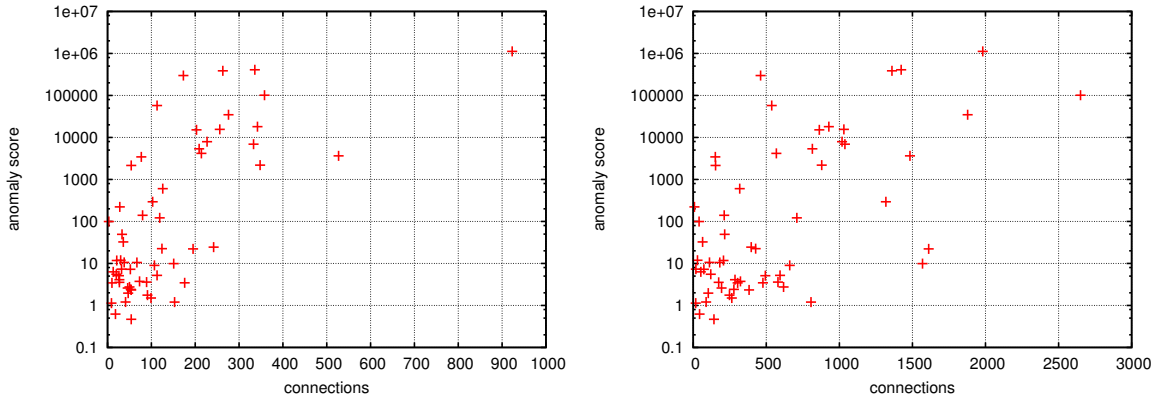
Figure 5.7: The number of request `ADU` records reported by `adudump` per day for the teleconferencing server.



(a) `SYN`/`SEQ` records

(b) requests

Figure 5.8: The anomaly score as a function of demand for the teleconferencing server, using three different demand measures.



(a) ignored connections

(b) rejected connections

Figure 5.9: The anomaly score for the teleconferencing server as a function of failed connection attempts.

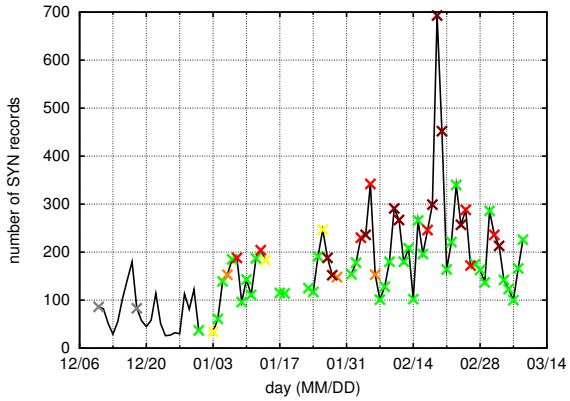


Figure 5.10: The number of unique client addresses per day for the teleconferencing server.

lower-left to the upper-right, I conclude that there is a positive correlation in all three cases. Note that because the y-axis is logarithmically scaled, the correlation is likely better explained by an exponential regression function rather than a linear regression function. I will not explore that issue here.

Figure 5.9 shows the anomaly score as a function of the connection failures. Again, these measures are positively correlated with the anomaly score: when the connection failures is above its mean, then the anomaly score tends to be above its mean, too. The same is true of the number of unique client addresses per day (Figure 5.10). If the number of unique clients per day was fairly constant, then I could conclude that a small number of clients are to blame for the high demand. The average client establishes between twenty and forty connections per day, and this number is relatively constant, regardless of demand and anomaly score.

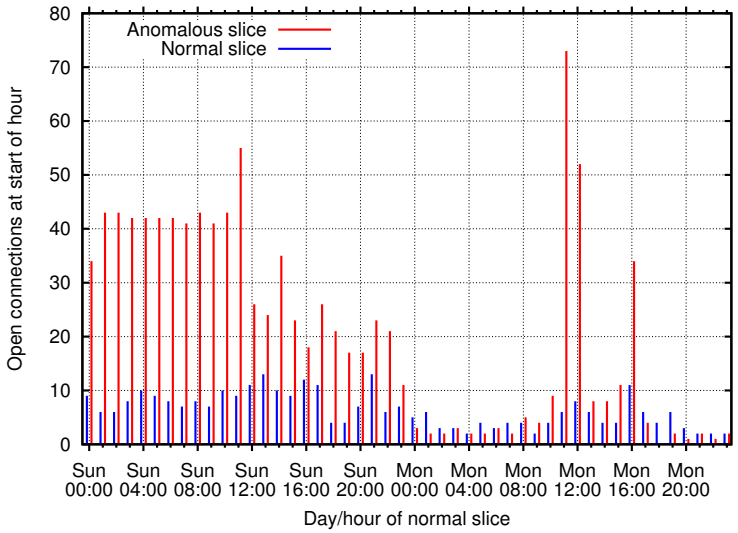


Figure 5.11: The number of open connections at the beginning of each hour in a normal and an anomalous slice of traffic. The anomalous slice was translated so that it could be plotted alongside the normal slice.

Yet another measure of load is the number of open connections at the server at one time. The more connections are open, the more likely the server is attempting to answer multiple requests simultaneously—a situation that can quickly lead to overloading. (Of course, connections can also be silent for extended periods of time.) Figure 5.11 shows the number of connections open at the server at the beginning of each hour, for both a two-day period of normal traffic and a two-day period of anomalous traffic. The anomalous traffic was taken from Thursday, February 19 to Friday, February 20. That Friday was the day with the highest anomaly score. The normal traffic was taken from Sunday, February 15 to Monday, February 16. The idea is to get representative and comparable periods of traffic—not to completely describe normal and anomalous traffic. The figure shows that there are more open connections during the anomalous period, again indicating a higher load on the server during anomalous periods. Ideally, it would be best to have periods come from the same weekdays, because traffic at UNC often follows a weekly pattern; however, for this server, there were no normal Thursday/Friday periods.

Although I cannot prove that high load *caused* the severe anomalies for the teleconferencing server, I have shown considerable evidence that suggests this conclusion. This evidence can serve the network manager in two ways. First, he is better prepared to tell the users who complain that he is already aware of the problem and investigating a solution. Second, he

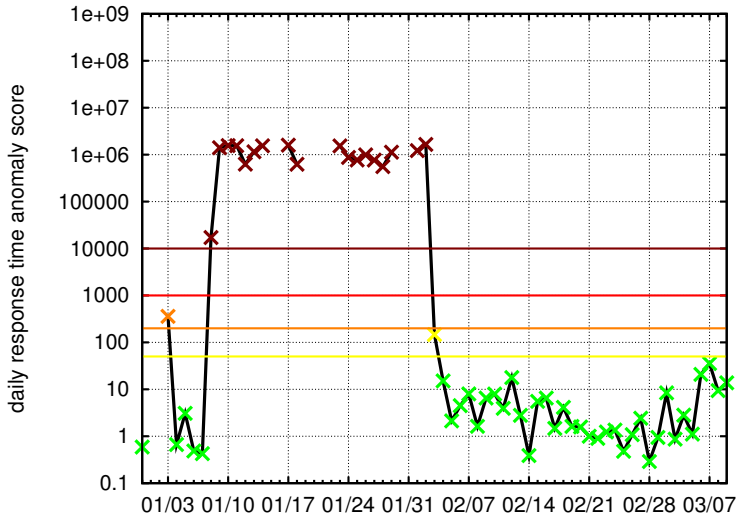


Figure 5.12: Anomaly score for POP3 server on a logarithmically-scaled y-axis. The labeled days on the x-axis are Saturdays.

has evidence to make a case to his boss that the service needs additional resources to function effectively. My system produced this evidence without any instrumentation of the server itself and without affecting the service in any way.

5.3 Case study: POP3 server

The second service I investigated operated on port 110, which is the port for POP3, the "post-office protocol" version 3. POP3 is used by email applications to fetch email from the email server. Of course, just because a service runs on port 110 does not necessarily mean that it is a POP3 server; however, this conclusion makes sense for other reasons, which we will see later.

Figure 5.12 shows the anomaly score per day for the server. There was an ongoing `critical` performance issue starting on January 8, 2009, and ending on February 3. Figure 5.13 shows that the issue was first detected at around 3:00 p.m. on January 8, and it reached `critical` status by midnight. Also, the issue began to wane around the same time on February 3, and the anomaly ended around 3:00 a.m. on February 4.

Figure 5.14 shows the distributions of response times comprising the anomaly, at both the beginning and end of the anomaly. The anomalous distributions are relatively typical for most of the distribution, differing only in last few bins, where they jump to about three seconds. In

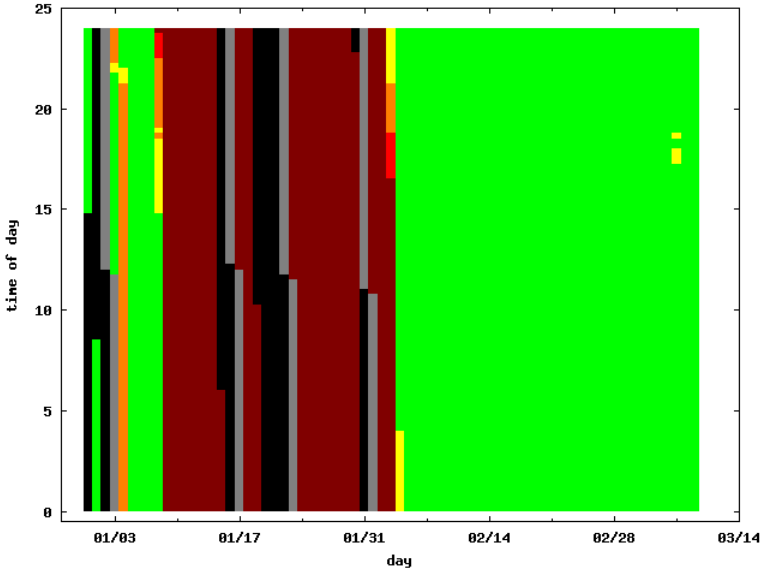


Figure 5.13: Anomaly score heat-map with a rolling-day time-scale for the POP3 server.

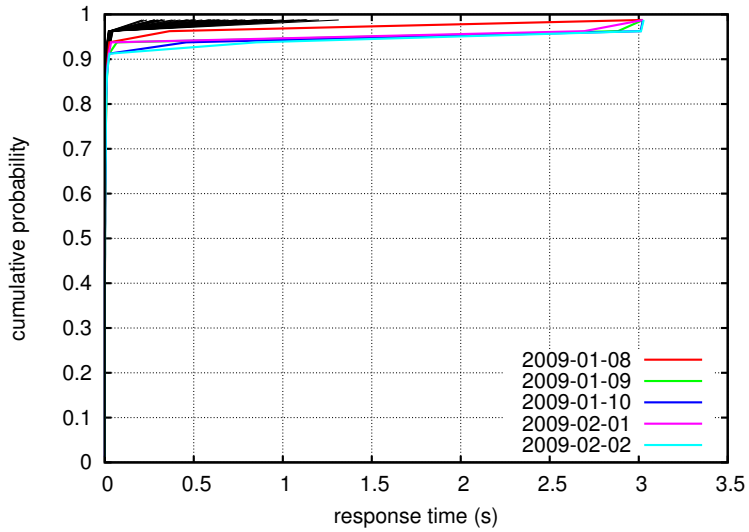
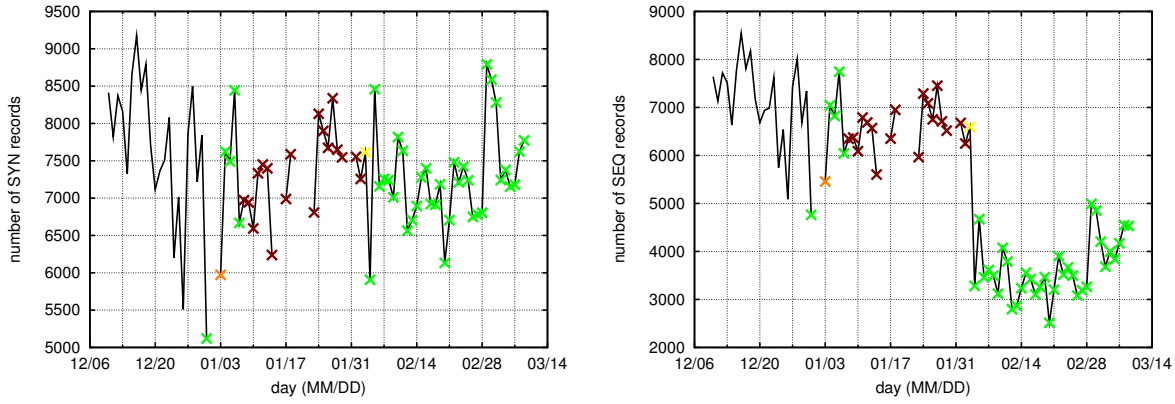


Figure 5.14: Distributions of response times leading to `critical` anomalies.



(a) SYN records

(b) SEQ records

Figure 5.15: The number of `SYN` (SEQ) records, or connection attempts (connection establishments), reported by `adudump` per day for the POP3 server. The color of the X indicates the anomaly classification of that day. Days without an X are from the training set. Breaks in the line represent outages.

general, such a consistent value for the last QF bin is indicative of a timeout mechanism on the server, which is an upper bound of the time to generate a response. I discuss the significance of three seconds later in this section.

The demand on the server in terms of the number of connection attempts per day is roughly the same during the anomaly as it is during typical operation, as shown in Figure 5.15(a). However, the number of connection establishments per day (Figure 5.15(b)) drops significantly as soon as the anomaly is over. The timing is suspicious, and an obvious question arises: is the drop in connection establishments somehow related to resolution of the performance issue? I will conjecture in the rest of this section that, indeed, the two are related.

Because the number of accepted connections drops after the anomaly, it would make sense if the number of requests dropped also. If that were not the case, then the nature of the traffic has changed in that the number of requests per connection has increased. But Figure 5.16 shows that, in fact, the number of requests drops commensurately with the number of accepted connections.

Because the number of connection attempts is roughly the same but the number of connection establishments decreases on February 4, the number of rejected connection attempts must commensurately increase. Indeed, Figure 5.17(a) makes this observation explicit. However, the number of rejected *clients* does not increase, which shows that there are a small number

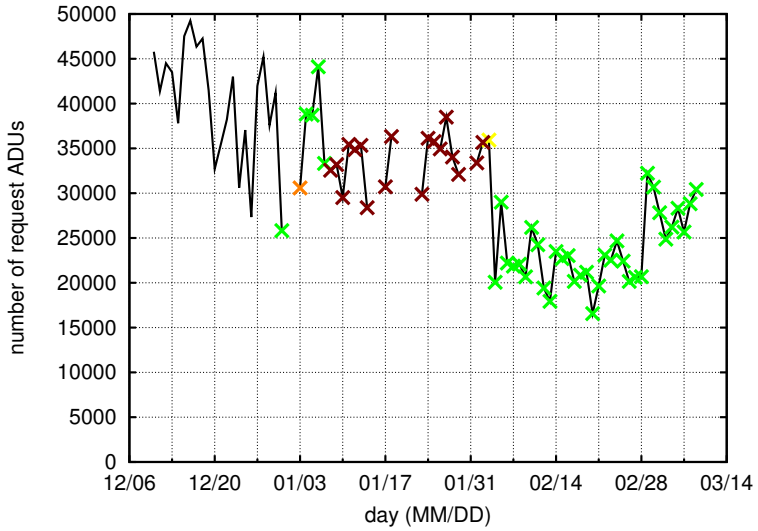
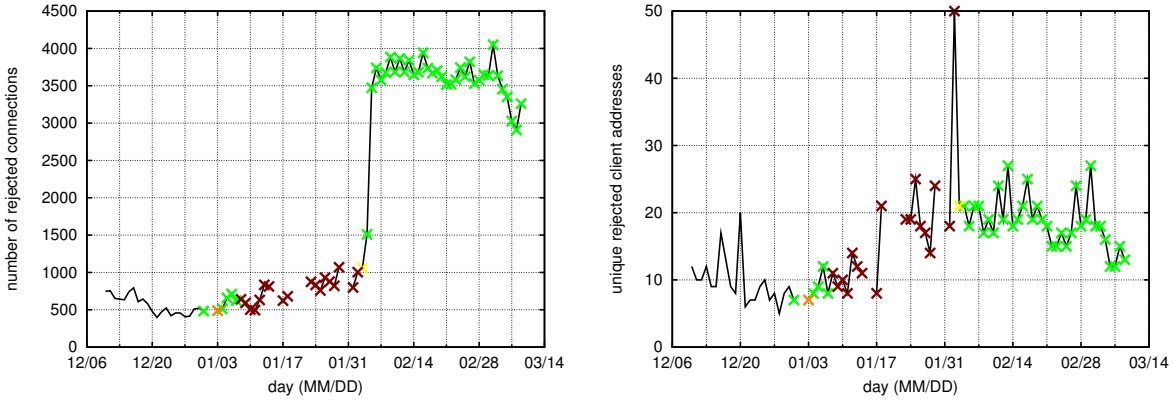


Figure 5.16: Number of request ADUs per day for the POP3 server.



(a) total

(b) unique rejected client addresses

Figure 5.17: Number of rejected connection attempts (a), and the number of unique client addresses rejected (b), per day, for the POP3 server.

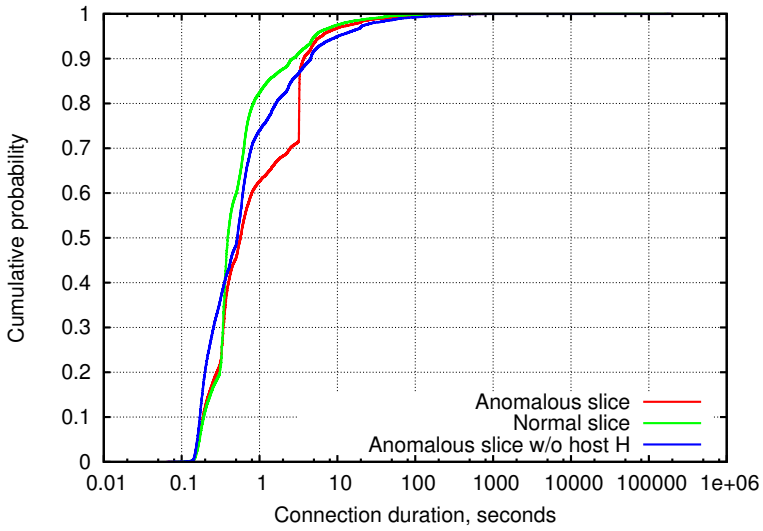


Figure 5.18: CDFs of connection durations in the normal period, anomalous period, and anomalous period with host H removed. Connection durations are taken by subtracting the timestamp of the `SYN` record from the timestamp of the `END` record reported by `adudump`.

of clients being rejected over and over again. In fact, a manual analysis of the `adudump` data shows that a single client is being rejected about 2,800 times per day, or about twice per minute. However, this client, which I label R (for Rejected), did not contact the POP3 server prior to February 4 at 4:35 p.m., so this does not explain why the number of connection establishments decreased as well. Curiously, another client (with a different IP address in a different network) successfully initiated a connection to the POP3 server about every thirty seconds, and this client, which I label H for heavy-hitter, ceased to contact the POP3 server on February 5 at 6:51 a.m. Both of these clients contacted the server far more frequently than any other client, and they strongly influence the performance and traffic characteristics, as we will see.

As in the previous section, I select a multi-day period of normal traffic and a multi-day period of anomalous traffic to compare. This time, I can line up the periods by week, so that the weekdays represented in each period are the same. The normal period includes all traffic from Sunday, January 4 to Wednesday, January 7, 2009, and the anomalous period is one week later, from January 11 to January 14. I deliberately choose a period before the anomaly occurs because I want to simulate the diagnosis process that the network manager might have gone through using real-time data displays.

Figure 5.18 shows the connection durations during each period. There is a mode at 3.2

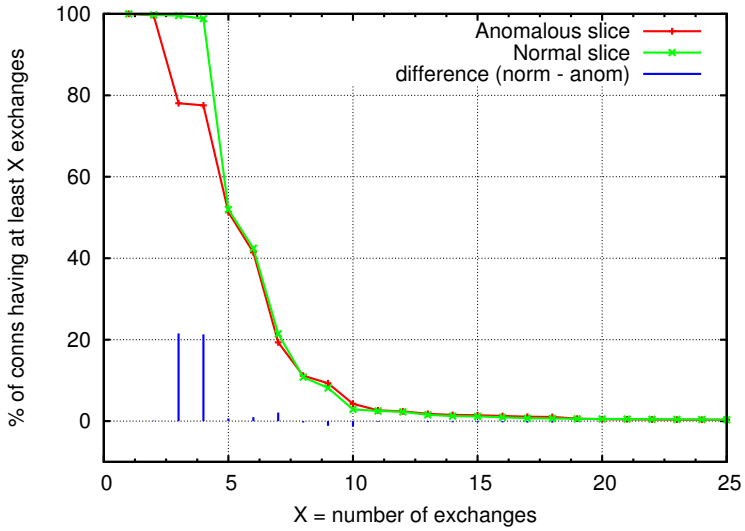


Figure 5.19: Percentage of connections having at least X exchanges for the POP3 server.

seconds in the anomalous period accounting for about 14% of the connections. Host H initiated 98% of the connections in this mode (which I selected as durations between 3.2 and 3.3 seconds), and the figure shows that removing host H also removes the mode.

So far, I have only considered contextual information about connections and network traffic provided by `adudump`. Now, I will begin to consider a richer sort of information that `adudump` provides: data about the structure of connections. Figure 5.19 shows the percentage of connections that have at least X exchanges. The anomalous period has about twenty percent fewer connections lasting beyond two exchanges, yet both periods have about the same number of connections lasting beyond four exchanges. However, all but about three percent of this difference is solely because of host H. Host H typically has the vast majority of its connections last for at least four exchanges (and most of them for exactly four), but, starting on January 7, about half of its connections only last for two exchanges.

There are more differences between the normal and anomalous periods in the connection structure. Specifically, I will now use *ordinal* analysis, which I introduced in Section 4.10. Figure 5.20 shows the ordinal response times. It is the second response time that is most to blame for the anomaly, because it is much longer in the anomalous period and still accounts for a significant proportion of response times (unlike *e.g.* the twelfth response time).

Figure 5.21 shows the ordinal request size and response size. The first request is slightly

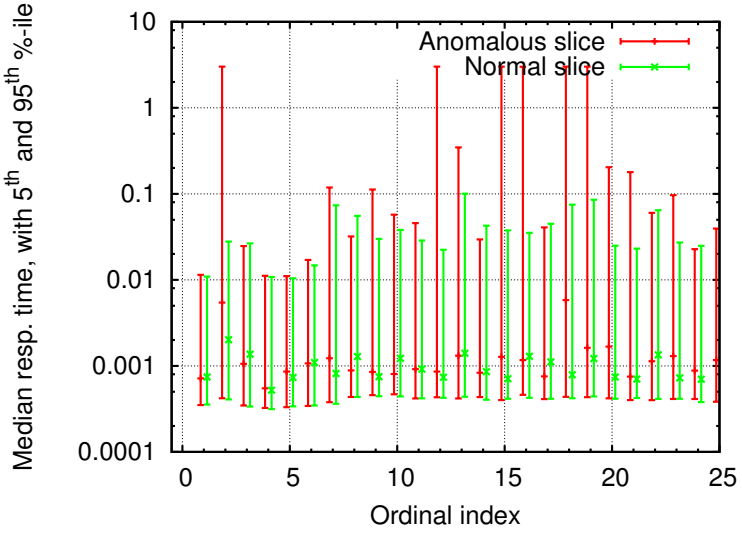
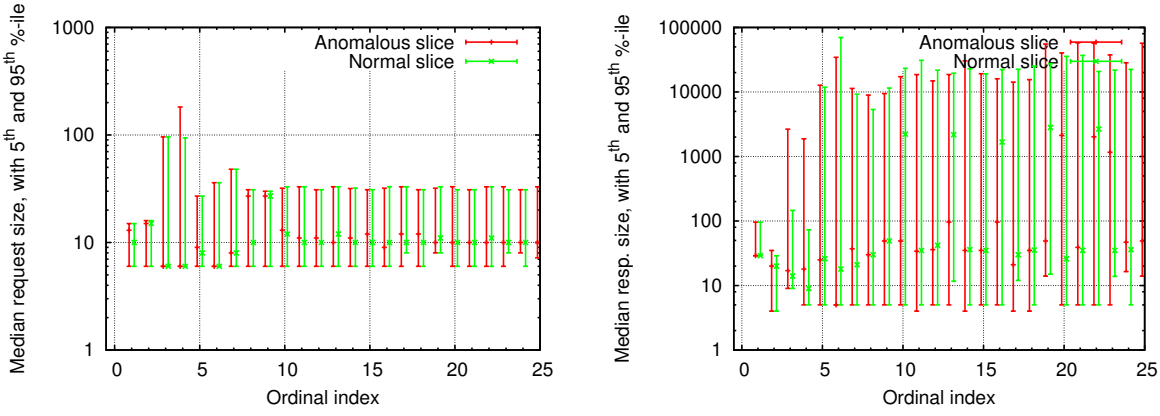


Figure 5.20: Median response time by ordinal for the normal and anomalous periods, with 5% and 95% error bars, for the POP3 server. For example, the point (and bars) at X= 2 represents the distribution of response times that occur for the second request/response exchange within their respective connection.



(a) request size

(b) response size

Figure 5.21: Median request and response size by ordinal for the normal and anomalous periods, with 5% and 95% error bars, for the POP3 server.

bigger in terms of the median, but generally, the requests are similar in both periods. Also, the requests are all small in an absolute sense. The only significant difference between the response sizes is that the third and fourth responses are bigger during the anomaly.

Recall that the server runs on port 110, which is the port for POP3. POP3 is described in RFC 1939, which gives the following example session:

```
S: <wait for connection on TCP port 110>
C: <open connection>
S:    +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>
C:    APOP mrose c4c9334bac560ecc979e58001b3e22fb
S:    +OK mrose's maildrop has 2 messages (320 octets)
C:    STAT
S:    +OK 2 320
C:    LIST
S:    +OK 2 messages (320 octets)
S:    1 120
S:    2 200
S:    .
C:    RETR 1
S:    +OK 120 octets
S:    <the POP3 server sends message 1>
S:    .
C:    DELE 1
S:    +OK message 1 deleted
C:    RETR 2
S:    +OK 200 octets
S:    <the POP3 server sends message 2>
S:    .
C:    DELE 2
S:    +OK message 2 deleted
```

```
C:    QUIT
S:    +OK dewey POP3 server signing off (maildrop empty)
C: <close connection>
S: <wait for next connection>
```

The `APOP` command does user authentication in one step, avoiding sending clear-text passwords and at the same time avoiding replay attacks. The key by which the user's password is encrypted is given in the "`+OK POP3 server ready`" command. However, not all servers support the APOP command, and a standard `USER` and `PASS` two-step authentication is provided as well. Here is an example of a connection using the two-step authentication:

```
S: <wait for connection on TCP port 110>
C: <open connection>
S:    +OK POP3 server ready
C:    USER mrose
S:    +OK User accepted
C:    PASS mrosepass
S:    +OK Pass accepted
C: <dialog continues...>
```

The POP3 server we have been considering in this section does not use the one-step `APOP` authentication because, as Figure 5.21(a) shows, the first request is too small. Therefore, I conclude that the server uses the two-step authentication. This explains why almost all connections (in the normal period, at least) have at least four exchanges: the two-step authentication, plus one to check for new mail, and one more to quit if there are no new messages. This also offers an explanation for why a connection might end after two exchanges: the authentication failed.

I will now offer a hypothesis for a correct diagnosis of the performance issue based on the data we have seen so far. This is the sort of hypothesis that the `adudump` data supports. If I were the administrator of this server (or if I had the time to talk to him or her), I would offer this hypothesis in the expectation that it could be verified as explaining the cause this anomaly. The

hypothesis is as follows. Many Linux distributions include pluggable authentication modules, or PAM, which handles authentication for a variety of services. The default configuration of PAM typically includes a three second waiting period upon a failed authentication attempt. This is designed to reduce the feasibility of an attacker flooding the server with password guesses and eventually guessing the right one. Recall that, as we saw in Figure 5.14, the anomaly manifests as response time distributions with a relatively large number of response times around three seconds. My hypothesis is that this anomaly is entirely due to the authentication failures of the heavy-hitter client H.

A major contribution of this case study is to demonstrate not only the diagnostic power of `adudump` data, but specifically the diagnostic power that information about the *connection structure* provides.

5.4 Case study: portal web server

I have introduced and carefully defined the anomaly detection methods in the previous chapter, and the past two sections have diagnosed anomalies that I detected with the methods. However, I have not yet shown that the anomalies flagged by the methods actually correspond to performance issues of interest to network managers. In this section, I will give an example of an anomaly that was a legitimate issue in the eyes of a network manager.

In addition to the `adudump` data measured at the border link of the UNC network (see Section 3.1.1, I have also collected an archive of all email sent to the UNC Information Technology Support (ITS) network services group (or ITS-NS). These emails include alerts of power outages, requests for VLAN changes, ITS change notices relevant to the network services group, and discussion of various network issues and problems. Occasionally, there is an email exchange about a performance issue with a server in the network.

One such issue that was posted to the email list was a performance problem with a portal server on Tuesday, April 8, 2008. The email noted that the portal, which primarily serves HTTP and HTTPs traffic, was "currently responding slowly to requests". The email also noted that, because many users use the portal as a launch point to other services, logging in to the portal and immediately navigating away from it, the regular login page was replaced with an

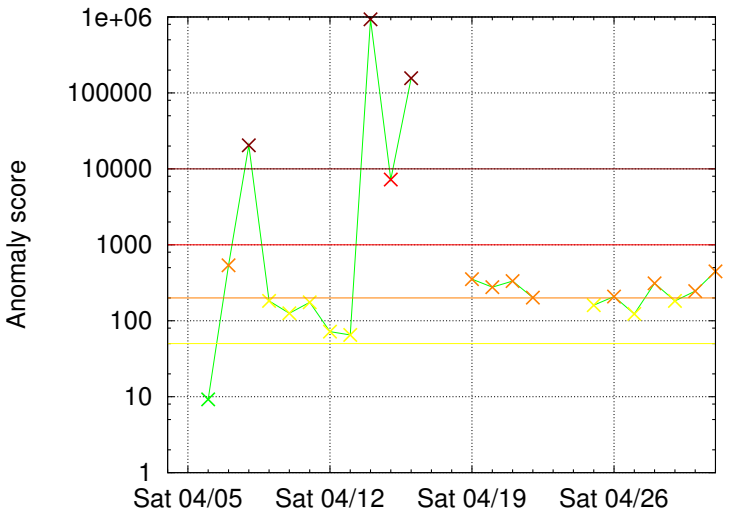


Figure 5.22: Anomaly score for portal server. Outages resulting in incomplete days are represented by breaks in the line. Dates are 2008.

interceptor page, so that such uses could occur without requiring a login and further taxing the system. The email was sent at 4:27 PM, and the performance issue was said to have started at 3:30 PM that same day. The interceptor page was put in place around 4:25 PM. Additional anomalies were also reported by ITS: there were emails to the ITS-NS email list on both April 14 and April 16 indicating performance issues with the portal.

These performance issues resulted in anomalies according to my methods. Figure 5.22 shows the anomaly score for the server before, during, and after the issue starting on April 8, plotted on a logarithmic scale. As before, the breaks in the line indicate outages in the `adudump` data collection process. The figure shows that the performance issue on April 8 was indeed visible to our methods, and it resulted in a `critical` anomaly.

Note that I did not *detect* this anomaly myself. At the time it occurred, I had not yet invented the anomaly detection methods I now use. Rather, I heard of this problem from the ITS-NS email list, and I have now gone back to the archived data to examine the issue in light of the current methods. The next section will discuss in detail a performance issue that I discovered independently. The contribution of this section is that the anomalies discovered by my methods do indeed correspond (at least in some cases) with genuine performance issues of interest to a network manager.

Figure 5.23 shows the distributions of response times before the issue (Sunday, April 6),

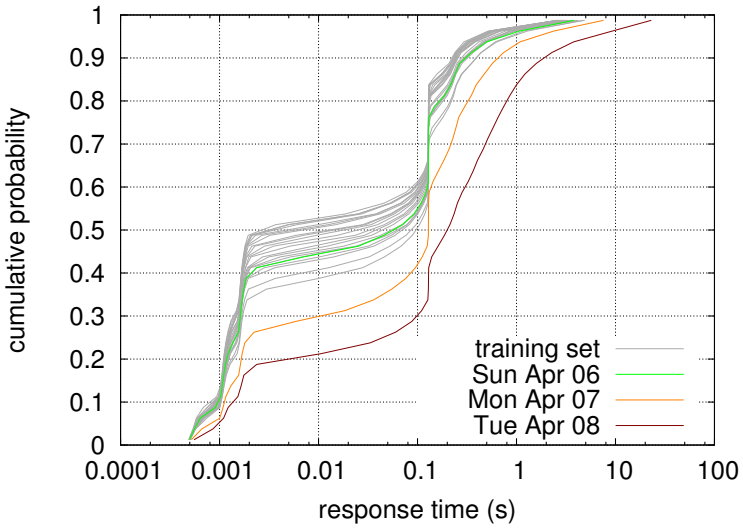


Figure 5.23: Training set (gray lines) and response time distributions for portal server.

leading up to the issue (Monday, April 7), and during the issue (Tuesday, April 8). The distributions are color-coded according to their anomaly score classification. The background distributions, or the distributions comprising the normal basis, are plotted in grey. The orange and scarlet distributions of April 7 and 8 are clearly heavier (*i.e.* larger, or shifted to the right) than the normal distributions—especially when considering that the X-axis is scaled logarithmically. One interesting result is that the data from `adudump` provided what could have been an early warning about the anomaly, at least sixteen hours in advance of when ITS reported noticing it.

As we saw in Figure 5.22, there was a subsequent anomaly from Monday, April 14, through Wednesday, April 16. Figure 5.24 shows the response time distributions during this anomaly. In fact, April 16 and especially April 14 are even heavier distributions than April 8.

A structural analysis similar to the one in the previous section shows that only about five percent of connections have more than three exchanges, and that this statistic is stable even during the anomaly (Figure 5.25(a)). It also shows that, although all ordinal response times are affected during the anomaly, it is the third and fourth that are the most affected (Figure 5.25(b)). I gave this information to the manager responsible for the portal, and he passed it on to the vendor of the portal package, but unfortunately, he never heard a response from them about this insight. Figure 5.26(a) shows that the first two request sizes are small

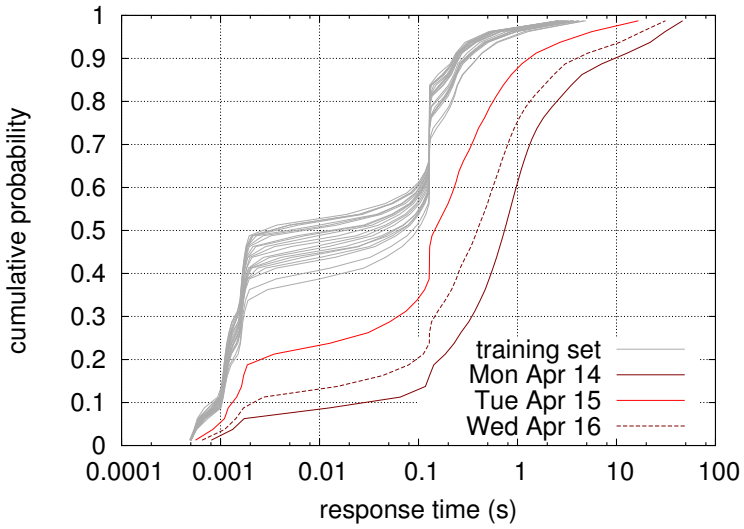
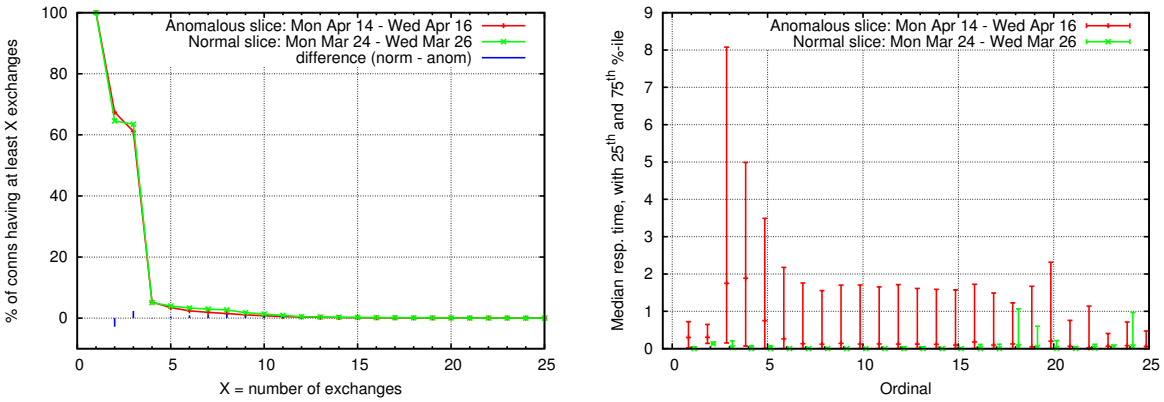


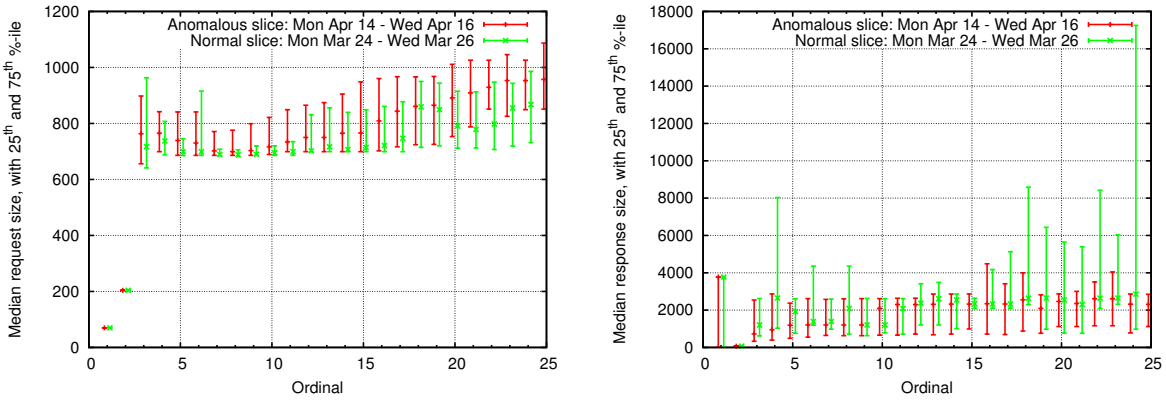
Figure 5.24: Training set (gray lines) and response time distributions for portal server.



(a) Percentage of connections having a given dialog length, for the normal and anomalous slices of the portal server

(b) Ordinal response times for the normal and anomalous slices of the portal server

Figure 5.25: Results from a structural analysis of the portal server



(a) request sizes

(b) response sizes

Figure 5.26: Results from a structural analysis of the portal server

and deterministic, but the third and beyond are larger; presumably, given that the server is an HTTP server, this means that the requests require more work, perhaps carrying authentication or session information. Figure 5.26(b) shows that the most important response sizes (*i.e.* the first three) are typical during the anomaly.

This server also had severe performance issues resulting in email messages to the ITS-NS list on several days in May, 2008, all of which were detected by my methods. The emails noted that the problems correlated with high load on the portal. Eventually, the server administrators discovered that the problem was due to a misconfiguration of the server that caused it to encrypt data passed from one process to another on the server itself. Since then, there have been no notable performance issues with the server, and in fact it has been strategically decommissioned since then.

In summary, the primary contribution from this case study is that the server performance problems that I detect are, at least in some cases, of interest to a network manager.

5.5 Case study: campus SMTP server

This section uses `adudump` data to investigate a performance issue with a campus SMTP server. SMTP, or the simple mail transfer protocol, is the protocol by which email is exchanged between source and destination email servers. The server with the performance issue is one of four that function as a virus and mail filter. External hosts sending mail to UNC addresses connect with

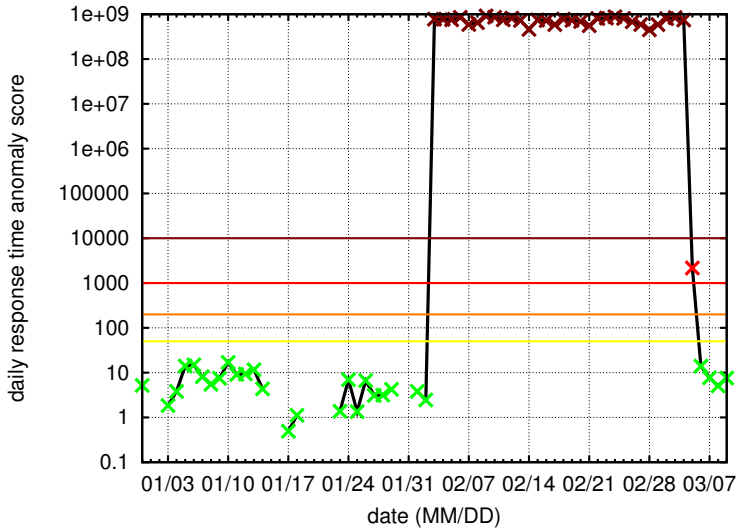


Figure 5.27: Anomaly score for campus SMTP server. Outages resulting in incomplete days are represented by breaks in the line. Dates are 2009.

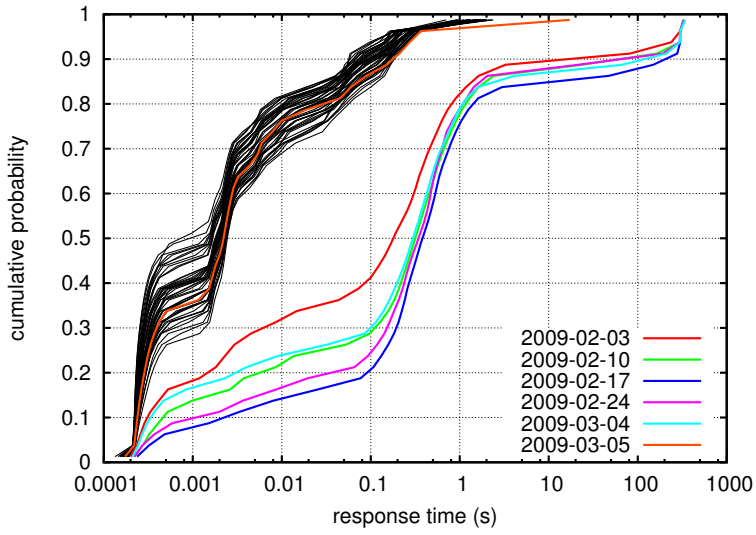


Figure 5.28: Distributions of response times both for normal operation (black lines) and selected anomalous distributions (colored lines), plotted on a logarithmic scale, for the campus SMTP server. Note that the anomaly on March 5 is only a `error` anomaly (*i.e.* with a score between 1,000 and 10,000).

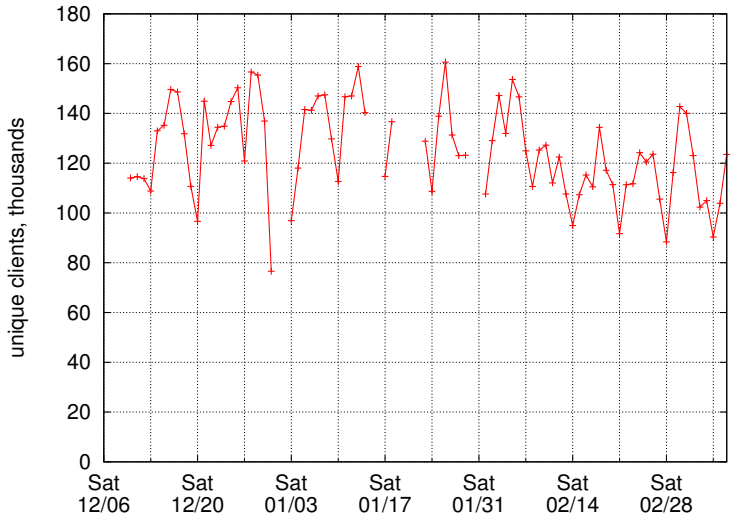
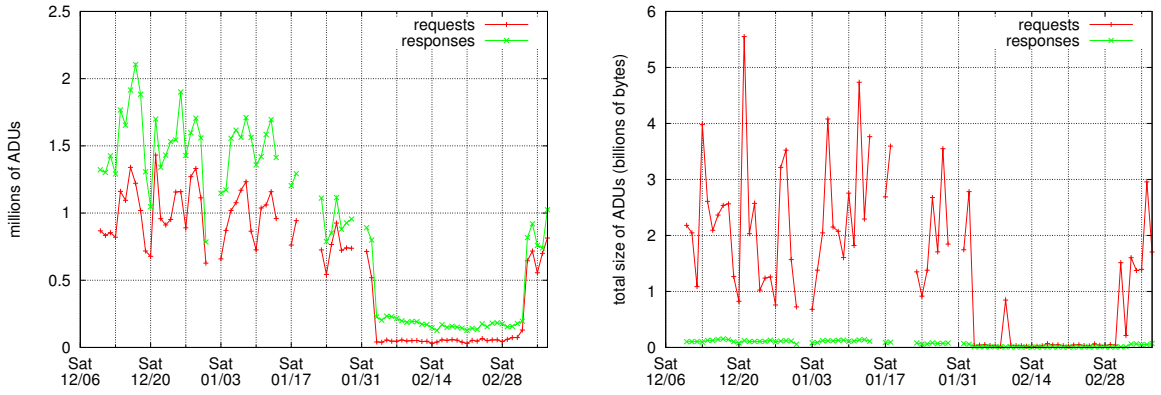


Figure 5.29: Number of unique IP addresses per day initiating a connection with the SMTP server.

these servers to deliver their mail. One of these servers accepts the message, determines whether it is spam and whether it contains a virus, and forwards legitimate mail to the internal UNC mail gateways. Likewise, UNC hosts sending email will connect with one of these servers, which will forward the email (to UNC addresses or not) if they are deemed legitimate. On February 3, one of the servers experienced a drastic performance issue, resulting in `critical` anomalies that endured without stop through February. The anomaly scores during this issue are the most extreme I have seen in the entire dataset, reaching nearly one billion. This particular SMTP server was the only one of the four affected; the other three saw no anomalies. The anomaly detection results are shown in Figure 5.27, and the distributions behind such results are shown in Figure 5.28.

Figure 5.29 shows the number of unique IP addresses per day initiating a connection to the SMTP server. Again, this is relative to our border-link vantage point, so connections initiated from within campus are not seen or accounted for here. Since there is a fairly steady number of clients, even during the anomaly, I conclude that the performance issue could not have been caused by a flash crowd of clients.

Figure 5.30(a) plots the number of request and response ADUs per day for the server, as reported by `adudump`. There is a sharp difference here during the issue. One initial curiosity is that the number of responses is greater than the number of requests. This is because I am



(a) count (b) total size

Figure 5.30: Count and total size of request and response ADUs per day for the campus SMTP server.

calling any ADU sent from the server (or connection acceptor) to the client (or connection initiator) a "response" ADU, even if it is not actually responding to any prior request ADU. This is a result of the design of the SMTP protocol, in which the server actually begins the application-level dialog by announcing itself. (The POP3 protocol, seen Section 5.3, shares this characteristic.)

Likewise, Figure 5.30(b) shows a similarly sharp shift in the total size of request and response bytes per day. Another implication of SMTP is evident here: the total size of the requests is greater than the total size of the responses. SMTP clients generally push data *to* the server, rather than other protocols such as HTTP in which the clients fetch data *from* the server.

To further compare the traffic comprising normal versus anomalous response time distributions, I aggregate the traffic for Sunday, January 4, through Saturday, January 10, as the "normal week", since I did not see any anomalies during this time and furthermore there were no outages. I also aggregate the traffic for Sunday, February 15, through Saturday, February 21, as the "anomalous week", since I saw continuous anomalies during this time.

Figure 5.31 shows the distributions of connection durations for both the normal and anomalous weeks. The connection duration is computed only on connections whose start and end were seen by `adudump` running on the monitor, and it is computed as the difference of the timestamps of the `END` and `SYN` records. In general, the anomalous week has much longer connections: half of them are more than 200 seconds long, whereas the normal week has only a small fraction

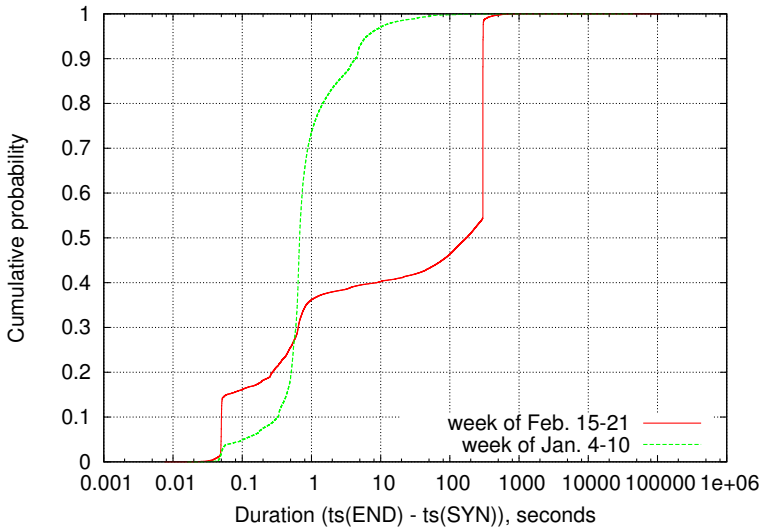


Figure 5.31: Connection duration for normal vs. anomalous weeks for campus SMTP server.

of connections longer than 20 seconds. There is also a small mode near 50 milliseconds for the anomalous week that is not present during the normal week. This is because of refused connections, in which the server immediately responds to connection requests with either a `RST` or a `FIN` segment (we cannot know which from the `adudump` data alone). I conclude this because, for these short-lived connections, a manual investigation of the data shows that `adudump` reported a `SYN` record immediately followed by a `END` record, with no intermediate records.

Figure 5.32 shows another significant difference in normal versus anomalous traffic: the number of open connections at a given time is much higher during the anomalous week. This increase is a logical consequence of the increase in connection duration.

But how many connection attempts were made per day? If there were far more connection attempts made during the anomaly (even though I already showed that the number of clients is roughly the same), then that might excuse or explain the poor performance of the server. However, Figure 5.33(a) shows that this is not the case: the number of connection attempts actually decreases during the anomaly.

The figure also shows that the difference between connection attempts and connection establishments increases during the anomaly. Figure 5.33(b) examines this difference in more detail, plotting the number of failed connections per day for the SMTP server. The number of failed connections will probably be of particular interest to many network managers, because

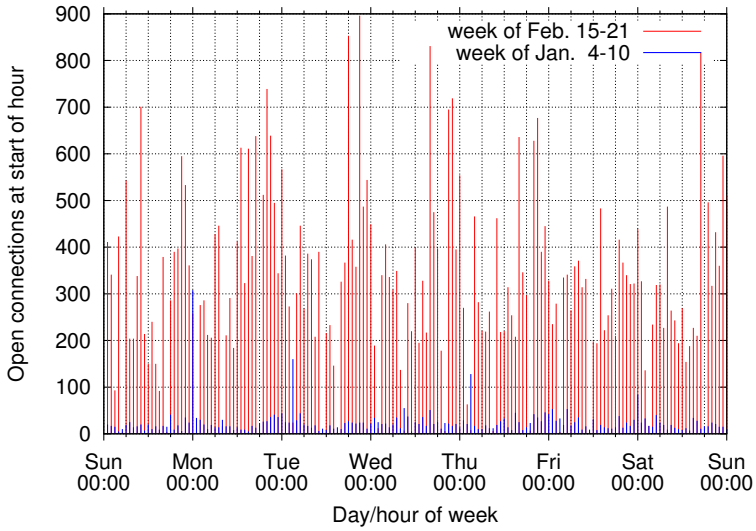


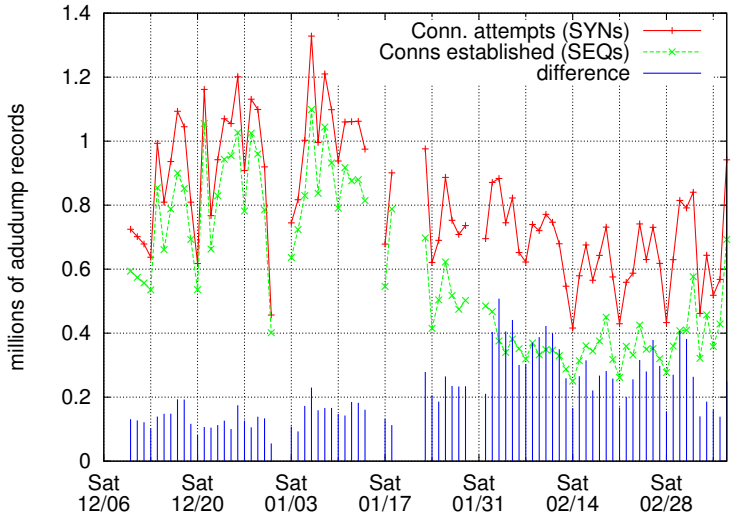
Figure 5.32: Open connections at the start of an hour for SMTP server, for normal week vs. anomalous week.

they represent an absolute failure to provide a service to a user.

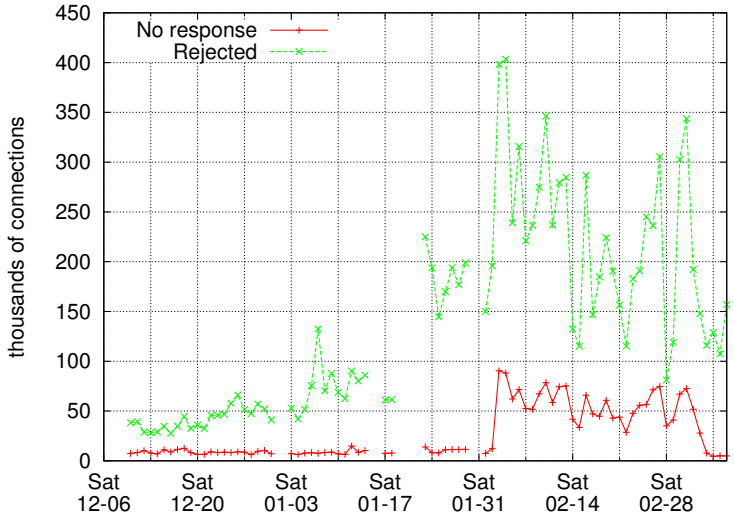
In the case of the SMTP server, there is such a causal relationship, although it is not as specific as the example above. Figure 5.34 is a scatter plot of the request size and the corresponding response time, during the normal week and the anomalous week, as selected above. The scatter plot shows that all "large" requests are followed by a "long" response time. Specifically, a manual investigation of the `adudump` data showed that, during the anomalous week, every request of at least 600 bytes always results in a response time of at least 30 seconds. Furthermore, although the normal week shows a generally positive correlation between request size and response time for larger requests, the anomalous week is dominated by response times of around 300 seconds. This seems to indicate a timeout value.

Figure 5.35 plots the response times conditioned on whether the request size was less than 600 bytes. If it was, the response time distribution is still quite different during the anomalous week (Figure 5.35(a)). However, if it was at least 600 bytes, then the distribution is extremely different for the anomalous week (Figure 5.35(b)).

Recall that SMTP is a protocol in which the main flow of data occurs from the client to the server. The application-level dialog involves a number of short exchanges, both to inform the client of the capabilities of the server and to inform the server of the "from" address and "to" addresses. The largest exchange is when the client transfers the email message to the server,



(a) Connection attempts (in millions)



(b) Connection failures (in thousands)

Figure 5.33: Connection attempts and failed connection attempts per day for the campus SMTP server

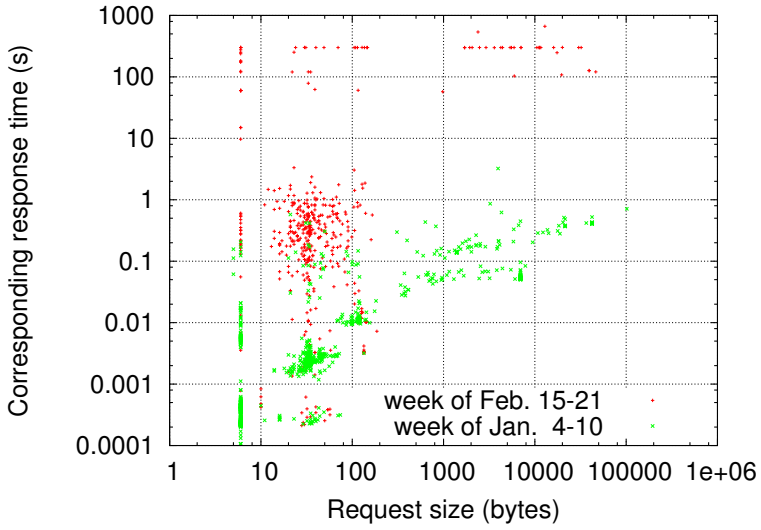
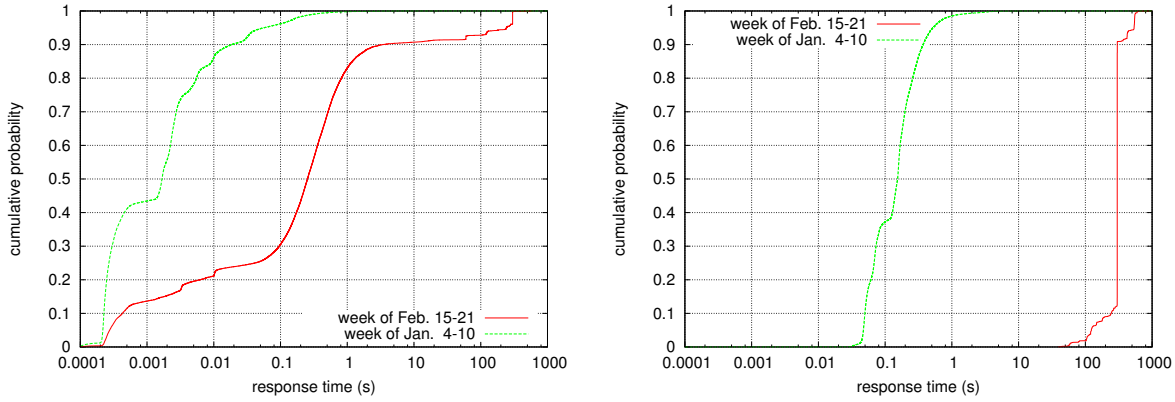


Figure 5.34: Scatter plot of request sizes vs. response times, for a normal week and an anomalous week. Note: the anomalous week is sampled at a rate of 1/100, and the normal week is sampled at a rate of 1/10,000.



(a) Response times when request less than 600 bytes (b) Response times when request was 600 bytes or more

Figure 5.35: Response time CDFs for campus SMTP server, split by whether the corresponding request was greater than or less than 600 bytes.

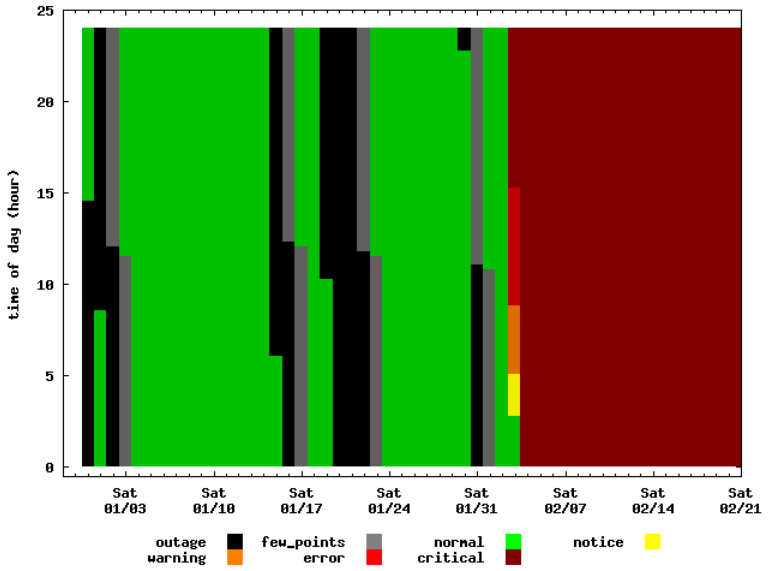


Figure 5.36: Time-scale analysis for SMTP server on rolling-day distributions.

and this exchange will often result in a request that is at least 600 bytes. Thus, it seems that the email message itself is what the server has a difficult time with during the anomaly. This is a valuable insight for the server administrator; however, note that the semantic meaning of the information is supplied not by my system, but by the human in the loop.

I spoke with one of the ITS employees responsible for this server. As it happens, the server, for an unknown reason, suddenly began having a very difficult time examining attachments from a popular office suite, causing the performance issue. The problem began when the mail filtering process on the server had to be restarted, which occurs occasionally when the process crashes. ITS has a "watchdog" process carefully measuring and reporting the operation of this server. Much of my findings were corroborated by the watchdog process, including the date the problem started, an indication of the severity of the problem, and the conclusion that the email messages themselves were triggering the problems. I also was able to provide additional information unavailable to the watchdog process, such as the number of failed connection attempts, the number of open connections, the number of successful connections, and the size and count of request and response ADUs. Furthermore, this sort of diagnosis can be done not only for any mail server, but for any server in the UNC network—even those without watchdog processes.

The performance issue with this server provides a good opportunity to investigate the *responsiveness* of the anomaly detection methods, because it had a sudden onset and resulted in

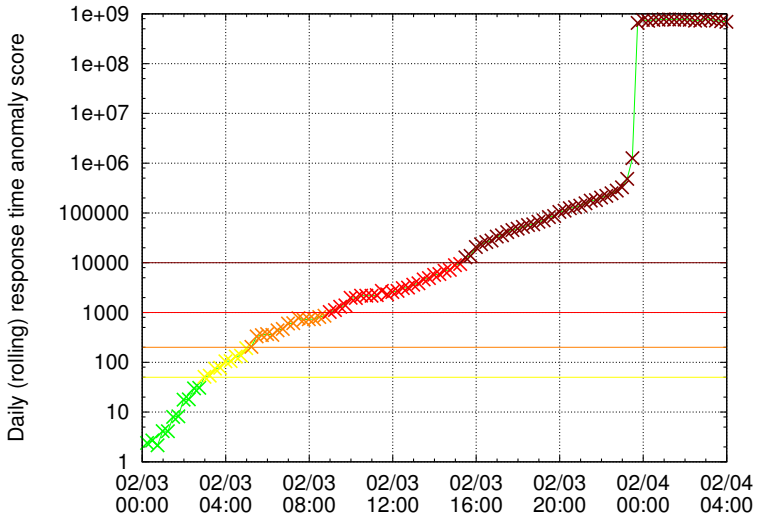


Figure 5.37: Anomaly score for SMTP server during the onset of the performance issue.

dramatically longer response times. Figure 5.36 shows the results of the anomaly detection done on a rolling-day time-scale. Figure 5.37 shows the anomaly score from each of the rolling-day anomaly tests at 15-minute intervals on February 3. The anomaly scores increase dramatically, and near the end of the day, as the last "normal" quarter-hour distributions drop out of the 24-hour window, the anomaly score jumps to nearly one billion. Note also that there were no anomalies of any severity prior to February 3, showing the stability and robustness of the rolling-day analysis.

Figure 5.38 shows that there was a drastic shift in the average response time occurring over the course of about an hour at midnight on February 3. This shift persisted for the duration of the performance issue. If the network manager had been alerted at the first occurrence of a `critical` anomaly, then he would have been alerted at 3:30 PM, about fifteen hours after the incident began. The first `error` anomaly occurs at 9 AM, about nine hours after the onset. Furthermore, there are two reasons why the network manager might have been alerted sooner: first, even a `notice` anomaly is unusual for this server; and second, the anomaly score timeseries is non-decreasing over the course of several hours. As it happens, although the watchdog process indicated an issue on February 3, it did not indicate the severity of the issue until February 4. (Unfortunately, I do not know the time of either of these alerts.)

Another aspect of this anomaly makes detection slower: the number of response times

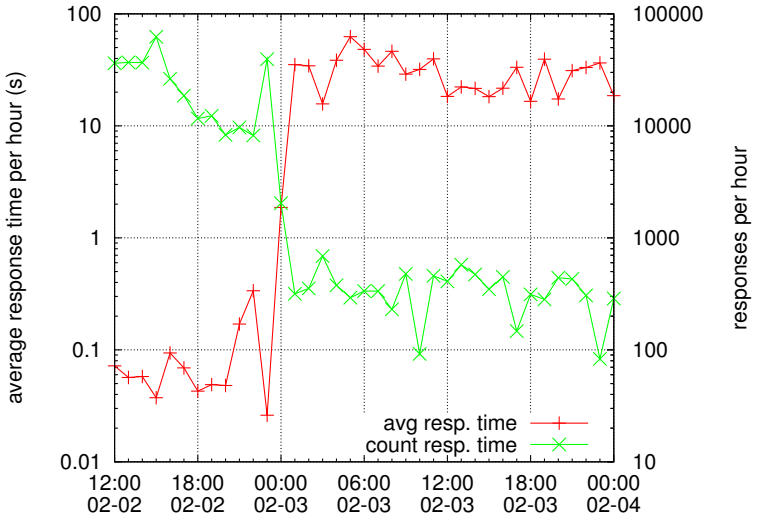


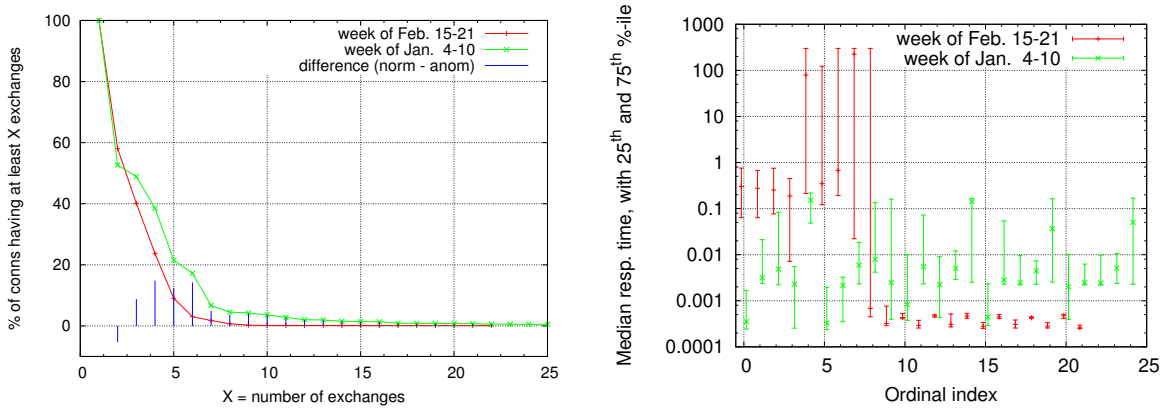
Figure 5.38: Average response time, and count of responses, per hour, for SMTP server during the onset of the anomalous behavior.

decreased drastically, as shown in Figure 5.38. As a result, the weight of the pre-anomalous quarter-hour distributions (*i.e.* those from February 2) dominates the combined rolling-day distribution, making the anomalous distributions less noticeable in the combined distribution. One potential solution to this problem is to give equal weight to each quarter-hour distribution comprising a combined 24-hour distribution. This is an area of future work.

I have done an ordinal analysis for the problematic SMTP server. Figure 5.39(a) shows the number of exchanges per connection for the SMTP server. About half of the connections have only one exchange. Relatively few connections have seven or more exchanges. There are generally more exchanges in connections from the normal period than connections from the anomalous period.

Figure 5.39(b) shows the median and first and third quartile of response times, by ordinal, for the normal and anomalous periods. The differences are so great that I had to plot them on a logarithmic scale. Although the first through fourth response times are clearly different during the anomalous period, the differences become extreme starting in the fifth response time, where the third quartile is over 100 seconds.

This phenomenon makes sense in light of the SMTP protocol. In a typical connection, the first exchange is a welcome message, often containing a static list of server capabilities and options. The second exchange identifies the sending email address. At least one additional



(a) Percentage of non-empty connections with at least X exchanges for SMTP server

(b) Q1/Q2/Q3 response time for SMTP server by ordinal, normal vs. anomalous

Figure 5.39: Results from ordinal analysis of the campus SMTP server.

exchange identifies the recipient email addresses. One more declares that the next exchange will contain the email message data. Thus, the email message is in the fifth exchange at the earliest, and later if there was more than one recipient address. This defined behavior explains the information from Figure 5.39(b), because the SMTP server (which, remember, is a virus and spam filter) must work harder when analyzing the actual email message.

Recall from Section 4.11 that, if a server's traffic has shifted, perhaps permanently, then the network manager can simply include some of the "anomalous" (now normal) distributions into the training set to reset the basis and redefine what is considered normal for the server. Figure 5.40 shows this redefined training set, which includes some of the distributions after the traffic shift on February 3^{rd}. This training set is then the input to the basis definition method (skipping the bootstrapping cross-validation step), which gives us a definition of normal in terms of the PCA-defined normal basis. This new normal basis is then used to test the new distributions. Figure 5.41 shows the resultant anomaly scores, only two of which are `notice`-level anomalies. The new scores are much lower than the `critical` scores of nearly one billion when the original basis was used. The actual anomalous distributions are shown in Figure 5.42, which shows that many distributions having the heavier shape were indeed considered normal. This example demonstrates the power of resetting the basis: even the most extreme anomalies in the dataset were reduced significantly by including a handful of anomalous days in a manually selected basis.

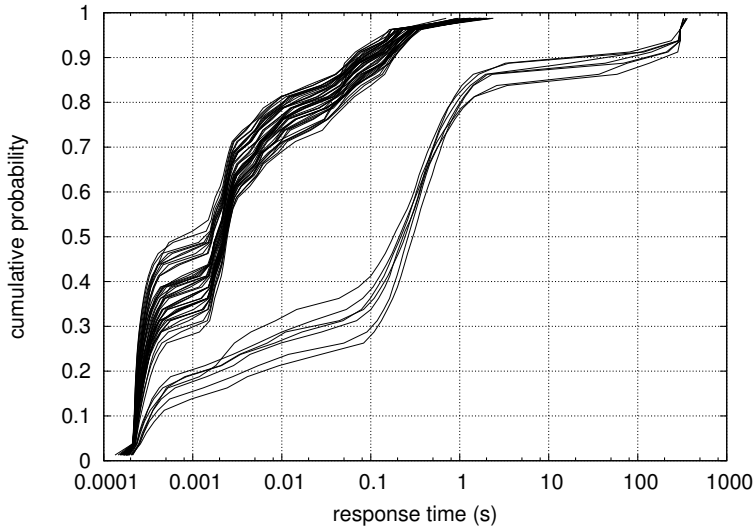


Figure 5.40: Manually selected training set: all days through Feb. 9 are included. The heavier distributions occur after the traffic shift on February 3^{rd}.

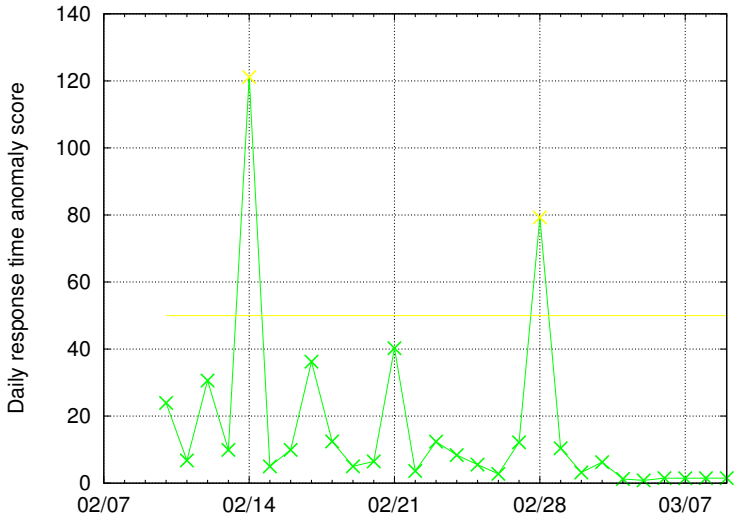


Figure 5.41: New anomaly timeseries, after resetting the basis.

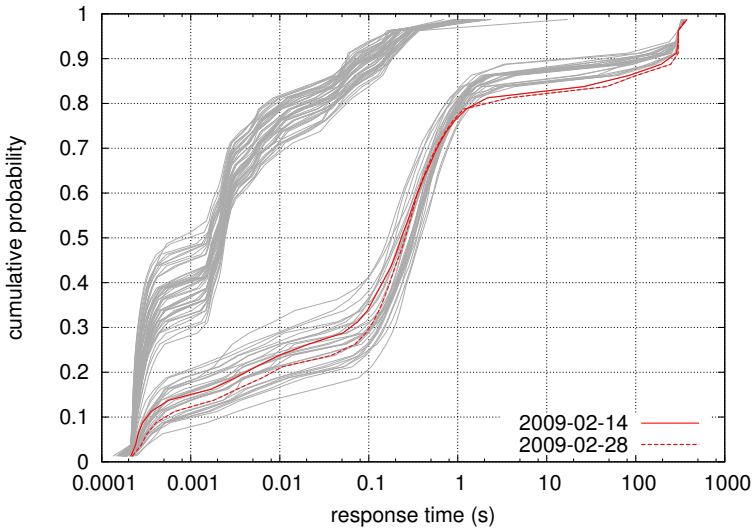


Figure 5.42: Normal and anomalous distributions after resetting the basis. Only two distributions were flagged as anomalous, and (despite the color) they were only `notice` anomalies. Thus, many of the previously anomalous distributions are now considered normal.

This section investigated a performance issue with a campus SMTP server. The anomaly scores for this server were the highest of any in the dataset. The problem was traced to the part of the SMTP dialog in which the email message was sent from the client to the server, and the server was unable to complete the virus and spam checking on the message. This diagnosis was corroborated by ITS staff. I also demonstrated the effect of resetting the basis, showing that even the most extreme anomalies can be accommodated in a simple operation if they represent new operating modes.

5.6 Case study: departmental web server

The fifth server performance issue that I will explore in detail involves a web (or HTTP) server assigned to a department within UNC. The anomaly detection results are shown in Figure 5.43. There are no anomalies prior to February 6. After February 6, `error` anomalies occur pretty consistently. Then, starting March 5, `critical` anomalies occur for a period of four days. In this section, I will first investigate the difference between normal operation and the `error` anomalies, and then I will investigate the difference between `error` anomalies and `critical` anomalies. Representative distributions of each period are shown in Figure 5.44. I will use a normal time period of Friday, January 23, to Sunday, January 25; an "error" time period of

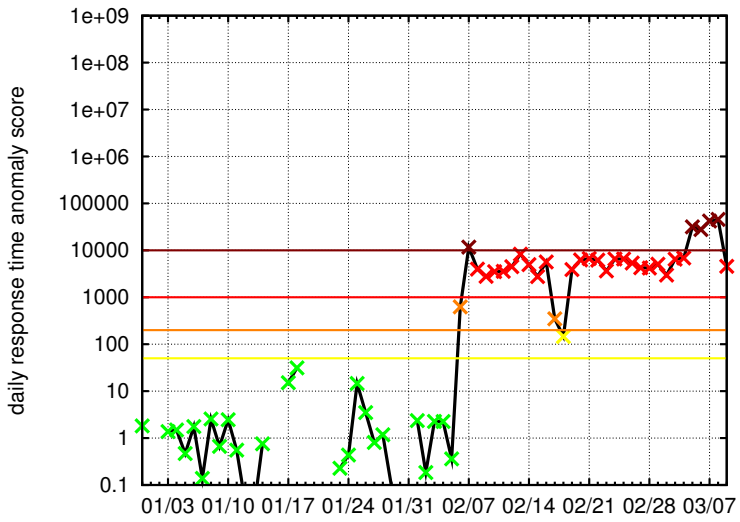


Figure 5.43: Anomaly score for departmental web server. Outages resulting in incomplete days are represented by breaks in the line. Dates are 2009.

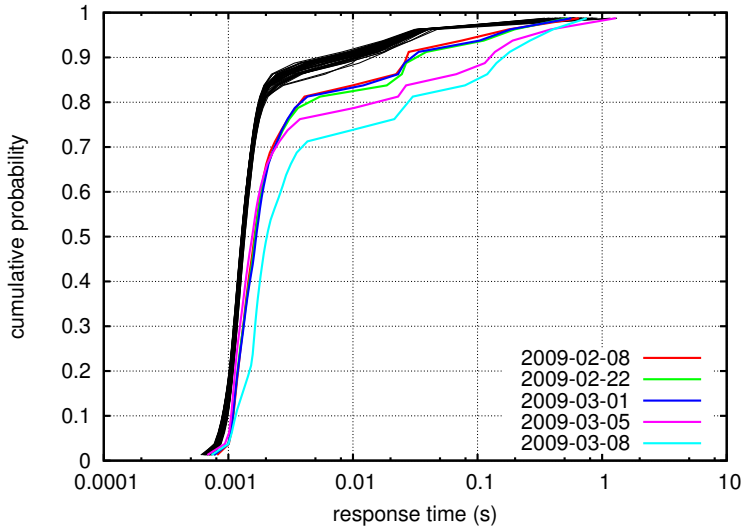


Figure 5.44: Distributions of response times for normal operation (black lines) and anomalous operation (colored lines). The first three colored lines are from the `error` period, and the last two lines are from the `critical` period.

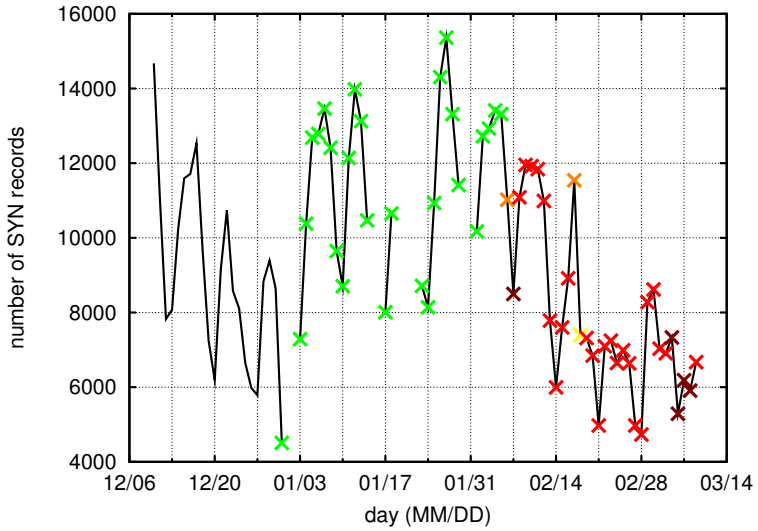
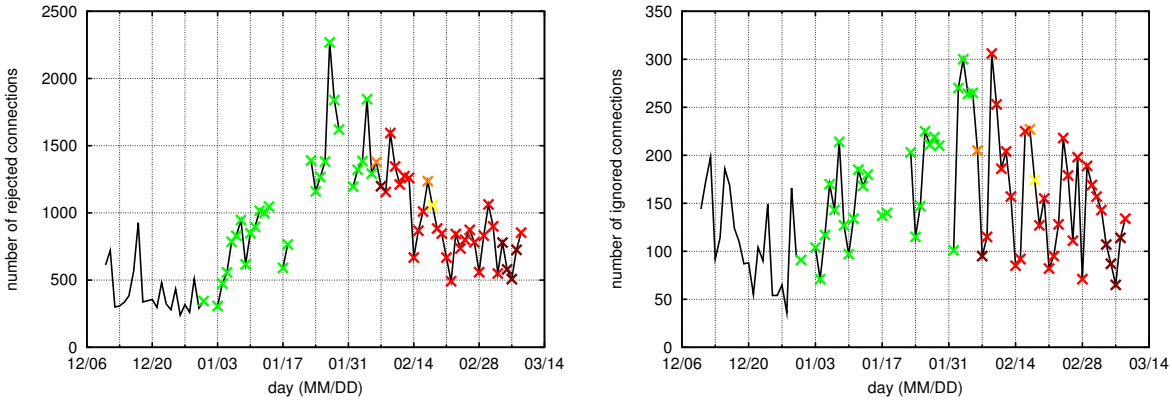


Figure 5.45: The number of connection attempts per day for the departmental web server.

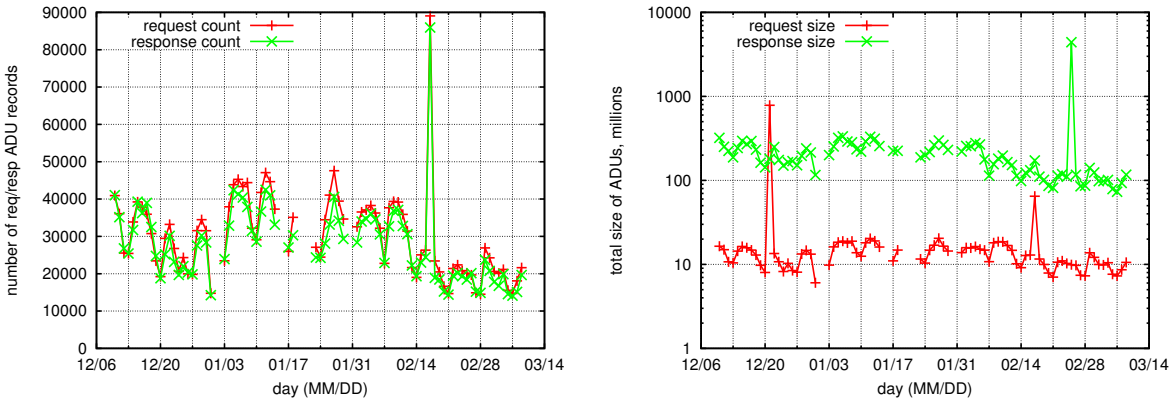


(a) rejects (b) ignores

Figure 5.46: The number of connection attempts that were rejected or ignored per day for the departmental web server.

Friday, February 27, to Sunday, March 1; and a "critical" time period of Friday, March 6, to Sunday, March 8.

In comparing the normal period to the "error" period, many metrics do not have a consistent, significant difference. This list of metrics includes the number of connection attempts (Figure 5.45), the number of connection failures (both rejected and ignored, Figure 5.46), number of requests and responses (Figure 5.47(a)), total size of requests and responses (Figure 5.47(b)), and the number of open connections at any given time (Figure 5.48). The response times were sometimes longer on average (Figure 5.49), but not always, showing that a threshold



(a) count (b) total size

Figure 5.47: Count and total size of request/response ADUs per day for the departmental web server.

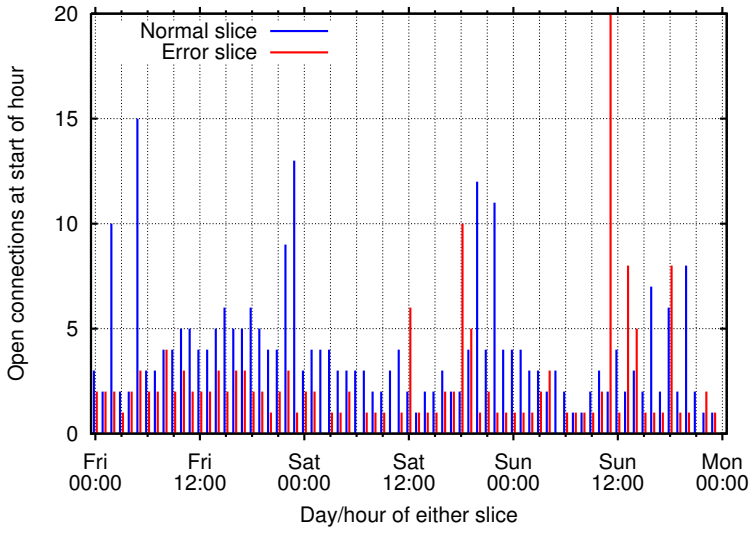


Figure 5.48: The number of open connections at the start of an hour for both the normal and error period of the departmental web server.

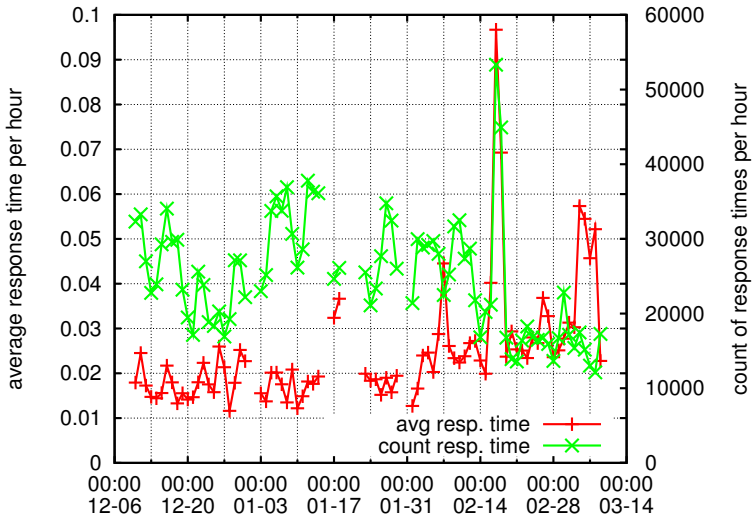
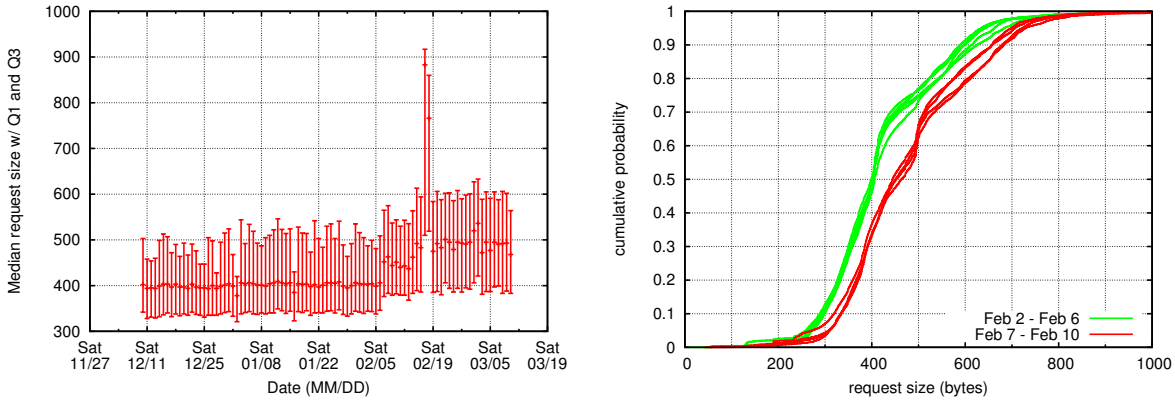


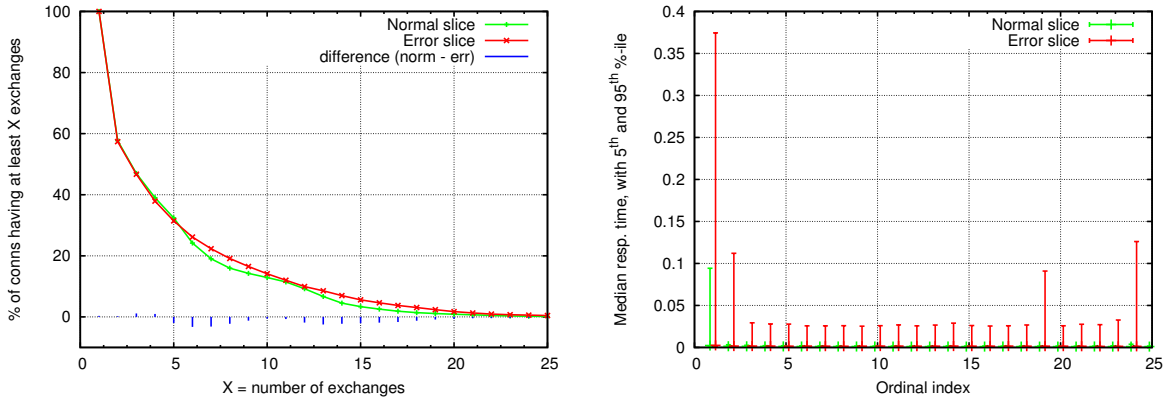
Figure 5.49: The number of response times and average response time per day for the departmental web server.



(a) Summaries of daily distributions (median with Q1/Q3 error bars) of request sizes.

(b) Daily distributions of request sizes, prior to February 7 and after February 7.

Figure 5.50: The request size for the departmental web server shifts on February 7, 2009.



(a) The number of connections having X exchanges for the normal and "error" periods of the departmental web server.

(b) The median response time, by ordinal, for the normal and "error" periods of the departmental web server.

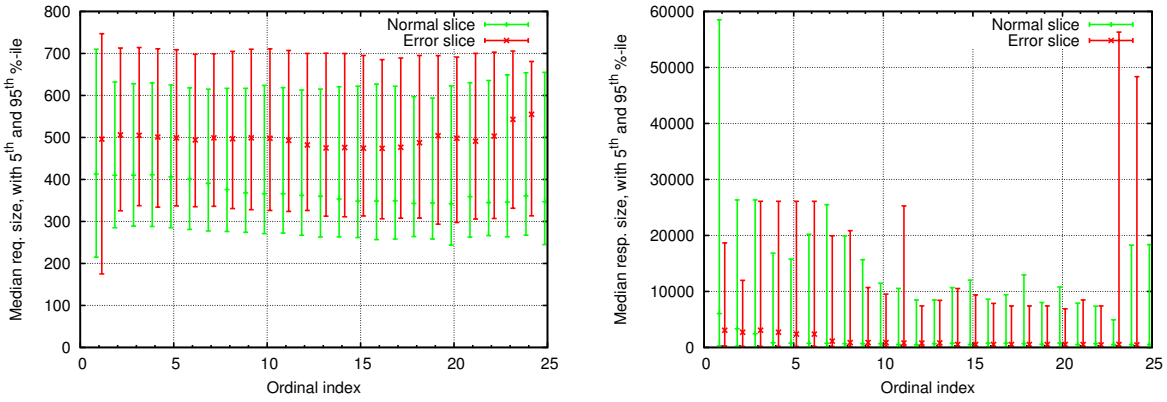
Figure 5.51: Results from an ordinal analysis of the normal period versus the "error" period for the departmental web server.

on the average response time is too simple an approach to account for the complex variation that a server's response times might have. In any event, the primary difference I could find was in the distributions of request sizes. On February 7, the median request size jumped from a stable value of about 400 bytes to between 450 and 500 bytes. Figure 5.50 shows how this distribution changed over time.

The other differences between the normal and "error" periods are revealed by an ordinal analysis. Figure 5.51(a) shows the distribution of the number of exchanges per connection. The error period's distribution is not significantly different than the normal distribution. Figure 5.51(b) shows the ordinal response times. The response times during the error period are consistently longer in the ninety-fifth percentile for all ordinals. Furthermore, the first response time is the longest, on average, in both cases.

Figure 5.52 shows the summarized distributions of request and response sizes, by ordinal. We already saw in Figure 5.50 that the request sizes are longer for the "error" period; Figure 5.52(a) shows that this is true for all ordinals. It is difficult to tell a coherent story for the ordinal response sizes, but they are clearly different between the normal and error periods.

Because the only differences I can find between the normal traffic and the error traffic are in the connection structure, and because the primary difference is in the size of requests to the server, I conclude that the changes in response time are because of a change in the *nature* of



(a) request size

(b) response size

Figure 5.52: The median request and response sizes, with 5/95 percentile error bars, by ordinal, for the normal and "error" periods of the departmental web server.

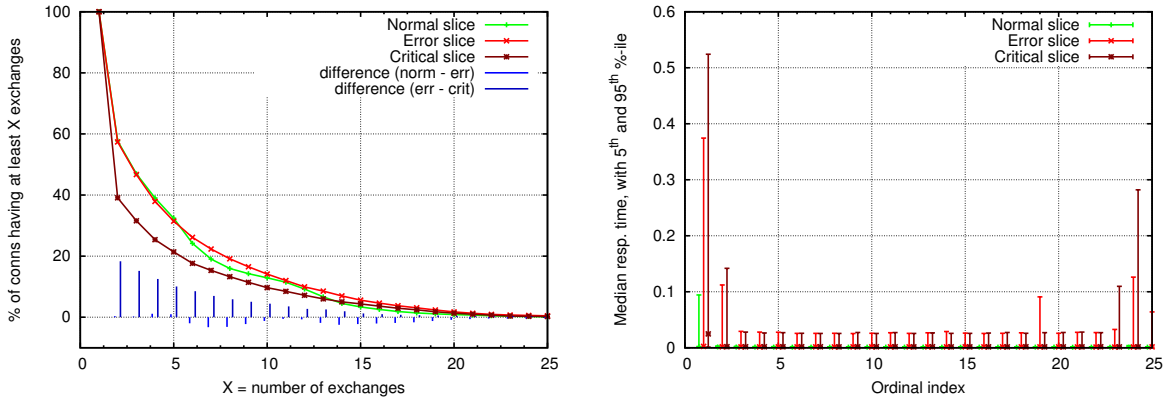
the server's traffic.

Now we turn our attention to diagnosing the "critical" issue by comparing the "error" period with the "critical" period. Again, I will omit the many metrics that do not have a significant difference, which is the same list as for the previous diagnosis of this section: the number of connection attempts, the number of connection failures (both rejections and ignores), number of requests and responses, total size of requests and responses, number of unique clients per day, and the number of open connections at any given time.

When comparing the "error" and "critical" periods, the difference is in the first exchange. First, Figure 5.53(a) shows that fewer connections last longer than a single exchange in the "critical" period. Second, Figure 5.53(b) shows that the primary difference in response times between the error period and the critical period is in the first ordinal.

Also, Figure 5.54 shows that the request sizes and response sizes match well between the "error" and "critical" periods for all ordinals, showing that, unlike the previous diagnosis, there is no substantial change in the nature of the requests and responses to account for the longer response times.

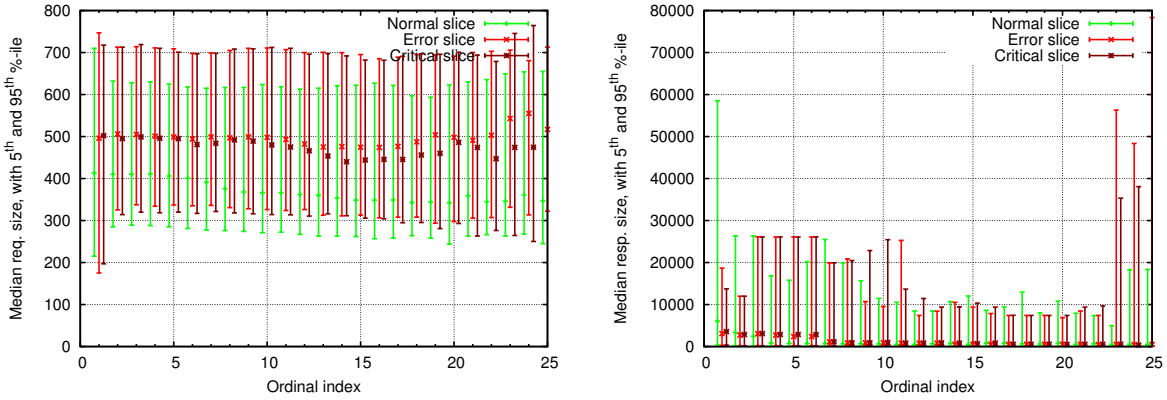
Figure 5.55 shows the rolling-day anomaly detection results. The overall results mirrors Figure 5.43, with a `error` anomalies beginning February 7 and `critical` anomalies beginning March 5. Figure 5.55(b) shows that, when there are anomalies in the overall analysis, there are also anomalies in a first-ordinal analysis, which makes sense because first response times



(a) The number of connections having X exchanges for the normal, "error", and "critical" periods of the departmental web server.

(b) The median response time, by ordinal, for the normal, "error", and "critical" periods of the departmental web server.

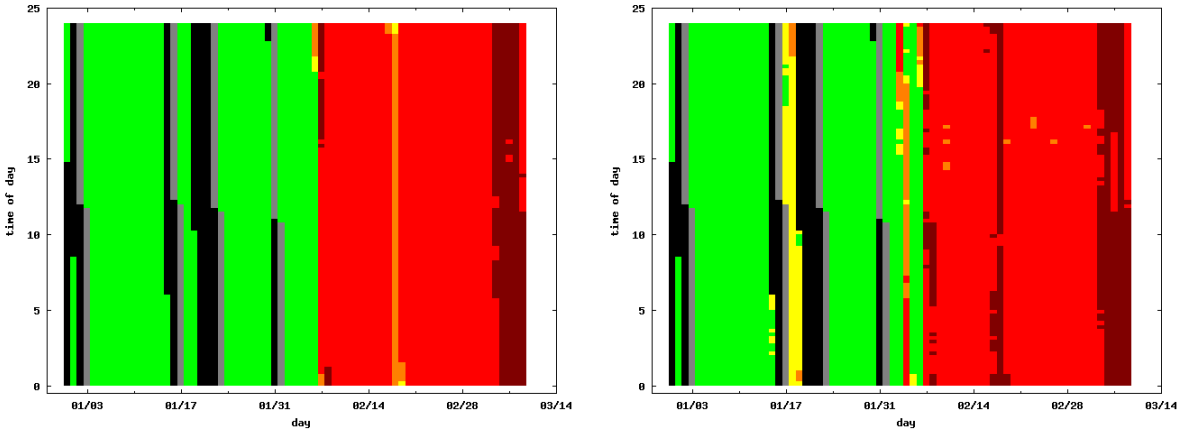
Figure 5.53: Results from an ordinal analysis of the normal period versus the "error" period versus the "critical" period for the departmental web server.



(a) request size

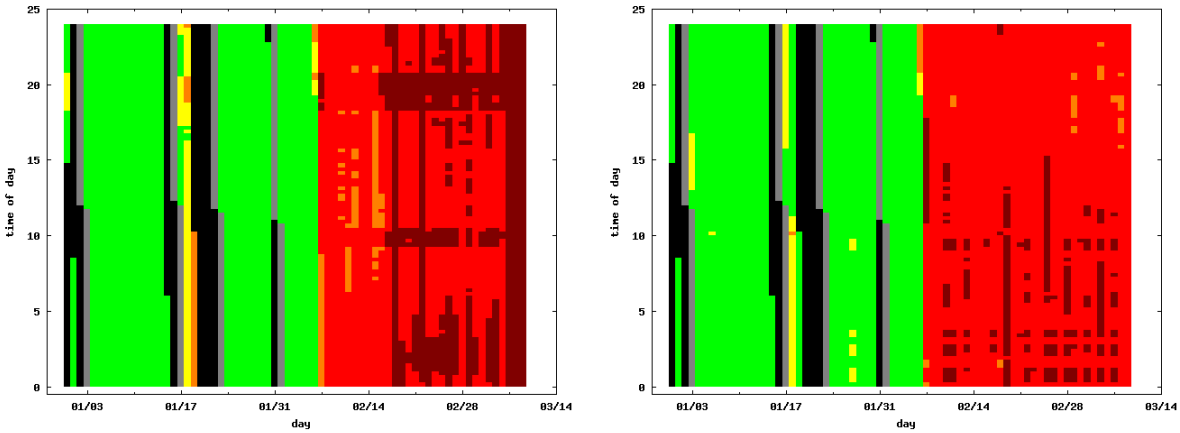
(b) response size

Figure 5.54: The median request and response sizes, with 5/95 percentile error bars, by ordinal, for the normal, "error", and "critical" periods of the departmental web server.



(a) overall (b) first ordinal

Figure 5.55: Anomaly detection results with a rolling-day time-scale for the departmental web server.



(a) second ordinal (b) third ordinal

Figure 5.56: Anomaly detection results with a rolling-day time-scale for the departmental web server.

comprise roughly one fifth of all response times. There are some anomalies with the first response time around January 17 and February 4 that I will not explore here.

Figure 5.56 shows the results of the anomaly detection of the second and third response times in each connection. In general, they agree fairly well with the overall anomaly detection. Note that the higher the ordinal number, the fewer samples there are in the distributions, and the methods are then subject to sampling error.

Figure 5.56(b) deserves particular attention because it demonstrates a shortcoming of the rolling-day technique. Recall from Section 4.9 that each test distribution for the rolling-day timescale is comprised of ninety-six quarter-hour distributions joined together. Ninety-five of

these quarter-hour distributions remain the same when going from one test distribution to the next; only one quarter-hour distribution is changed at each quarter-hour advance of the day-long window. Then, the test distribution at one window location (*e.g.* 11:45 a.m. to 11:44 a.m. the next day—the y-axis on the heat-map plot) is tested against a basis defined from the historical distributions *at the same window location*. Over time, each of these ninety-six window locations evolves in the usual way, by incorporating normal test distributions into the basis and rejecting anomalous test distributions from basis according to the `anom_thres` setting.

Note the `critical` anomalies that line up in rows. Note also the yellow blocks on January 27. These are short-lived and seemingly sporadic because the anomaly score is right on the border of the `anom_thres` setting for several hours. Having some windows deemed anomalous and others deemed normal means that the basis begins to diverge, even for neighboring window locations. The distributions during this time were essentially a "preview" of the anomalies to come, and failing to incorporate the preview makes the anomalies more anomalous, which is why they have `critical` anomaly scores. I call this problem *basis divergence*, and I consider it a direction to explore for future work.

In summary, this section attempted to diagnose two issues for a departmental web server. First, the server began seeing `error` anomalies because of a shift in the nature of the traffic—specifically, larger requests and different responses than before. Second, the server began seeing `critical` anomalies that correspond with a longer first response time and more connections with only one exchange. This section again showed the usefulness of the ordinal analysis.

5.7 Case study: departmental SMTP server

The sixth server performance issue that I will explore in detail involves an SMTP server assigned to a department on campus (a different department than the web server of the previous section). This server had an occasional `notice` anomaly in January, and one in February, but Figure 5.57 shows that the `critical` anomalies began on March 5 and continued through March 9, the last day I have analyzed.

Like the departmental web server of the previous section, the traffic statistics for the server were not significantly different for the anomalous period. There was no significant difference

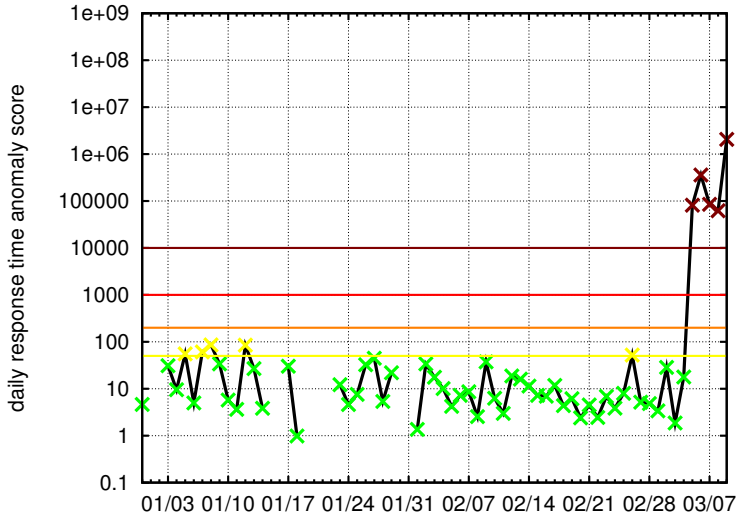


Figure 5.57: Anomaly score for departmental SMTP server. Outages resulting in incomplete days are represented by breaks in the line. Dates are 2009.

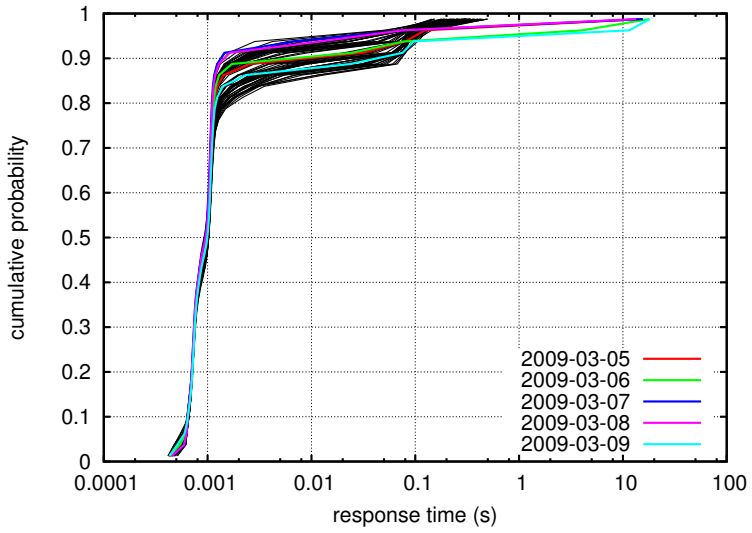
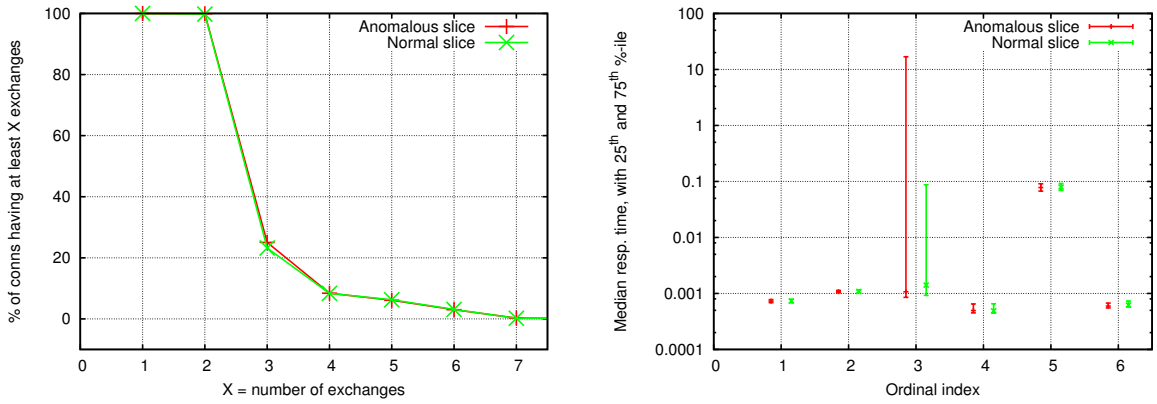


Figure 5.58: Distributions of response time distributions for normal operation (black lines) and anomalous operation (colored lines).



(a) The number of connections having X exchanges for the normal and anomalous periods of the departmental SMTP server.

(b) The median response time, by ordinal, for the normal and anomalous periods of the departmental SMTP server.

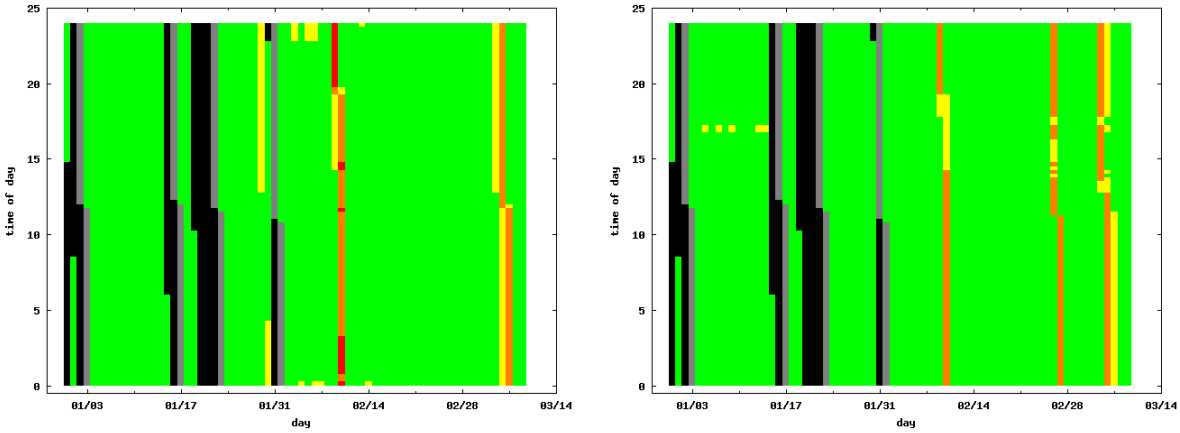
Figure 5.59: Results from an ordinal analysis of the normal period versus the anomalous period for the departmental SMTP server.

among the count of connection attempts, count of connection establishments, count of connection failures, unique client addresses, request/response counts, nor the request/response total sizes. Defining a normal period from Friday, February 20 to Monday, February 23 and an anomalous period from Friday, March 6 to Monday, March 9, the number of exchanges is similar as well (Figure 5.59(a)). Only three percent of connections have a sixth exchange, and less than 0.2 percent have more than six exchanges, so I will ignore ordinals beyond the sixth. The ordinal request size and response sizes are similar as well, and I do not plot them here.

Figure 5.59(b) shows the ordinal analysis of response times for the normal and anomalous periods. The fifth and third response times are the largest in general. The biggest difference is in the third response time, whose third quartile jumps from about 100 milliseconds to over ten seconds—at least a hundredfold increase. Because this is an SMTP server, we know that the third exchange is when the client transmits the first (of one or more) "To:" addresses.

The observation that the first and second response times are similar during the anomaly is interesting. Figure 5.60 confirms this observation: during the anomalous period of March 5–March 9, the normally `critical` anomalies are at worst a `warning` anomaly for the first and second response times. (The `warning` anomalies are explained by the anomalous distribution being heavier in the last quartile, that is, beyond Q3.)

I do not know any more detail about what happened with this server, and I have not spoken



(a) first ordinal

(b) second ordinal

Figure 5.60: Anomaly detection results as a heat-map with a rolling-day time-scale for campus SMTP server.

with its system administrator. However, knowing that the "To:" address is causing the delay is an insight that one could imagine would be helpful for the administrator trying to find the root cause of the problem.

5.8 Case study: unknown service

The seventh server with severe anomalies that I will study in detail has a service running on a port not listed in the assigned ports list (*i.e.* an ephemeral port). The server seems to be affiliated with a department on campus (based on the domain name), but I do not know its function. I will call this server the "unknown" service.

The server had no anomalies except for two multi-day bursts of `critical` anomalies from January 23–25 and February 5–8 (see Figure 5.61). Many of the anomalous response times are lower than is typical, but there are also many response times that are much higher than is typical (see Figure 5.62).

The anomaly score is anti-correlated with the demand for the server, whether the demand is measured in the number of unique clients (Figure 5.63(a)), the number of connection attempts (Figure 5.63(b)), the number of connection establishments (Figure 5.64(a)), the number of failed connection attempts (Figure 5.64(b)), the number of requests (Figure 5.65), or the number of open connections (Figure 5.66(a)). For the latter plot, I used a normal period from

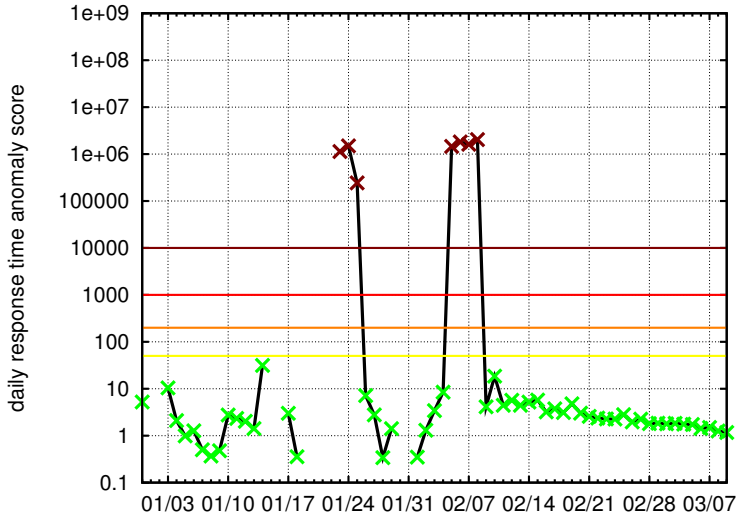


Figure 5.61: Anomaly score for the unknown service. Outages resulting in incomplete days are represented by breaks in the line. Dates are 2009.

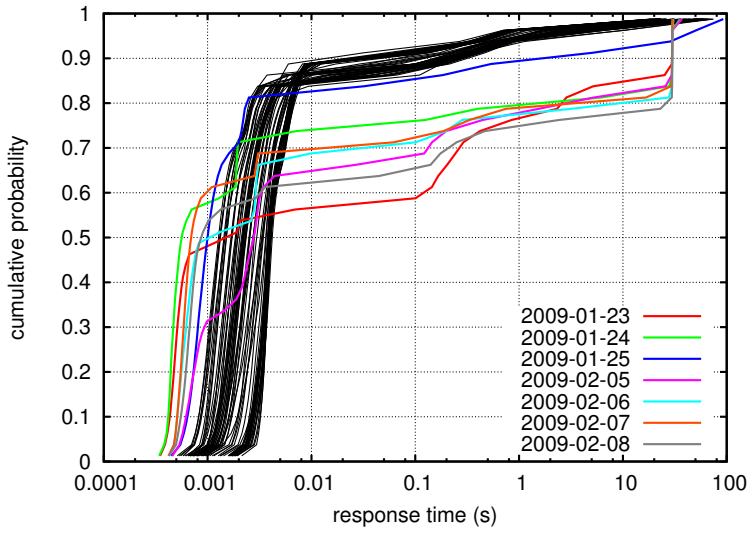
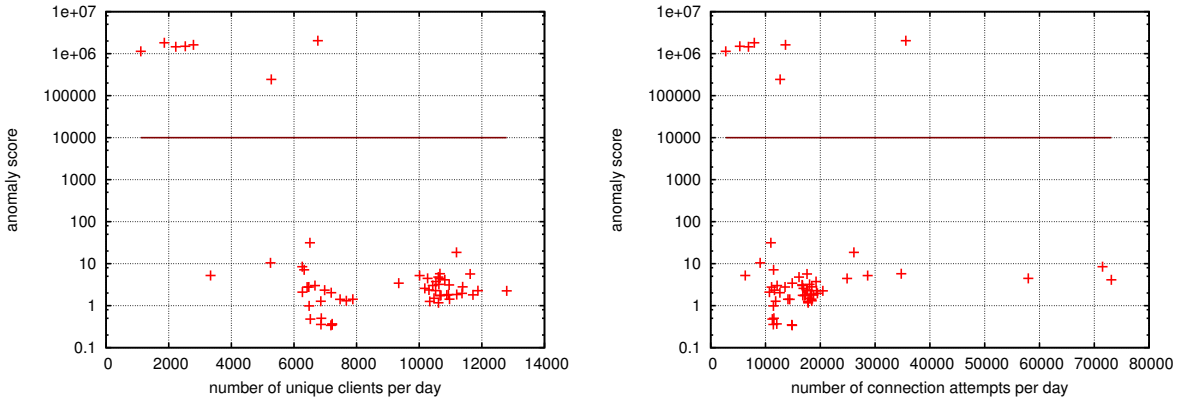
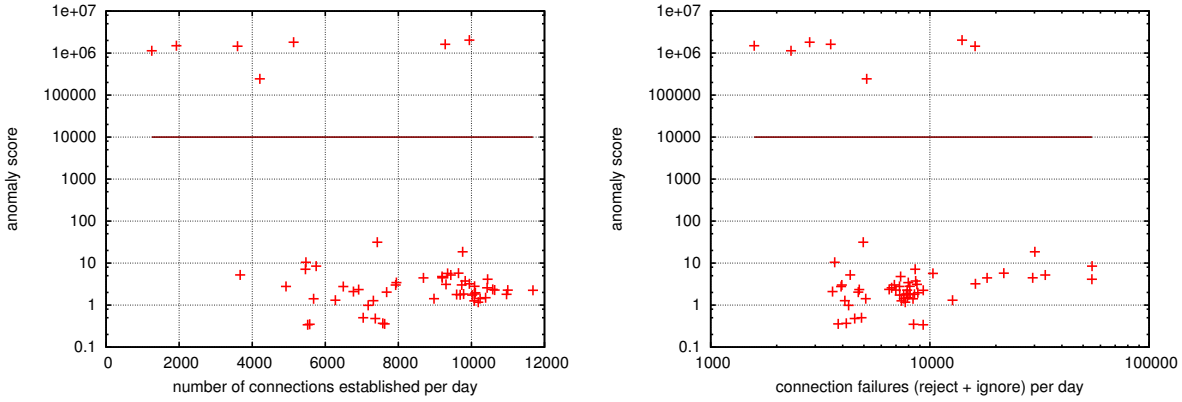


Figure 5.62: Normal response time distributions (black) and `critical` response time distributions (colored) for the unknown service.



(a) unique client addresses (b) connection attempts

Figure 5.63: Number of unique clients attempting a connection per day, and the number of total connection attempts for the unknown service, both plotted versus the anomaly score.



(a) connection establishments (b) connection failures

Figure 5.64: Connection establishments and connection failures per day for the unknown service, both plotted versus the anomaly score.

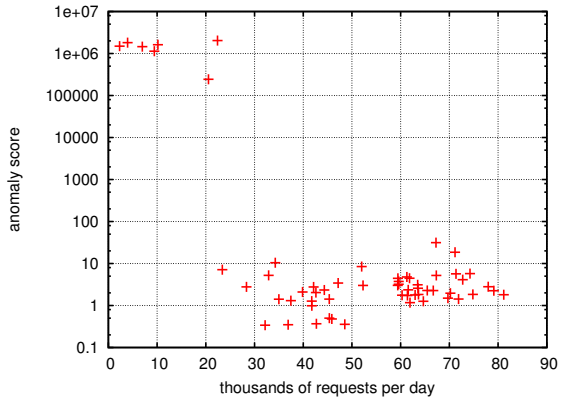
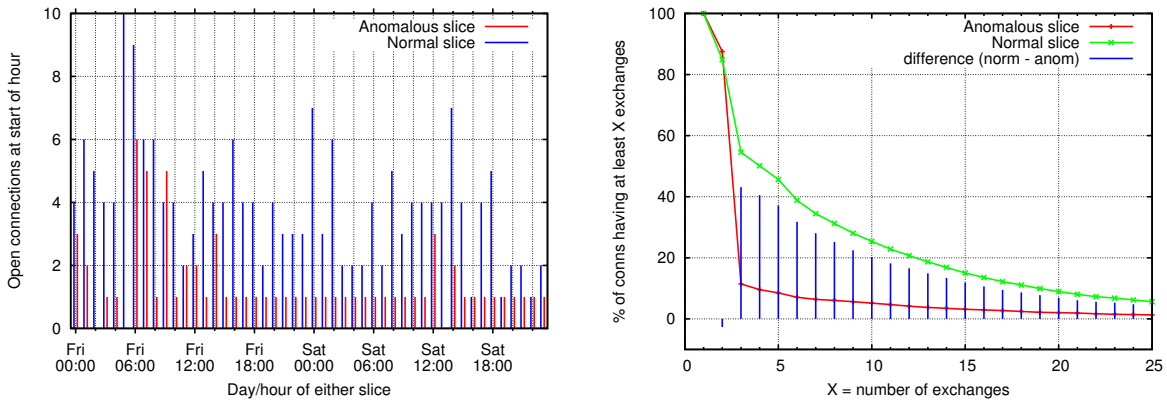


Figure 5.65: Number of requests per day for the unknown service versus the anomaly score.



(a) open connections at the beginning of an hour

(b) proportion of exchanges per connection

Figure 5.66: Comparisons between normal and anomalous periods for the unknown service.

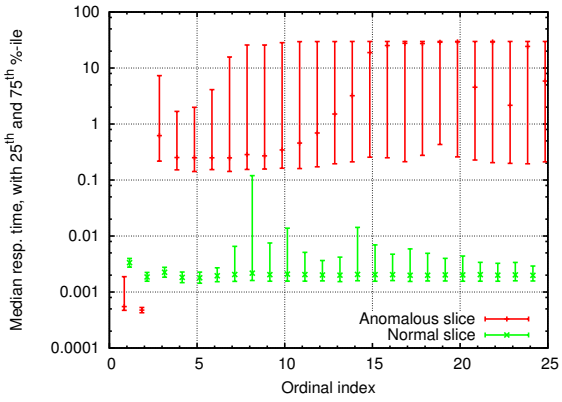
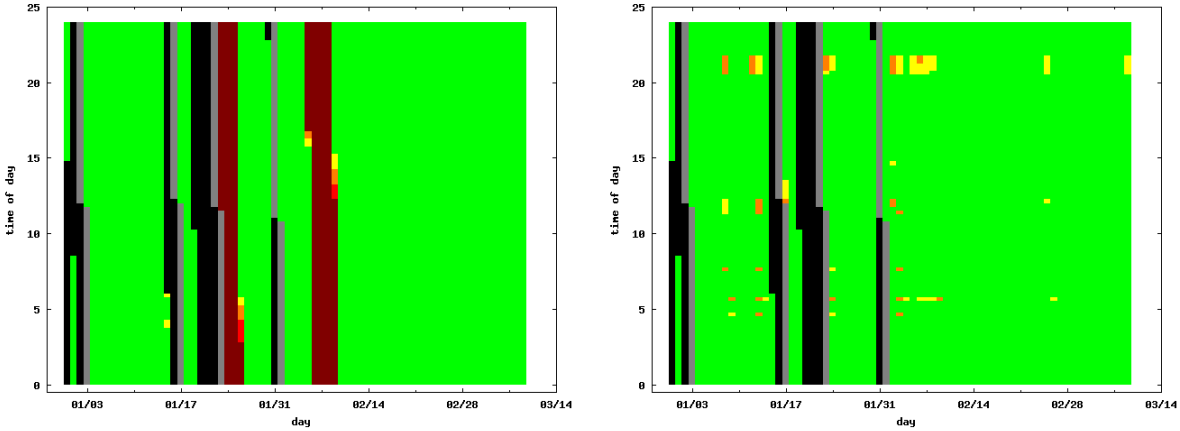


Figure 5.67: Median response time, with Q1/Q3 error bars, for the response times by ordinal for the unknown service.

Friday, January 9 to Saturday, January 10 and an anomalous period from Friday, January 23 to Saturday, January 24. The number of exchanges per connection, on the other hand, does have a significant difference, with 43% fewer connections having three exchanges during the anomalous period (Figure 5.66(b)).

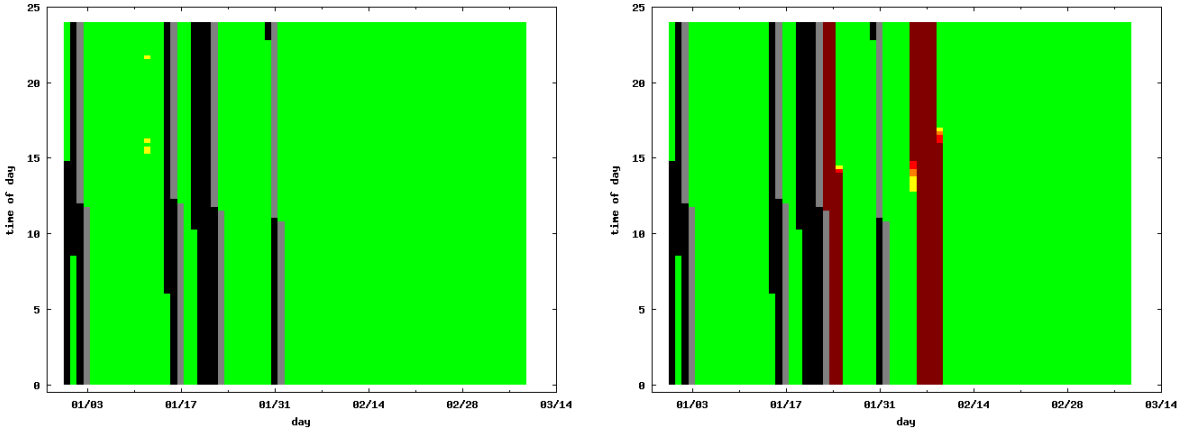
Figure 5.67 shows that the first and second response time per connection are small, but all response times after the second are, on average, quite large. This ordinal breakdown suggests that the third response time and beyond are the principal contributors to the `critical` anomalies. This hypothesis is supported by the rolling-day analysis of Figure 5.68 and 5.69. The overall anomaly detection results match the anomaly detection results for the third ordinal response time; the first and second ordinal response times have few anomalies at any time.

For completeness, I also show the request and response sizes by ordinal in Figure 5.70, even



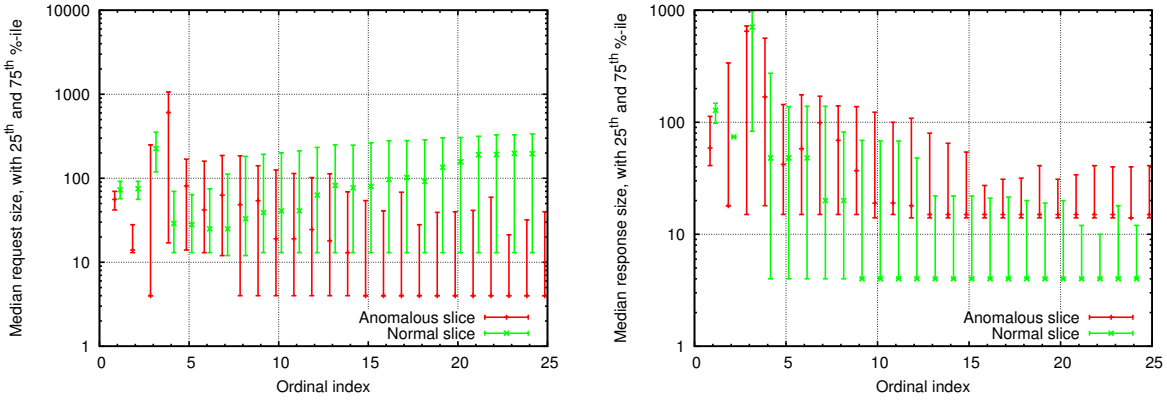
(a) overall results (b) first ordinal

Figure 5.68: Anomaly detection results for the unknown service, both overall and for the first ordinal only.



(a) second ordinal (b) third ordinal

Figure 5.69: Anomaly detection results for the unknown service for the second and third ordinal.



(a) request size (b) response size

Figure 5.70: Median request/response size, with Q1/Q3 error bars, by ordinal, for the unknown service.

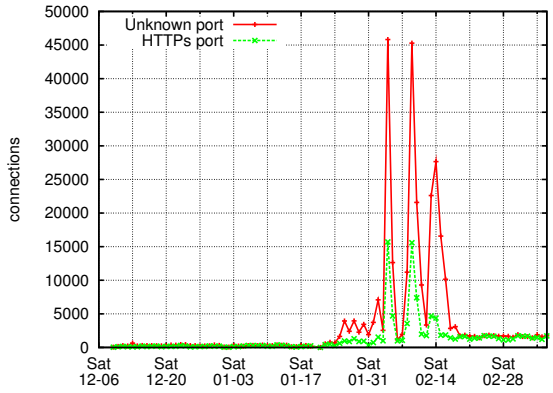


Figure 5.71: Number of connection rejections for both services on the unknown service.

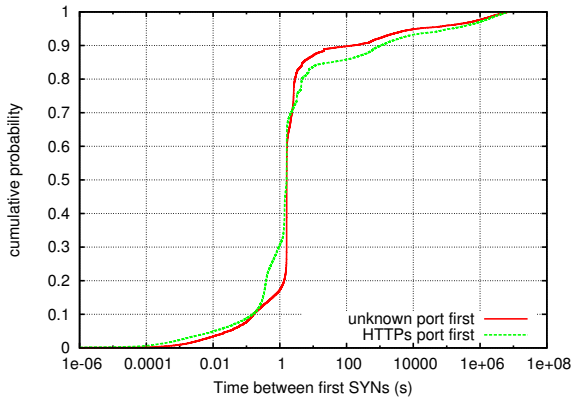
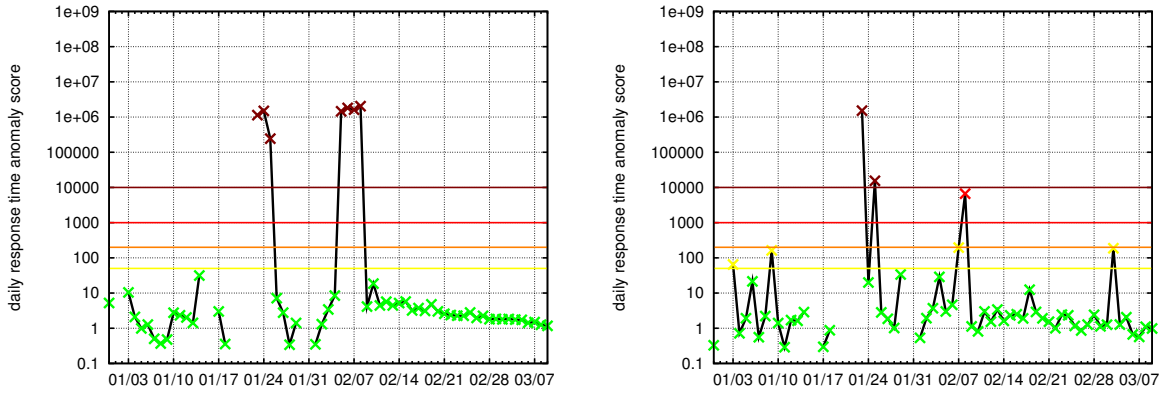


Figure 5.72: Time, in seconds, between a particular client contacting either service.

though I can draw no clear conclusion from the data.

So far, I have shown two things: that the anomalies tend to occur when demand is low, and that the anomalies have longer response times on the whole. The first observation is particularly unusual. Typically, we would expect that an increased load would overload the server, resulting in longer response times. As it happens, however, this system actually has another service running on it on port 443 (HTTPs). The remainder of this section will focus on the HTTPs service.

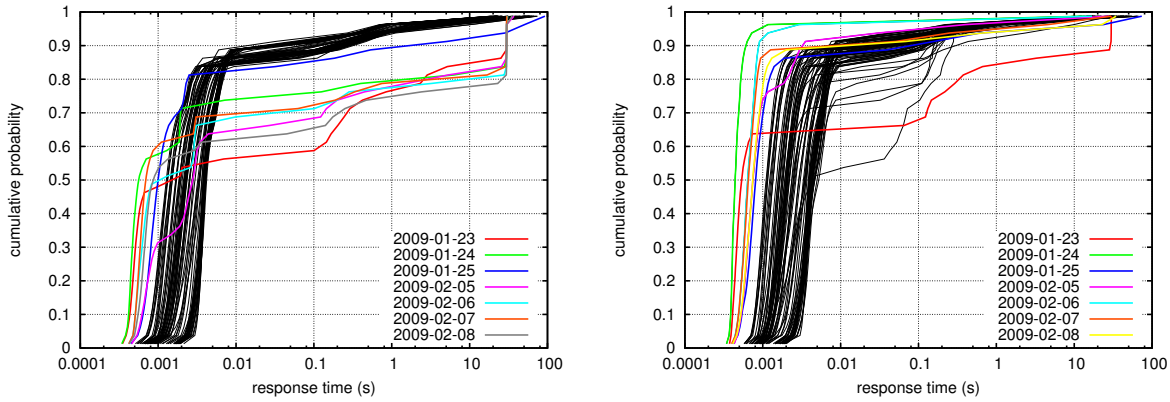
First, the demand on the two services is not independent. Figure 5.71 shows the number of rejected connection attempts for each of the services. Both servers have an unusually high demand on February 4 and February 9. As it turns out, of the 646,052 IP addresses in the entire dataset that initiate a connection to either service, 161,018 (25 percent) initiate a connection to both services. Of these, 83 percent (or 134,263) contact the unknown port first, and 17



(a) unknown port

(b) HTTPs port

Figure 5.73: Anomaly score timeseries for both services.



(a) unknown port

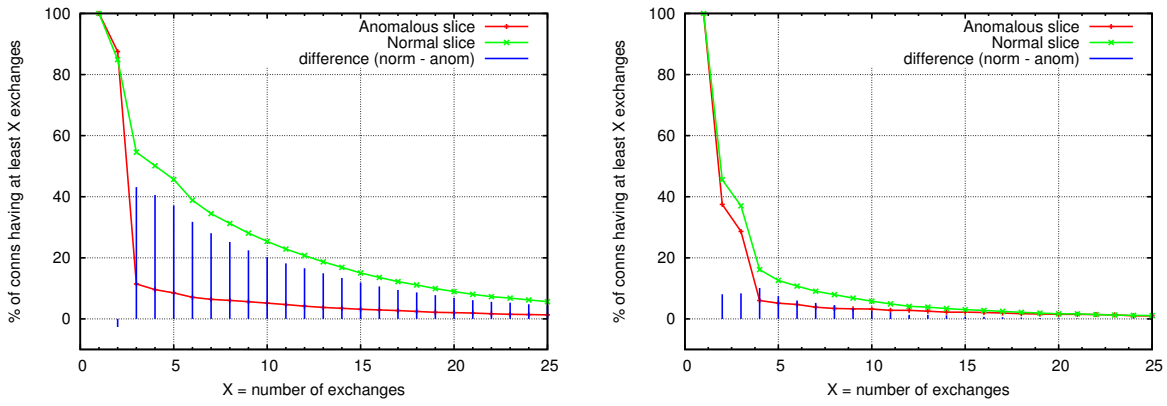
(b) HTTPs port

Figure 5.74: Highlighted response time distributions for both services.

percent (or 26,755) contact the HTTPs port first. In either case, over 80 percent of clients contact both ports within 10 seconds (see Figure 5.72). Thus, it appears that the services are not independent of each other, both because the server resources are shared, and also in terms of clients' usage.

Figure 5.73 shows the anomaly score timeseries for both services. The anomaly on January 23 occurs in both cases, with similar scores. Also, the anomaly on January 25 is `critical` in both cases, although the scores differ by more than a factor of ten. However, in general, the anomalies do not match well between the services.

The reason for this is shown in Figure 5.74, which highlights the response time distributions that were classified as anomalous for the unknown port. This means that some of the highlighted distributions for the HTTPs port were not actually considered anomalous. First, both services



(a) unknown port

(b) HTTPs port

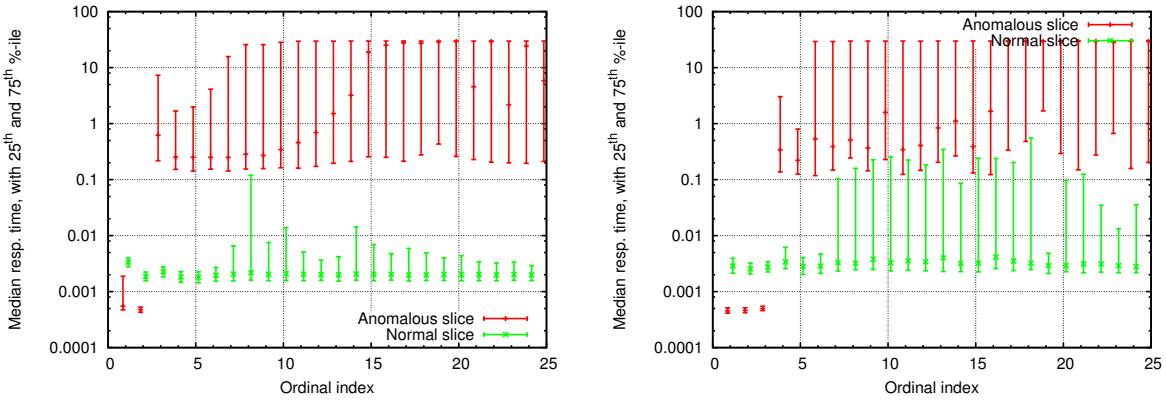
Figure 5.75: Proportion of connections having at least X exchanges, in both services, for both a normal period and an anomalous period.

exhibit a timeout around 30 seconds. The HTTPs server is interesting because the highlighted days—again, those which were anomalous when considering the unknown service—often have lighter distributions than the average. This is a rather confusing result. The lighter distributions suggest that the cause of the anomalies was isolated to only the server running on the unknown port. But the days that were anomalous for both servers suggest that there was a shared cause—perhaps a bottleneck resource that both processes shared, like CPU or RAM.

Is it possible that the unknown port was also speaking HTTP or HTTPs? I cannot know for certain because of the design of my solution, which deliberately ignores application payload data. However, Figure 5.75, which shows the proportion of connections lasting for at least X exchanges, shows that it is more likely that the two services are for different applications. The normal and anomalous periods were the same as in the previous discussion: Friday, January 9 to Saturday, January 10 and Friday, January 23 to Saturday, January 24, respectively.

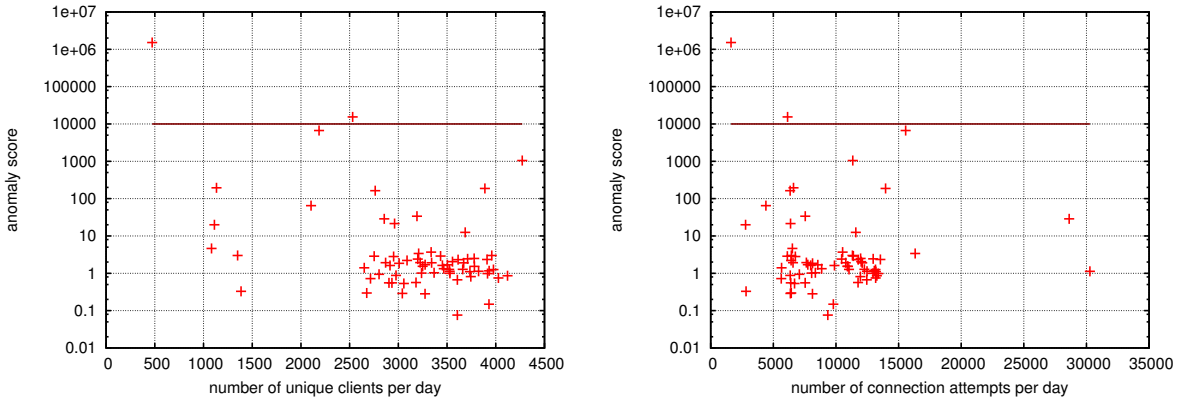
Digging deeper, I plot the ordinal statistics for the response times for both servers in Figure 5.76. In both cases, the first few response times in a connection are smaller during the anomalous period, and all the others are larger during the anomalous period. However, for the unknown server, this is true for the first two response times in each connection, whereas, for the HTTPs server, this is true for the first three response times in each connection. This confirms that the two services are probably serving different applications.

Lastly, I show the same load scatterplots as before, which show that the higher anomaly



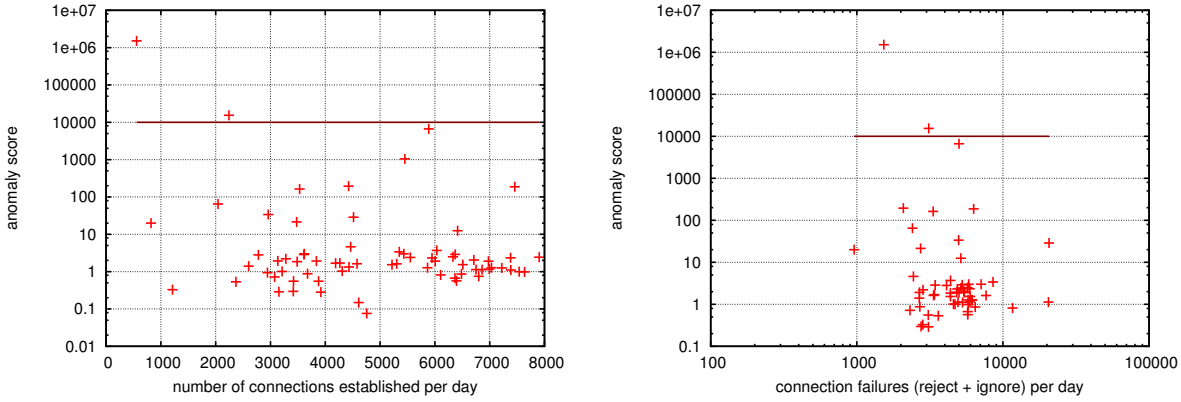
(a) unknown port (b) HTTPs port

Figure 5.76: Proportion of connections having at least X exchanges, in both services, for both a normal period and an anomalous period.



(a) unique client addresses (b) connection attempts

Figure 5.77: Number of unique clients attempting a connection per day, and the number of total connection attempts for the unknown service, both plotted versus the anomaly score.



(a) connection establishments (b) connection failures

Figure 5.78: Connection establishments and connection failures per day for the unknown service, both plotted versus the anomaly score.

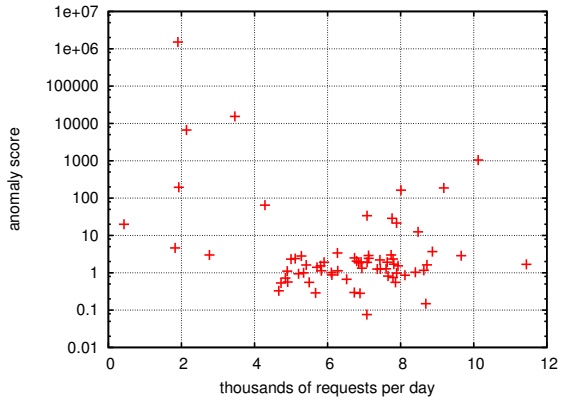


Figure 5.79: Number of requests per day for the unknown service versus the anomaly score.

scores generally occur when load (as measured in various ways) is lower; that is, load and anomalous performance are anti-correlated. This is shown by connection attempts (Figure 5.77), connection establishments and connection failures (Figure 5.78), and the number of requests (Figure 5.79).

In summary, just as I found no clear explanation for why the unknown service was experiencing anomalous performance, I also found no explanation among the HTTPs service measurements. This probably means that the cause does not manifest in the data collected by `adudump`. For example, if the administrator of the servers' system was running a memory-intensive application on the system at the time of the anomalies, then I cannot know it.

5.9 Case study: server with strong outliers

I now turn my attention to a server with several `critical` anomalies that reveals a shortcoming of my anomaly detection methods. This server ran on an uncommon port, and the domain name suggests that it is affiliated with a non-academic department on campus. I will refer to the server as the "non-acad" server.

Consider Figure 5.80. This figure is curious. Many of the anomalies—especially the `critical` ones—last for exactly twenty-four hours. For example, consider the anomaly on January 25. The anomaly score goes directly from `normal` to `critical`, without any transition. This fact suggests that the quarter-hour bin just added to the window was either so anomalous that it shifted the combined distribution strongly, or else there were many more

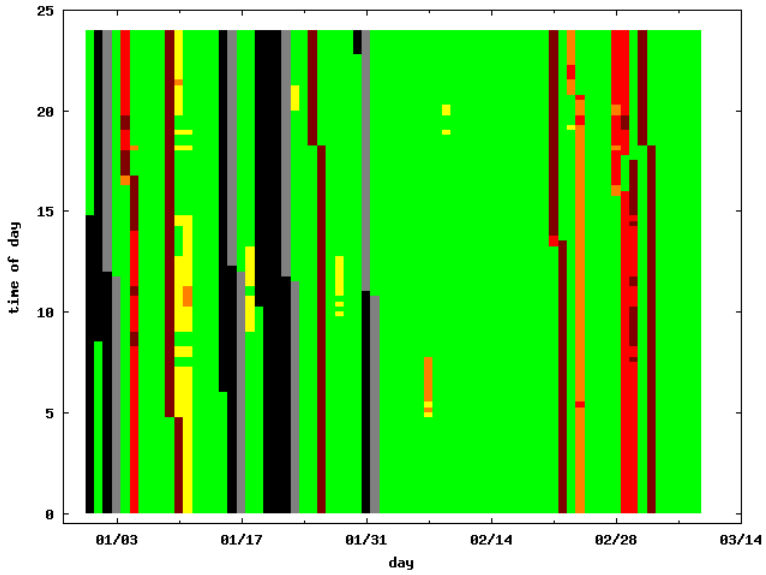


Figure 5.80: The anomaly detection results with a rolling-day timescale for the non-acad server.

data points in the new quarter-hour bin, thus overshadowing the points already in the window. I will call the quarter-hour bin of interest the anomalous bin.

Figure 5.81 shows the distribution comprising the anomalous bin, plotted along with the other quarter-hour bins in the window. Typically, the longest response time in a distribution is about forty milliseconds. However, about five percent of response times in the anomalous bin are about thirty seconds. These are severe anomalies. As it happens, all of these outlying response times are from a single client. Furthermore, all of the major anomalies for this server are similarly the fault of a burst of outliers.

I think it is good that the anomaly detections methods on a rolling-day time-scale detect such an anomaly. However, it would be useful to augment the methods with some indication to the network manager that the anomaly is the fault of a single quarter-hour bin. I leave this augmentation for future work.

5.10 Limitations

A discussion of the limitations of my methods is in order. First, although I consider the single vantage point an overall advantage of my approach, it is a double-edged sword. There are two disadvantages of having a single, passive vantage point. First, when attempting to diagnose

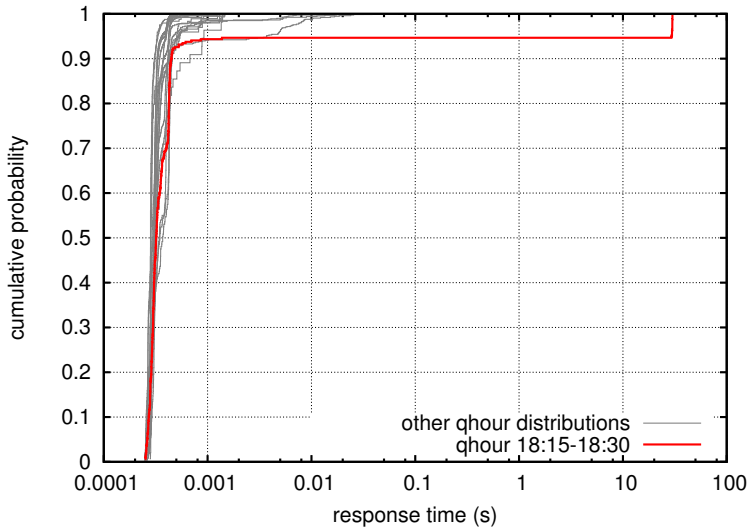


Figure 5.81: The quarter-hour response time distributions prior to the anomalous distribution (black), and the quarter-hour anomalous distribution (red), for the non-acad server.

an anomaly, I only know of the traffic flowing past my vantage point. A significant amount of traffic might be originating on other paths, increasing the load of the server, yet I will never know. Second, when my methods discover a performance anomaly, I cannot know for certain whether the increased response times are actually because of a less responsive server, or perhaps a very congested network path with long queueing delays between the vantage point and the servers. However, I assume that other tools exist to manage the performance of the network itself and detect such symptoms as packet loss in the network; furthermore, if a network problem manifested as a server performance problem, it is likely that other servers sharing the problematic network path would also exhibit performance problems, thus indicating a problem with the shared resource.

Another limitation of my approach is that I do not have visibility into the servers. For example, CPU utilization, page faults, queue lengths, *et cetera* are not metrics I can measure with my approach alone.

Lastly, as seen in Sections 5.5 and 5.4, my diagnosis ability is fundamentally limited by the application-independent approach. My methods were unaware of any "filter process" on the SMTP server, and I could not have known that traffic was being encrypted between processes on the portal server. However, I argue that even with such a limitation, the information my methods can provide to a human server administrator is enough to help the administrator

quickly isolate the cause of the problem.

5.11 Chapter Summary

The contributions described and used in the case studies of this chapter include:

- the ability to detect performance issues in the network, using a completely passive approach
- the ability to detect a variety of performance issues for a variety of types of servers and response time distributions
- the ability to diagnose, at a high level, the cause of an anomaly, by examining in detail how the characteristics of protocol or usage change during the anomaly
- the ability to analyze server performance at a variety of time-scales
- the ability to *quickly* detect performance issues
- a method for comparing the abstract *structure* of an application-level dialog between two sets of traffic, in order to describe anomalies
- the ability to *tune* the anomaly detection method by adjusting parameters
- the ability to *manually* select a definition of normal performance

Chapter 6

Comparison Study

The previous chapters have introduced my methods for detecting and diagnosing performance anomalies of network servers. This chapter explores an evaluation study of the methods.

6.1 Introduction

In the previous chapter, I showed that my methods were capable of detecting legitimate performance issues, and furthermore that the measurements of `adudump` were useful in diagnosing these issues. This chapter focuses on evaluating my methods against another server performance management approach. Calculating a false positive or false negative approach is not feasible because neither approach is authoritative. However, comparing the two approaches in terms of the performance issues identified by each will illustrate the weaknesses and strengths of my approach.

The evaluation of this chapter is a comparison between my methods and Nagios, a tool that can identify certain types of server performance issues. I discuss a set of Nagios log files and the information they contain in the next section, as well as the setup of `adudump` that I use in this chapter. Section 6.3 shows how I identify performance issues from the Nagios logs, and how I use my methods to identify performance anomalies. Section 6.4 compares the Nagios issues with the anomalies and attempts to explain the various causes for why the sets do not align. Lastly, Section 6.5 discusses the implications of this evaluation study and concludes.

6.2 Data

To evaluate my performance anomaly methods, I compared the results against several months of log files from Nagios. Nagios is "a powerful monitoring system that enables organizations to identify and resolve IT infrastructure problems before they affect critical business processes."[1] Nagios monitors applications, services, operating systems, network protocols, system metrics, network infrastructure, and other components. Various modules perform different types of monitoring, such as pinging an end-system, measuring the CPU or memory utilization, ensuring certain processes are running, *etc.* Nagios can also submit legitimate requests for a variety of services, and measure the response time of such requests.

Nagios log files record a variety of events, including Nagios information and warnings, checkpoint information when log files are rotated, notifications, external commands, and service alerts. Only a small subset of the information was useful for the evaluation of this chapter. Example records from a Nagios log file are shown below, with sensitive information such as server names masked.

```
[1228021200] LOG ROTATION: DAILY
[1228021200] LOG VERSION: 2.0
[1228021200] CURRENT HOST STATE: <HOSTNAME>;UP;HARD;1;PING OK - Packet loss =
 0%, RTA = 0.12 ms
[1228021200] CURRENT SERVICE STATE: <HOSTNAME>;memory;OK;HARD;1;49%
[1228165078] SERVICE ALERT: <HOSTNAME>;HTTP <SERVICENAME>;WARNING;SOFT;1;HTTP
 WARNING: HTTP/1.1 200 OK - 10.016 second response time
[1228165138] SERVICE ALERT: <HOSTNAME>;HTTP <SERVICENAME>;WARNING;SOFT;2;HTTP
 WARNING: HTTP/1.1 200 OK - 10.001 second response time
[1228165198] SERVICE ALERT: <HOSTNAME>;HTTP <SERVICENAME>;WARNING;HARD;3;HTTP
 WARNING: HTTP/1.1 200 OK - 10.660 second response time
[1228165198] SERVICE NOTIFICATION: <USERNAME>;<HOSTNAME>;HTTP <SERVICENAME>;W
ARNING;notify-by-email;HTTP WARNING: HTTP/1.1 200 OK - 10.660 second response
 time
```

[1] http://nagios.org/

All of the records have a corresponding timestamp at the beginning of each line. The `SERVICE ALERT` records show an example of an HTTP server that was experiencing a performance degradation. Nagios measures the response time by sending a request to the server, and it discovers that the response time was above some threshold value. The first two alerts were "soft" alerts, but the third alert was a "hard" alert, and it results in Nagios attempting to notify a particular user of the issue. By default, Nagios sends a request to each server every five minutes. If the response time is above the threshold (*i.e.* a "soft" alert), then Nagios begins to send a request every one minute. The next section will discuss in detail which records in the log files identify relevant performance issues.

The Nagios log files I will use in this section cover December 9, 2008, to March 9, 2009. This instance of Nagios monitors seventy-one servers within the domain of the web systems group at UNC. There were a total of 51,550 records in 4.5 megabytes of uncompressed textual log files.

For the results in this chapter, I used a different monitoring vantage point. Instead of monitoring traffic from the border link of campus, I monitored the root of the server VLAN. Almost all of the servers within UNC, including the servers managed by the web systems group, were assigned to the server VLAN. This means that all requests to a web systems server (except for those sent by other servers with the same VLAN) pass through the root of the server VLAN, and thus were captured by the `adudump` process running on the server VLAN. The border link vantage point would miss any traffic that was sent by a user within the UNC network.

The packets were passively copied to the monitor by a port-mirroring process running on the router at the root of the server VLAN. The `adudump` logic had to be enhanced to deal with such an access method, because there is no inherent directionality with such copied packets, so it is not always possible to determine whether a packet was going into or out of the server VLAN.

6.3 Methodology

I have two fundamentally different sources of data that I will be comparing. The first set was from Nagios, and the second set was from `audump`. The first step in comparing the two datasets was identifying the overlap. I will discuss this in two parts: identifying service performance

issues from Nagios logs and detecting performance anomalies of web system services.

Identifying service performance issues from Nagios logs

Many different Nagios events were recorded in the log files. Only a small subset of these events were of interest to the evaluation of this chapter. To identify which Nagios events correspond to service performance issues of web systems services, I iteratively filtered irrelevant records with the Linux "grep" utility. As more records were filtered, eventually only the records of interest were left.

Tables 6.1 and 6.2 summarize the results of filtering done at each step. Step 1 filtered information related to the Nagios process itself. Step 2 filtered the state checkpoints that occur every day, when the log files were rotated. Step 3 filtered notification records, which reflect an attempt by Nagios to contact an administrator, and external commands issued by an administrator through Nagios. Step 4 reflects records related to fault management, which is outside the scope of my methods. Step 5 filters records that report that a service experiencing a performance issue has resumed normal operation. I filter these records because I am not concerned here with the duration of the issues, only the time at which they occurred. Step 6 filters measurements that require administrative access to the system on which a server is running, which also is outside the scope of my methods. Step 7 filters service flapping records, which occur when the behavior of a service is operational but fluctuates by a relatively large amount. Step 8 filters a set of records that have an unknown purpose. These records might have been from a custom plugin, because the format of these records break the typical convention in a few ways, and furthermore the word "consumption" is misspelled "consuption". In any case, these records were probably irrelevant. Step 9 is also related to fault management, reflecting time-out events from unresponsive servers. (Of course, time-out events could also reflect a server that takes a very long time to respond. I filter these records anyway, because I cannot determine which of these situations actually occurred.) Step 10 filters errors that were found by examining the content of an HTTP response; they require special information beyond what my methods access. Step 11 is concerned with clusters of multiple servers providing a service. Step 12 filters information relevant to network management but not to server performance management. Step 13 filters unknown records from some "alarm" module. Lastly, Step 14

Table 6.1: iterations of filtering Nagios records

step	type	strings	matches	left
1	Nagios info	LOG ROTATION LOG VERSION Auto-save Caught SIG Finished daemonizing Nagios 2.9 starting Successfully shutdown] Warning:	2630	48606
2	Checkpoints	CURRENT HOST STATE CURRENT SERVICE STATE	39441	9165
3	Notifications	SERVICE NOTIFICATION EXTERNAL COMMAND HOST NOTIFICATION	4479	4686
4	Fault management	HOST DOWNTIME ALERT SERVICE DOWNTIME ALERT HOST ALERT Connection refused	807	3879
5	OK status	;OK;	1402	2477
6	Invasive measures	;PROCS; ;Puppet; ;LOAD; ;FILESYS; ;cpu; or ;CPU; ;memory; ;MDSTAT; ;JAVA memsize; ;global status; ;SYMLINKS; ;LDAP; ;METASTAT;	1143	1334
7	Service flaps	FLAPPING	74	1260
8	unknown tests	Check Test Check dev Prod Contrib Production	171	1089
9	timeouts (fault mgmt)	timed out Timed Out Socket timeout after	615	474

Table 6.2: iterations of filtering Nagios records (continued)

step	type	strings	matches	left
10	HTTP content	No data received from host redirection creates an infinite loop HTTP CRITICAL - string not found 500 Internal Server Error 503 Server Error 503 Service Temporarily Unavailable 503 Service Unavailable 403 Forbidden 404 Not Found	104	370
11	cluster info	cluster	23	347
12	network info	;DNS; ;PING; No route to host	63	284
13	unknown	;alarms;	2	282
14	soft issues	SOFT	243	39

filters "soft" alerts of server performance. These were cases in which a single response time exceeds the threshold. They can be indicative of a legitimate performance issue, but they were more likely to reflect an unlikely but normal response time. The managers were not notified of such soft alerts, and I will likewise ignore them in my analysis.

In the end, 39 service performance alert records were left on 11 servers. All of these records were service alerts recording that the response time of an HTTP request took longer than some threshold value. The default threshold is ten seconds, but, for some services, members of the web systems group have selected a longer or shorter threshold. The lowest threshold in the logs is five seconds, so that is the threshold I will use for the analysis of the next section. Table 6.3 lists the salient information available in each service performance alert record.

Detecting web system server performance anomalies

After identifying the performance issues from the log files of the Nagios instance of the web systems group, I need to also identify performance issues from the `adudump` instance running on the server VLAN for the same servers. I am interested in any server that Nagios monitors, so I first identified seventy-one server names from the Nagios logs. I associated these server names with IP addresses (with some help from the web systems group). I then selected all `adudump` records that involved one of these IP addresses, regardless of the port, which resulted in 57 GB

Table 6.3: salient information from service performance alert records

name	example	description
timestamp	1228097713	time of record as a unix epoch
hostname	web025	the short hostname of the server
given name	HTTP jboss	the name (mnemonic) given to the server within Nagios
severity	WARNING or CRITICAL	presumably this is which threshold the response time exceeded
level	HARD or SOFT	typically, the first two alerts were SOFT, then the third is a HARD
count	3	the number of alerts for this server over some window
HTTP response	HTTP/1.1 200 OK	the response code from the HTTP server
response time	7.736	the measured response time that exceeds the threshold

of textual `adudump` data.

One weakness of my methods is that I cannot reliably detect anomalies if there are not enough observations to create a profile of normal behavior with minimal sampling error. (I have not quantified the role of sampling error in my methods; that is a topic of ongoing research.) Therefore, I have two *frequency thresholds* that a server address/port pair must pass before I analyze its data for anomalies. I chose a threshold of at least 1,000 requests or responses per day. If a server address/port pair exceeded this threshold for at least 90 percent of the 87 days (or 78 days) of `adudump` data, then I included it in the next step. (All of the servers that were included exceeded 1,000 requests in at least 85 of the 87 days.) I identified a total of twenty-one such server address/port pairs on seventeen server addresses. It is difficult to quantify how many servers were excluded, because I cannot reliably determine directionality from the server VLAN vantage point and thus cannot distinguish clients from servers.

For these twenty-one address/port pairs, I executed my anomaly detection methods with the following parameter settings: `basis_error` = 0.01, `window_size` = 60, `tset_size` = 21, `num_bins` = 40, and `anom_thres` = 1000. I chose a high setting of `anom_thres` because I wanted to focus on only the most obvious anomalies.

6.4 Evaluation

Of the 71 servers in the Nagios logs, 11 of them had a total of 39 performance issues, as identified using the methods discussed in the previous section. Next, I want to compare these issues against the anomalies detected by my methods. However, nine of these servers (comprising 37 of the 39 issues) did not pass the frequency thresholds. This leaves only two servers (each with one issue) to analyze. However, even these two issues did not have a corresponding anomaly as identified by my methods (although one of the servers had an anomaly on a later date). Later in this section, I will analyze the anomalies found by my methods that had no corresponding issue identified by Nagios. First, however, I examine these 39 issues that Nagios reported from the perspective of the data collected by `adudump`.

Note that I cannot calculate a false positive or false negative rate. Nagios does not represent the final truth for these servers. It sometimes identifies issues that network managers do not care about, and, conversely, important issues can sometimes be missed. Therefore, a "false positive" or "false negative" when comparing my methods against Nagios logs is not necessarily "false". Furthermore, since Nagios identifies performance issues based on a small number of single response time measurements compared to a fixed threshold, whereas my methods identify anomalous distributions of response times, comparisons between the two should be carefully qualified.

Analyzing unmatched Nagios issues

Of these 39 issues on 11 servers, 26 occurred on 2 servers that had very little traffic seen by `adudump` for any port. One of these servers, which I will name `alpha`, had 25 performance issues identified in the logs, and the other server, which I will name `beta`, had one. (I give false names to the servers throughout this chapter.) In the case of each of the 26 issues, there was no data seen by `adudump` in the hour surrounding the issue. For example, see Figure 6.1, which plots the number of response time observations per hour for `beta` around the time of the issue. The issue's timestamp is marked with a vertical red line, and there were no response times in that hour. (There were a few in the following hour, but none were greater than the Nagios threshold of 5 seconds.)

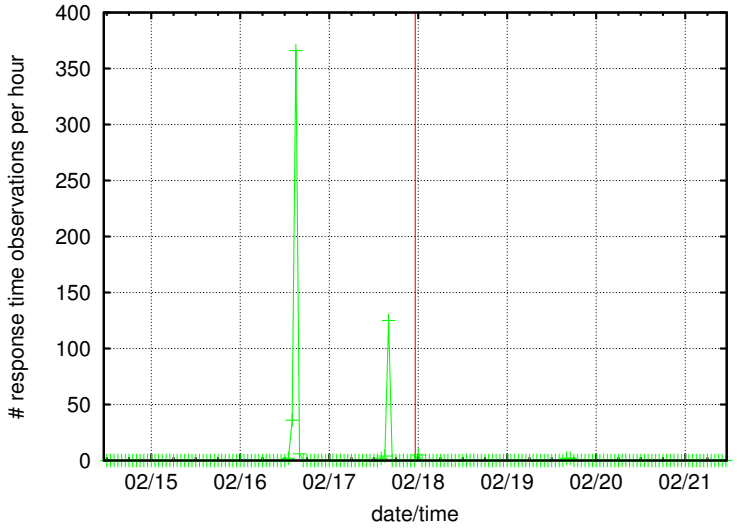


Figure 6.1: Count of response time measurements for `beta` server around the time of its only issue.

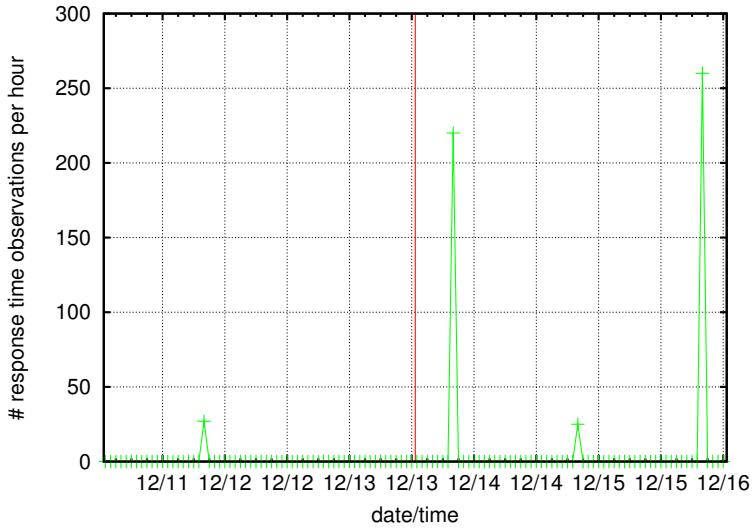


Figure 6.2: Count of response time measurements for `alpha` server around the time of its first issue.

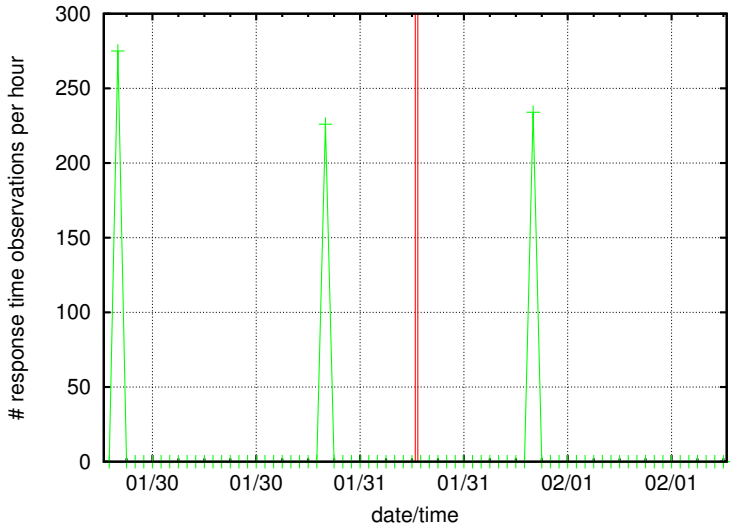


Figure 6.3: Count of response time measurements for `alpha` server around the time of its second and third issues, which were separated by 18 minutes.

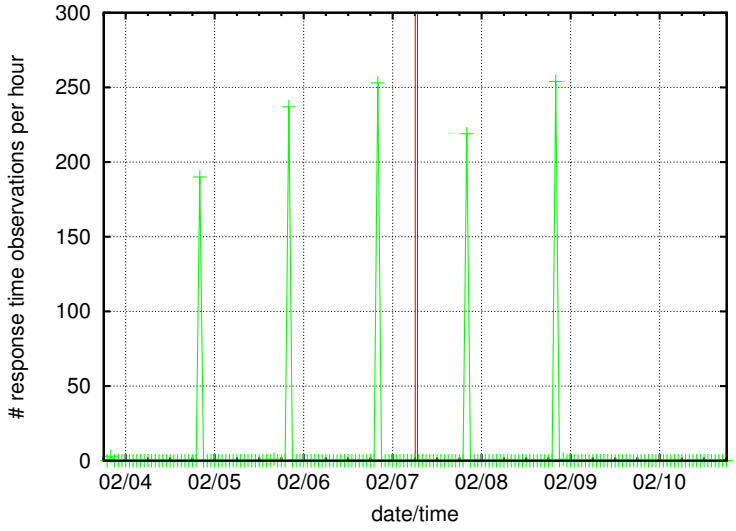


Figure 6.4: Count of response time measurements for `alpha` server around the time of its fourth and fifth issues, which were separated by 37 minutes.

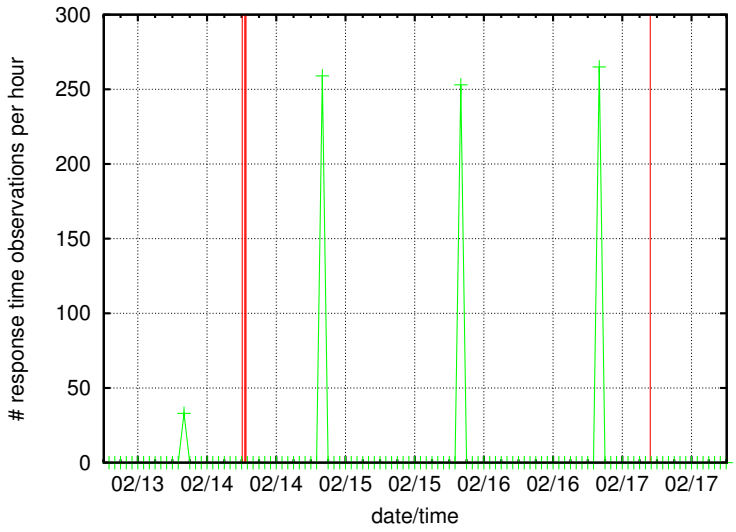


Figure 6.5: Count of response time measurements for `alpha` server around the time of its sixth through ninth issues. The sixth and seventh issues were separated by 27 minutes, and the seventh and eighth issues were separated by 12 minutes.

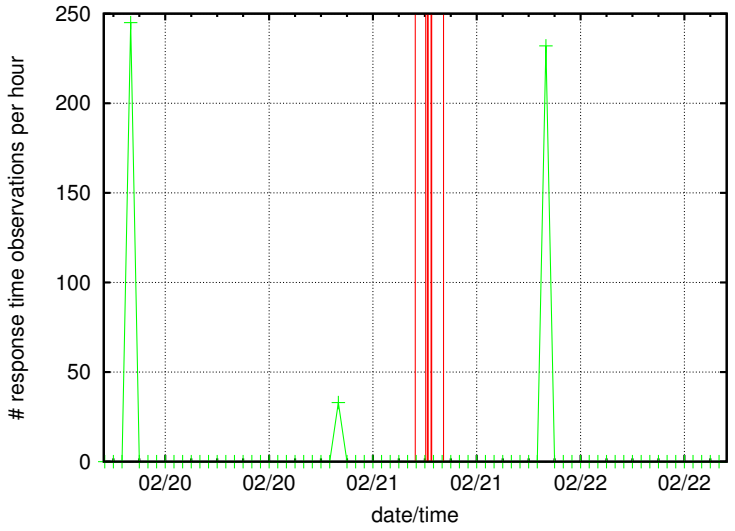


Figure 6.6: Count of response time measurements for `alpha` server around the time of its 10^{th} through 16^{th} issues, which were separated by 72 minutes, 12 minutes, 5 minutes, 20 minutes, 5 minutes, and 82 minutes, respectively.

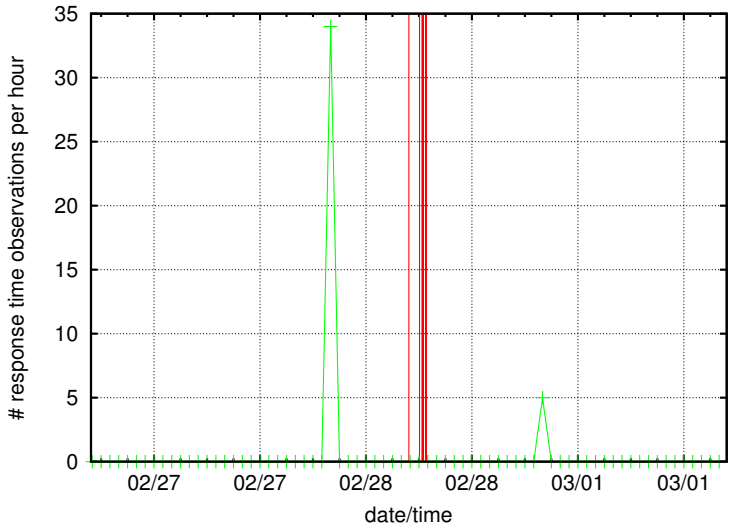


Figure 6.7: Count of response time measurements for `alpha` server around the time of its 17^{th} through 23^{rd} issues, which were separated by 72 minutes, 17 minutes, 5 minutes, 5 minutes, 15 minutes, and 5 minutes, respectively.

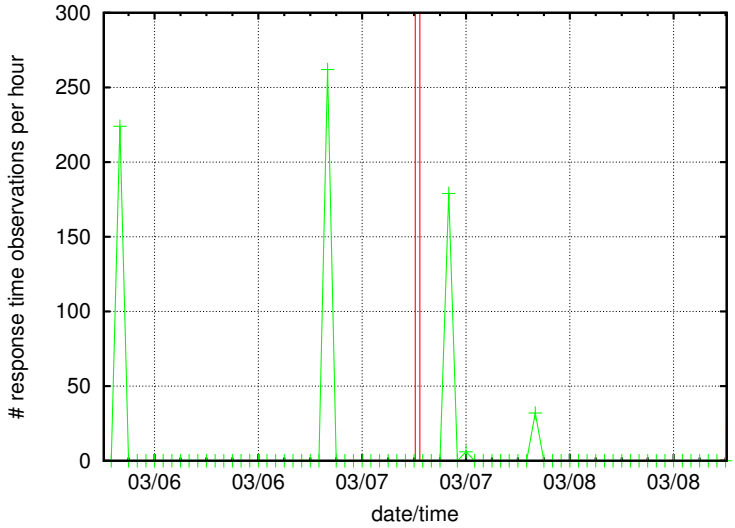


Figure 6.8: Count of response time measurements for `alpha` server around the time of its 24^{th} and 25^{th} issues, which were separated by 32 minutes.

Similarly, Figures 6.2 through 6.8 plot the 25 issues for server `alpha`. Note that the issues often occurred near each other, with only a few minutes of separation between them. This indicates that the issue is more likely to be a legitimate concern.

It is important to consider why `adudump` did not see much traffic for these servers. The reason is that the vantage point of the monitor, at the root of the server VLAN, did not see much traffic for the server. This can occur for several reasons. First, it might be the case that most of the traffic to this server originates from within the server VLAN, in which case the traffic will not flow through the root of the VLAN, and the monitor will not see any packets from these connections. For example, it is clear from the above plots that the requests sent by the Nagios system to the servers was not seen by `adudump`, so I conclude that the packets from these connections did not transit the root of the server VLAN, and the Nagios system is itself within the server VLAN. This is an important point when considering where to place a vantage point, and it illustrates the trade-off between the number of servers measured by the monitor, and the likelihood that all of the traffic for each server was measured.

A second possible reason why `adudump` did not see much traffic for these servers it that there simply was not much traffic for the servers to begin with. This is a fundamental limitation of passive measurement techniques: if there is not much traffic already on the network, then the passive measurement cannot measure much traffic, and the confidence of statistical conclusions is lower. It is important to note that, based on the information I have available, I cannot distinguish between this case and the previous case, because I do not know (and cannot measure) how much traffic is flowing to these servers from other systems within the server VLAN.

A third possible reason why `adudump` did not see much traffic is that the server's IP address might have changed between the monitoring vantage point and the server itself. For example, as packets transit a network address translation (NAT) system, the addresses of connections will be changed. This will typically occur to multiplex multiple clients onto a single public IP address. NAT can also occur on the server side with some load balancing techniques. However, this conjecture is unlikely in this case, because load balancing is typically deployed only for services so popular that a single server cannot handle the demand, and in such a case, we would expect to see high traffic from the monitor's vantage point.

A fourth possible reason is that perhaps the server was not actually in the VLAN that I

Table 6.4: remaining Nagios issues

#	server name	timestamp
1	gamma	Wed 2009-03-04 14:28:58
2	delta	Tue 2008-12-09 14:57:17
3	epsilon	Fri 2009-01-16 12:00:32
4	epsilon	Tue 2009-03-03 12:26:48
5	epsilon	Fri 2009-03-06 08:43:38
6	zeta	Tue 2009-01-20 08:34:13
7	eta	Tue 2009-02-17 22:46:24
8	theta	Tue 2009-02-17 23:11:51
9	iota	Tue 2009-02-17 23:26:51
10	iota	Tue 2009-02-17 23:26:51
11	kappa	Tue 2009-02-17 23:42:01
12	lambda	Tue 2009-02-17 23:53:51
13	lambda	Tue 2009-02-17 23:54:01

monitored.

Out of the 39 Nagios issues on 11 servers, there are 13 issues on 9 servers left to explain. The remaining issues are summarized in Table 6.4. I will address each of these issues in turn.

The first issue, for server `gamma`, has almost no data seen by `adudump`. Figure 6.9 shows the number of response times seen per hour around the issue. The hour of the incident saw a single response time of 0.824 milliseconds, and the hour after the incident saw two response times of 0.631 milliseconds and 0.540 milliseconds. Clearly, none of these is remotely close to a threshold of five seconds. Furthermore, the hour before the incident saw 71 response times, with an average of 35.587 milliseconds and zero response times above five seconds. Furthermore, `adudump` does not see nearly enough data for this server for my methods to work. See Figure 6.10. Some days have less than 100 response time measurements.

The second issue, for server `delta`, occurred less than an hour after I began measuring server VLAN traffic on December 9, 2008, so there was no past data to compare it against (see Figure 6.11). This means that, even if there was enough data for my methods, the methods would still be in the process of creating a normal basis and could not compute an anomaly score for a distribution of response times for this day. However, even looking only at the response time measurements from `adudump`, the average response time for this period is low (Figure 6.12), and there are zero response times over five seconds.

The third issue, for server `epsilon`, occurred on January 16, just after the end of an `adudump`

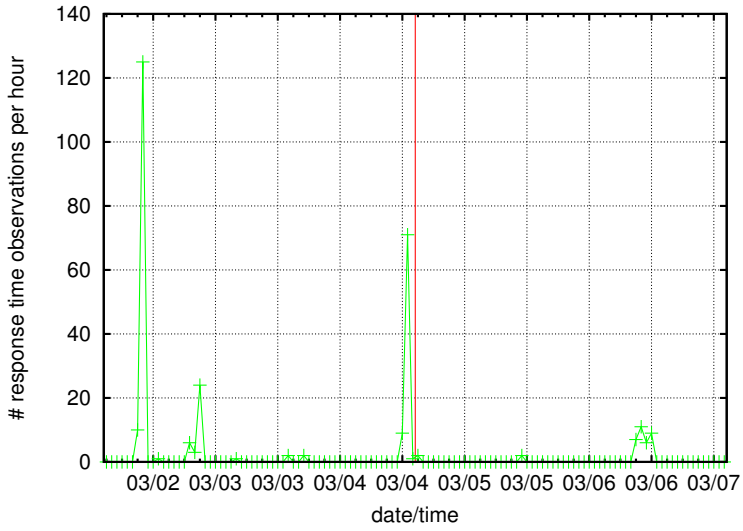


Figure 6.9: Count of response time measurements for `gamma` server around the time of its only issue.

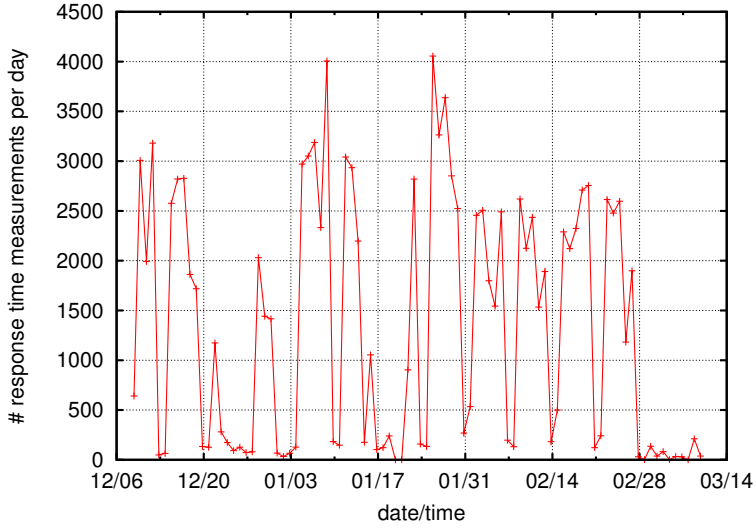


Figure 6.10: Count of response time measurements per day for `gamma` server.

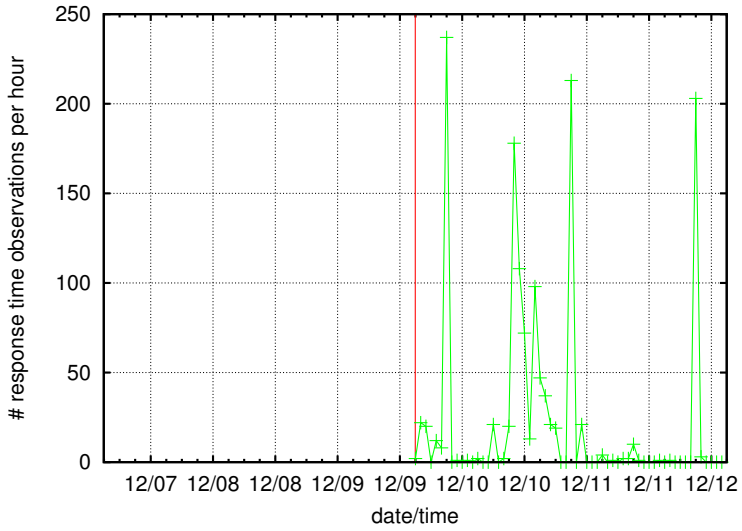


Figure 6.11: Count of response time measurements for `delta` server around the time of its only issue.

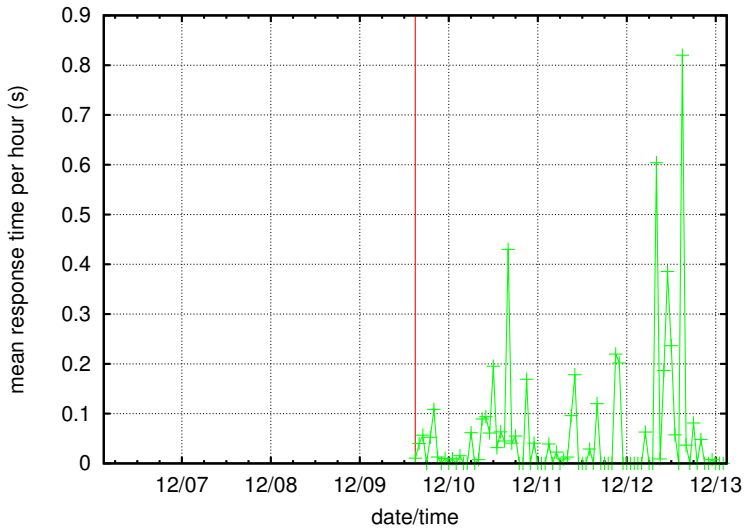


Figure 6.12: Average response time per hour for `delta` server around the time of its only issue.

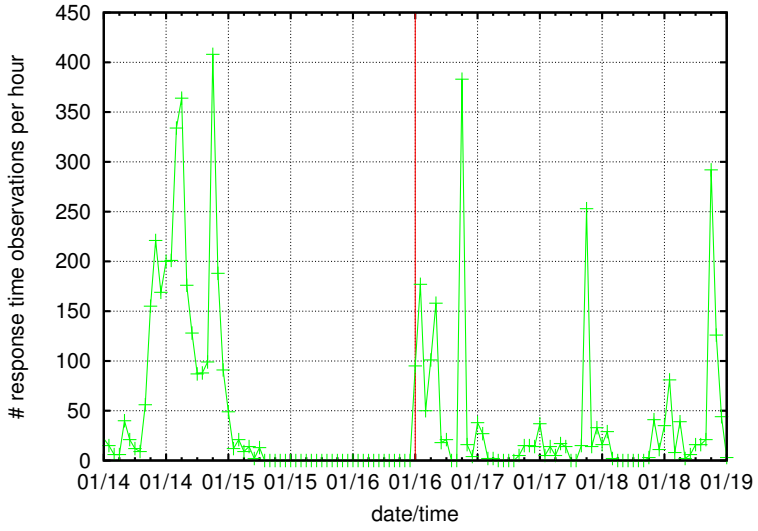


Figure 6.13: Count of response time measurements for `epsilon` server around the time of its first issue.

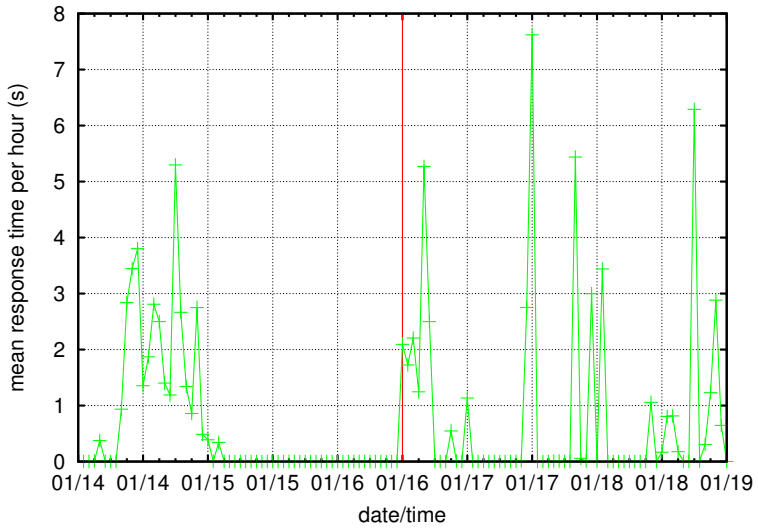


Figure 6.14: Average response time per hour for `epsilon` server around the time of its first issue.

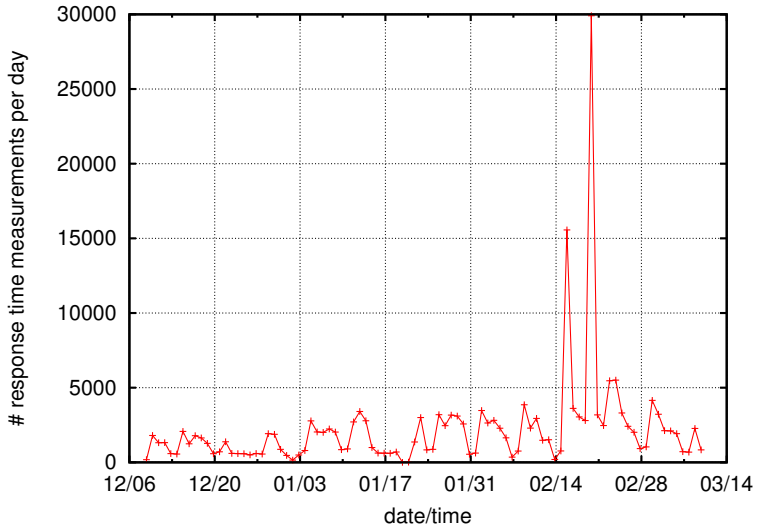


Figure 6.15: Count of response time measurements per day for `epsilon` server.

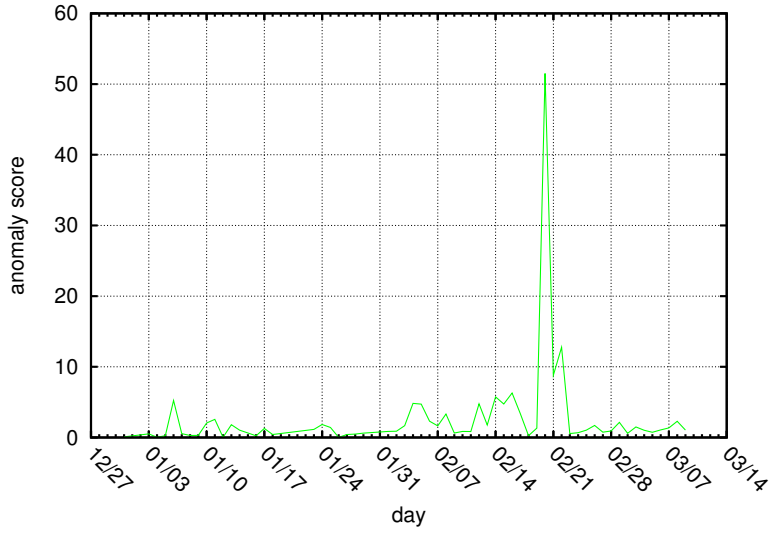


Figure 6.16: Daily anomaly score for the `epsilon` server.

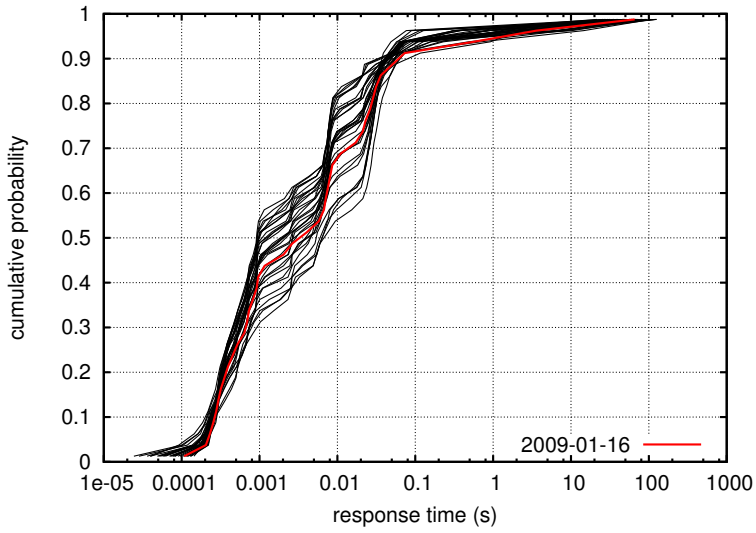


Figure 6.17: Response time distributions for the `epsilon` server, with the distribution containing its first issue highlighted.

measurement outage (see Figure 6.13). The average response time per hour is shown in Figure 6.14, and is not unusual for the hour of the issue. There are also zero observed response times greater than five seconds. Normally, my methods would ignore days that contain any part of an outage. Furthermore, the number of response time observations per daily distribution is often low (see Figure 6.15). However, I ran my methods on the available data anyway. Figure 6.16 shows the anomaly score. The score for January 16 is nowhere near the anomaly threshold of 1000, so the data for January 16 is not considered anomalous. Also, Figure 6.17 shows the distributions comprising this test. The black distributions form the normal basis for the test on January 16, whose distribution is highlighted in red. The highlighted distribution is well described by the envelope of previous distributions.

The fourth issue, which occurred on March 3, 2009, is also for server `epsilon`. The number of response time observations is relatively high (Figure 6.18) during the issue, but neither the average response time (Figure 6.19) nor the proportion of response times over the five second threshold (Figure 6.20) are unusually high. Furthermore, the anomaly score is low (Figure 6.16), and the response time distribution for March 3 is well described by the envelope of previous distributions (Figure 6.21).

The fifth issue, which occurred on March 6, 2009, is also for server `epsilon`. The number of response time observations for that hour is fairly typical (Figure 6.22). The average response

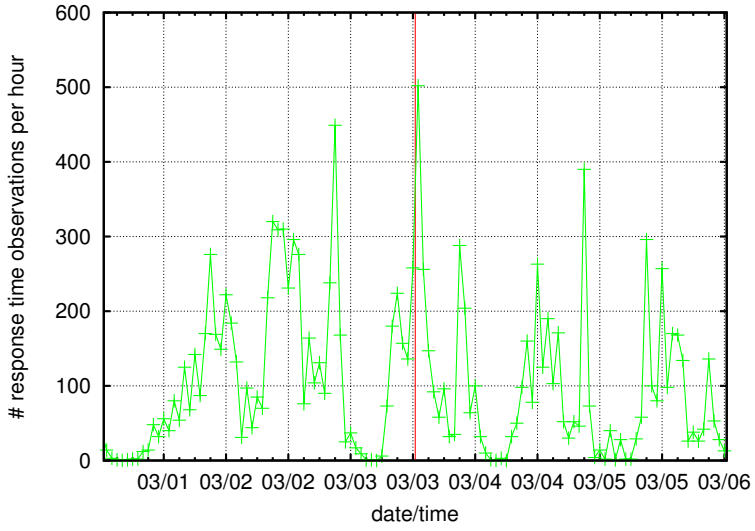


Figure 6.18: Count of response time measurements for `epsilon` server around the time of its second issue.

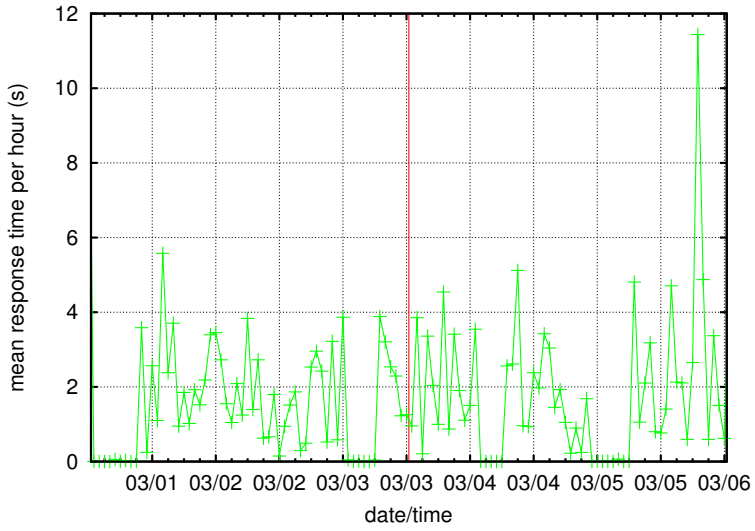


Figure 6.19: Average response time per hour for `epsilon` server around the time of its second issue.

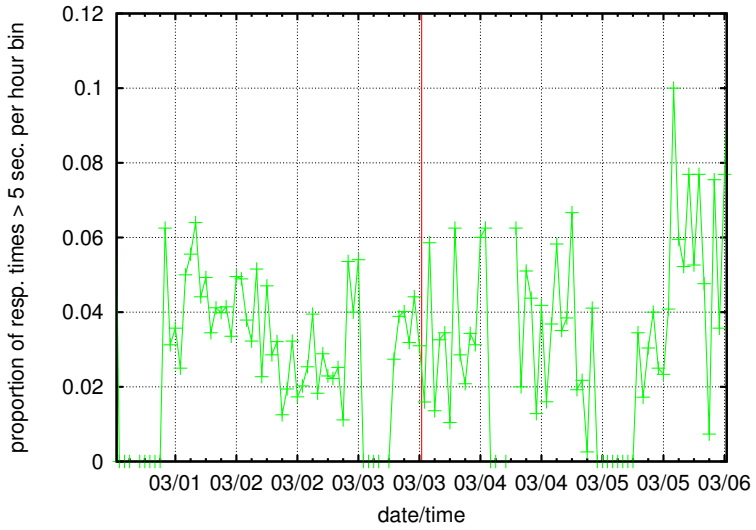


Figure 6.20: Proportion of observed response times over the five second threshold per hour bin for `epsilon` server around the time of its second issue.

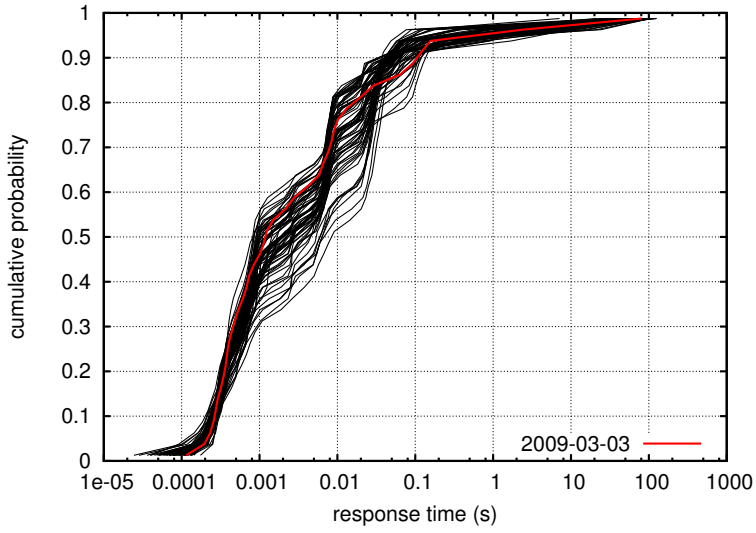


Figure 6.21: Response time distributions for the `epsilon` server, with the distribution containing its second issue highlighted.

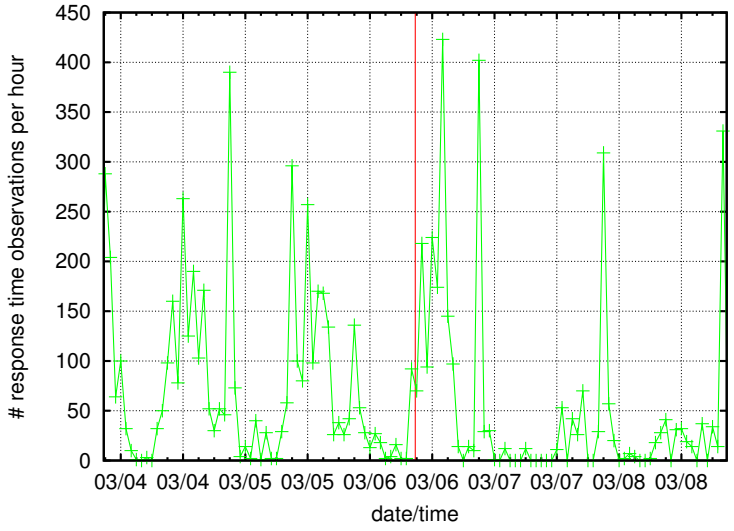


Figure 6.22: Count of response time measurements for `epsilon` server around the time of its third issue.

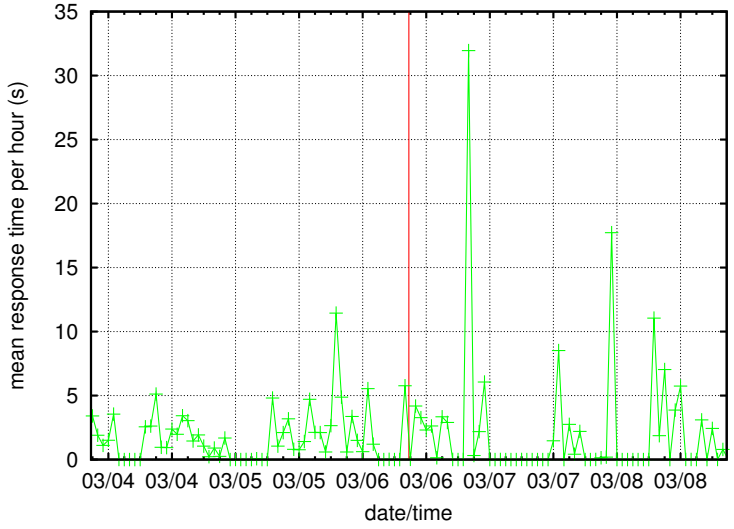


Figure 6.23: Average response time per hour for `epsilon` server around the time of its third issue.

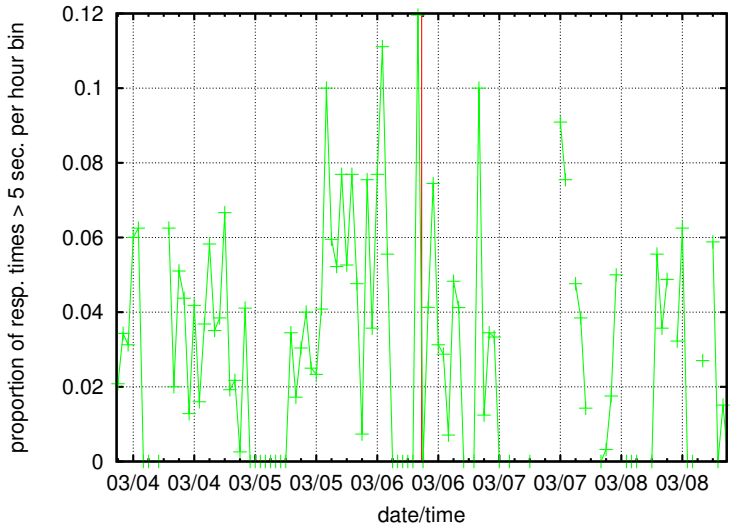


Figure 6.24: Proportion of observed response times over the five second threshold per hour bin for `epsilon` server around the time of its third issue.

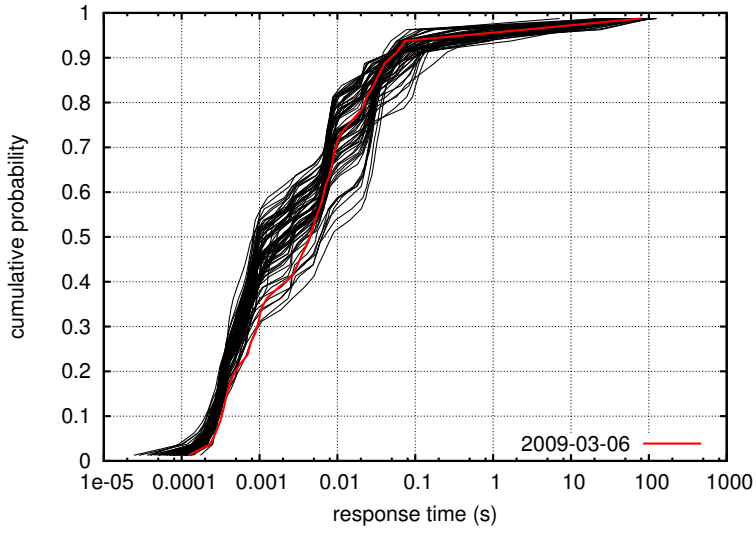


Figure 6.25: Response time distributions for the `epsilon` server, with the distribution containing its third issue highlighted.

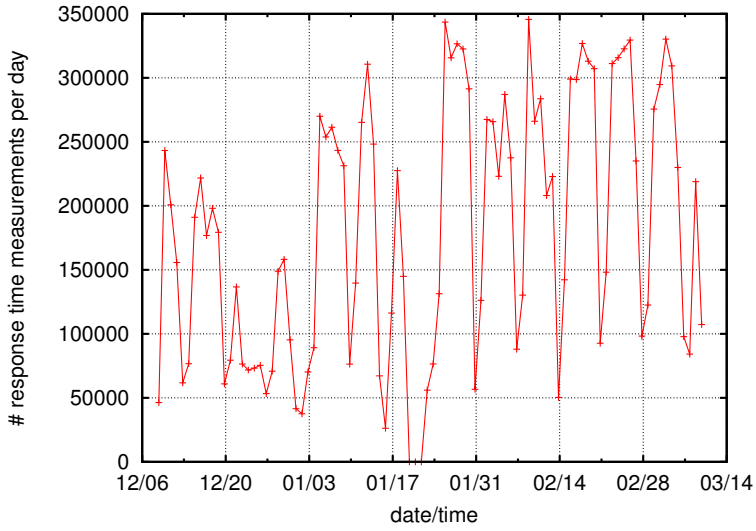


Figure 6.26: Count of response time measurements per day for `zeta` server.

time is relatively high (Figure 6.23), and the proportion of response times above five seconds is very high (Figure 6.24). It appears that the response times that `adudump` has observed support Nagios's assertion that there is a performance issue at this time. However, the anomaly score is still low (Figure 6.16), and the distribution of response times for March 6 is not visually anomalous (Figure 6.25). Notice that response times greater than five seconds are not unusual for this server. Digging deeper in the Nagios logfiles, I found that it is not uncommon for Nagios to find a "soft" issue with this server (*i.e.* a single response time larger than five seconds), but a "hard" issue (*i.e.* three successive response times larger than five seconds) is rare. There were 38 soft issues and only three hard issues. Thus, I argue that, in this case, my methods are more tolerant to the natural performance of the server. Relatively long response times are not unusual for this server, and they do not result in an anomaly.

The sixth issue, for server `zeta`, occurred on January 20, 2009. The server had plenty of data for my methods to work without concern for sampling error (Figure 6.26). However, as shown in Figure 6.27, there was unfortunately a measurement outage during the time of the issue, so I cannot analyze the issue further.

The seventh issue, for server `eta`, occurred on February 17, 2009. This server also was one of the ones passing the frequency thresholds, so it was analyzed for anomalies. No anomalies were found, however. The number of responses per hour was low during the issue, but it was

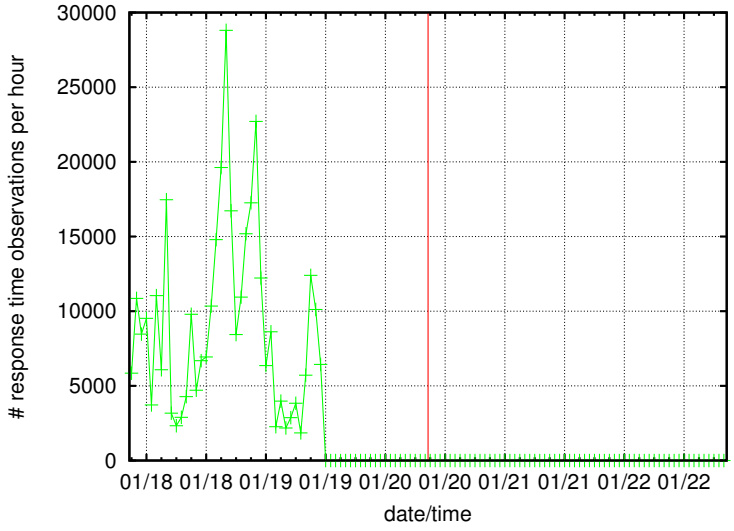


Figure 6.27: Count of response time measurements for `zeta` server around the time of its only issue.

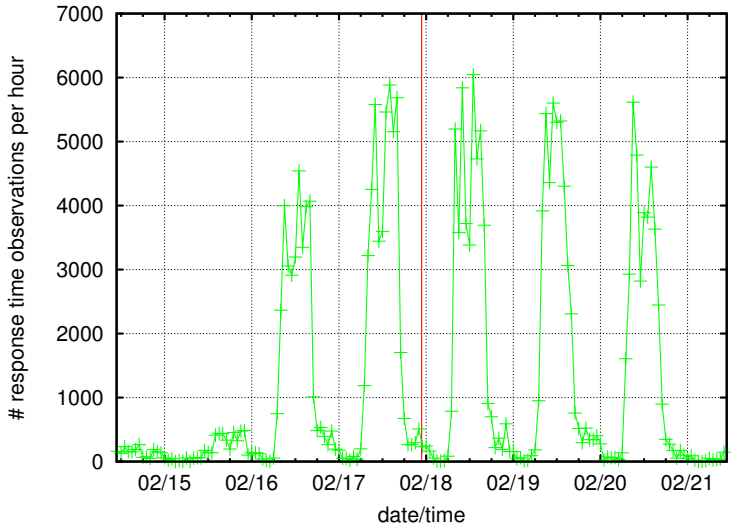


Figure 6.28: Count of response time measurements for `eta` server around the time of its only issue.

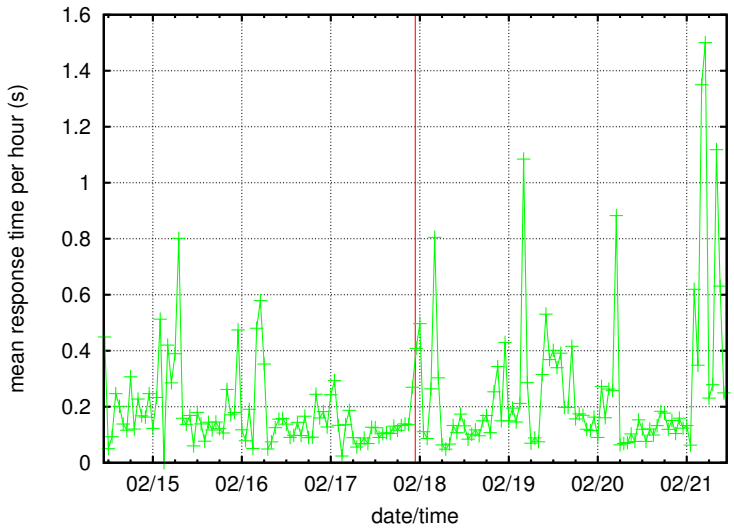


Figure 6.29: Average response time per hour for `eta` server around the time of its only issue.

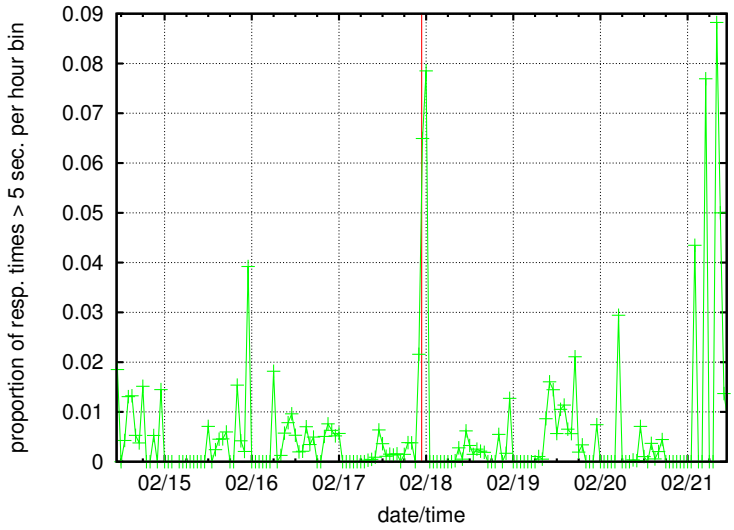


Figure 6.30: Proportion of observed response times over the five second threshold per hour bin for `eta` server around the time of its only issue.

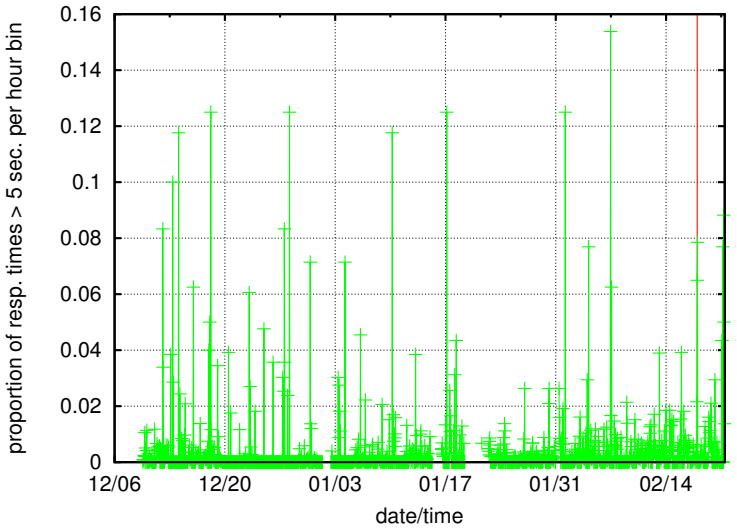


Figure 6.31: Proportion of observed response times over the five second threshold per hour bin for `eta` server.

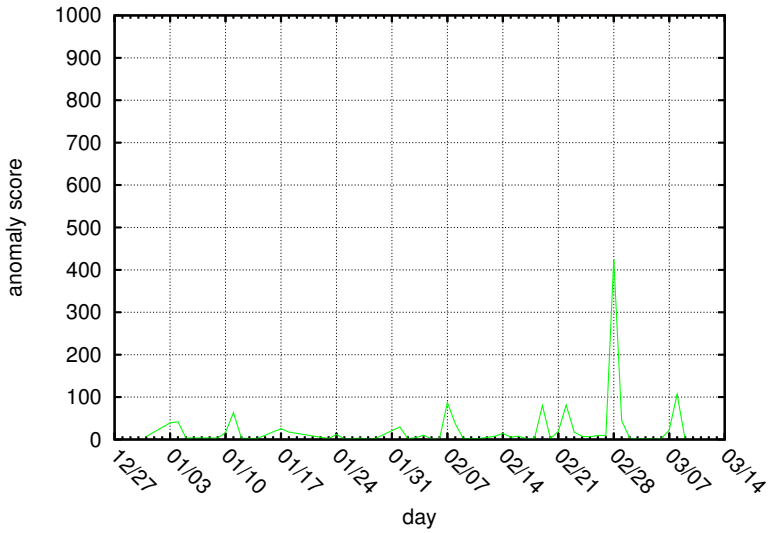


Figure 6.32: Daily anomaly score for the `eta` server.

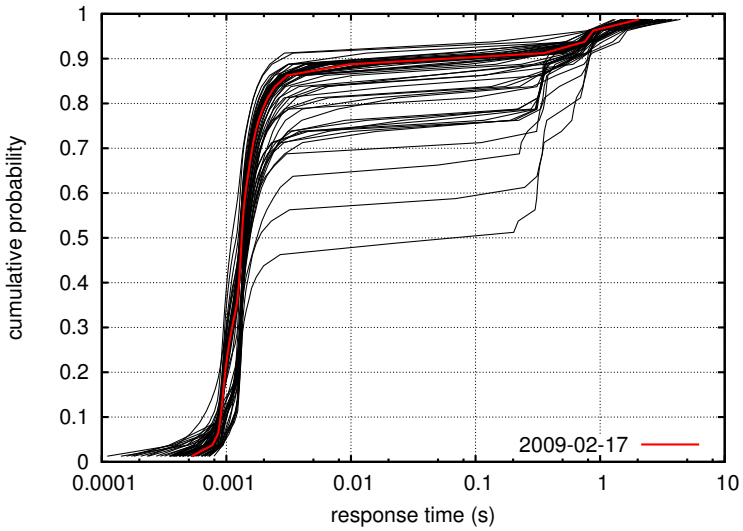


Figure 6.33: Response time distributions for the `eta` server, with the distribution containing its only issue highlighted.

typical for the time of day (Figure 6.28). The average response time per hour was somewhat high, but not significantly so (Figure 6.29). The proportion of response times greater than five seconds was the highest seen for several days (Figure 6.30), but it was hardly the highest seen ever (Figure 6.31). Furthermore, the anomaly score was well under the threshold (Figure 6.32), and the distribution was well described by the normal basis (Figure 6.33). I note that several distributions in the normal basis appear to be anomalous here. The reason for this is that all of these were in the training set, and the spread among the training set was high enough (even when excluding the apparently anomalous distributions) to cause a relatively low anomaly score. I have noted previously that improving the cross-validation of the training set is an issue for future work. I conclude that Nagios was simply unlucky in the measurements that it took. The probability of a response time above the threshold at that point was (based on `adudump`'s measurements) around 0.06, and although three successive events is improbable, it can nonetheless happen.

The eighth issue, for server `theta`, also occurred on February 17, 2009. I do not know why Nagios flagged this server as having a performance issue. It had a typical number of responses (Figure 6.34) and a typical average response time (Figure 6.35). The proportion of response times over five seconds was zero during the hour of the issue (Figure 6.36). Furthermore, the anomaly score was far below the threshold (Figure 6.37), and the distribution of response times

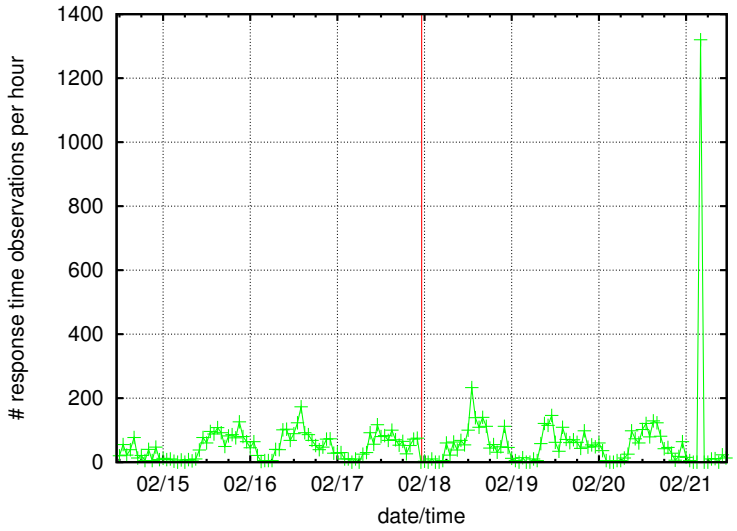


Figure 6.34: Count of response time measurements for `theta` server around the time of its only issue.

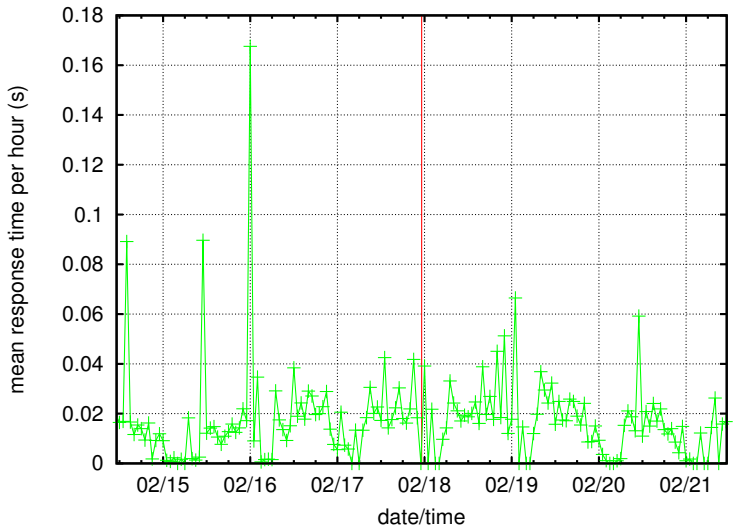


Figure 6.35: Average response time per hour for `theta` server around the time of its only issue.

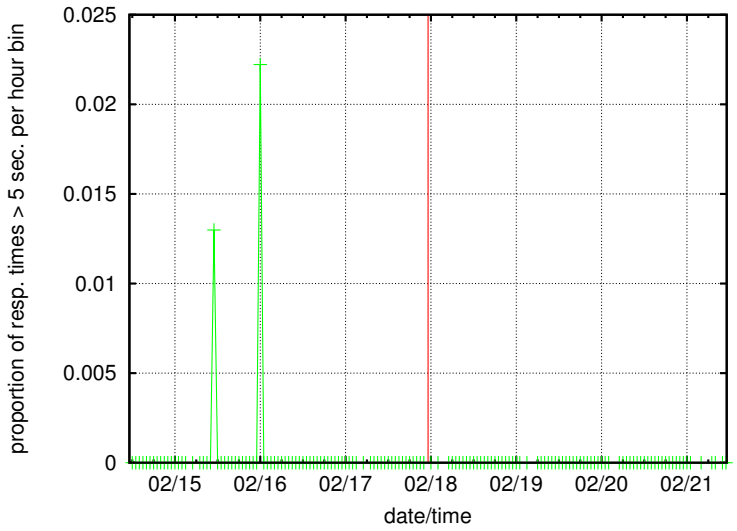


Figure 6.36: Proportion of observed response times over the five second threshold per hour bin for `theta` server around the time of its only issue.

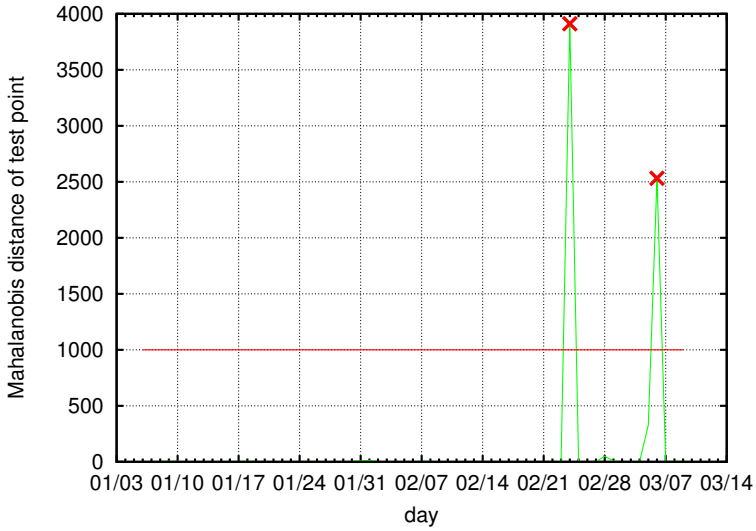


Figure 6.37: Daily anomaly score for the `theta` server.

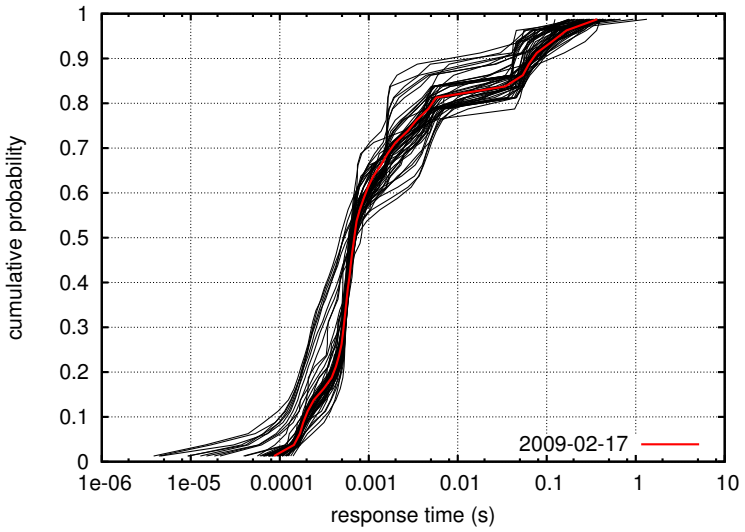


Figure 6.38: Response time distributions for the `theta` server, with the distribution containing its only issue highlighted.

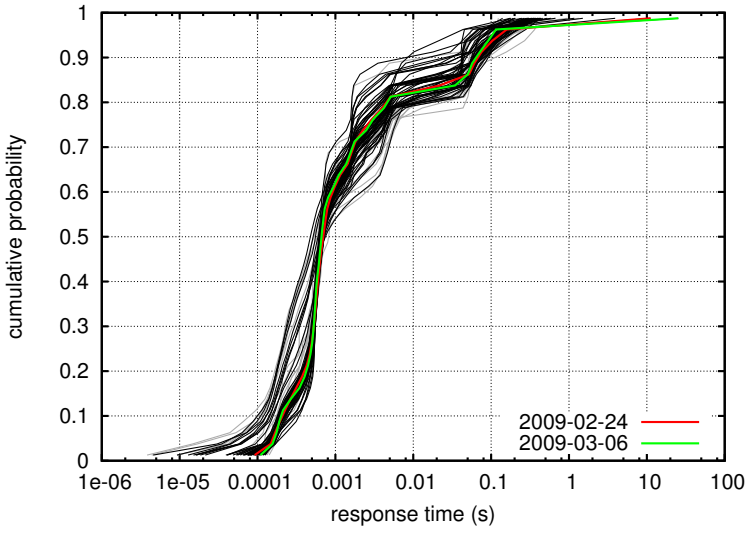


Figure 6.39: Response time distributions for the `theta` server, with the distribution containing its only issue highlighted.

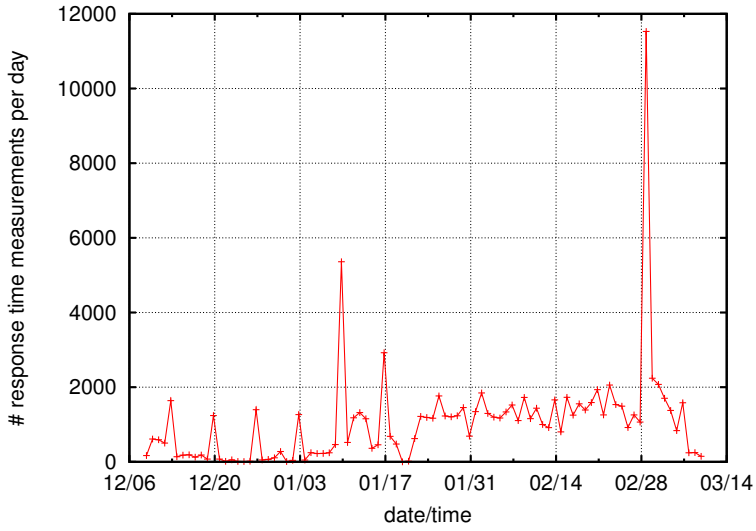


Figure 6.40: Count of response time measurements per day for `iota` server.

for February 17 was typical (Figure 6.38). I also note that the two anomalies indicated in Figure 6.37 are anomalous because there were more response times of ten seconds or more than for any other day, so it seems odd, based on the data available to `adudump`, that Nagios would decide that February 17 was anomalous but not February 24 or March 6.

Issues 9 and 10 are both for server `iota`, and both issues actually have the same timestamp (although the response times as reported in the Nagios log file are different: 6.261 seconds and 6.276 seconds, respectively). Therefore, I will treat them as a single anomaly. The server has very few observed response times (Figure 6.40). Furthermore, there were only two response times for the hour containing the issues (see Figure 6.41). Either the server is only used infrequently, or `adudump` running on the border link is not privy to the traffic for this server.

The eleventh issue, for server `kappa`, also occurred on February 17, 2009. I do not know why Nagios flagged this server as having a performance issue. It had a typical number of responses (Figure 6.42) and a typical average response time (Figure 6.43). The proportion of response times over five seconds was zero during the hour of the issue (Figure 6.44). Furthermore, the anomaly score was far below the threshold (Figure 6.45), and the distribution of response times for February 17 was typical (Figure 6.46).

Issues 12 and 13 were for server `lambda`. The issues were only ten seconds apart, so I will analyze them together. There were generally not very many response times (Figure 6.47), and

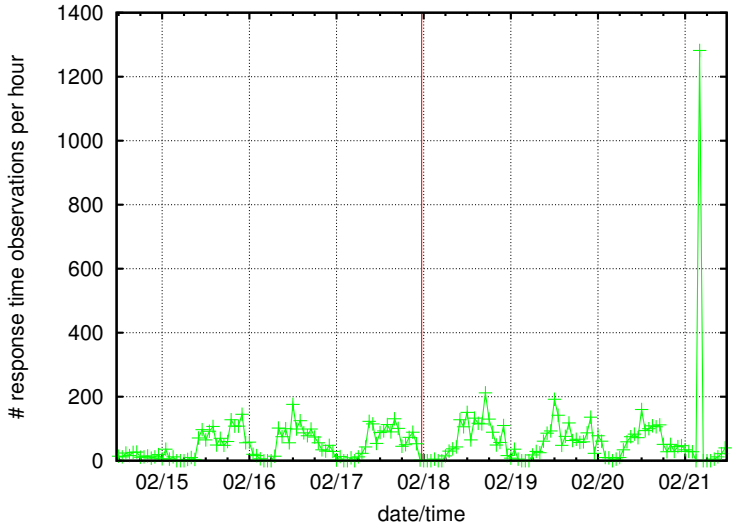


Figure 6.41: Count of response time measurements for `iota` server around the time of its two (simultaneous) issues.

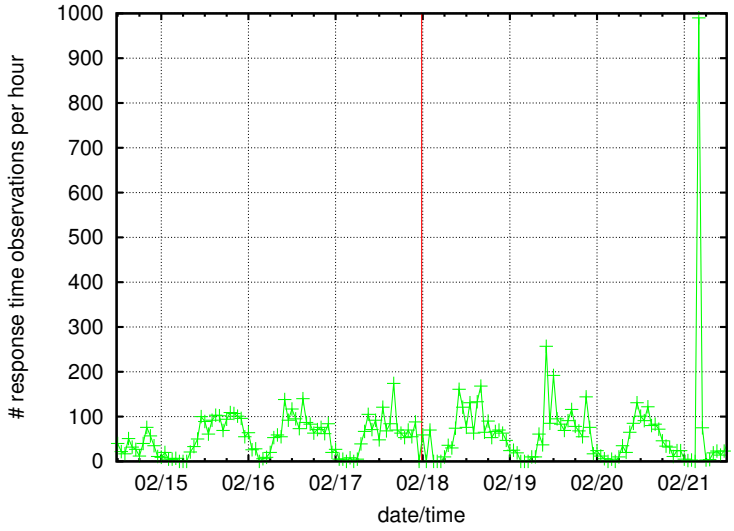


Figure 6.42: Count of response time measurements for `kappa` server around the time of its only issue.

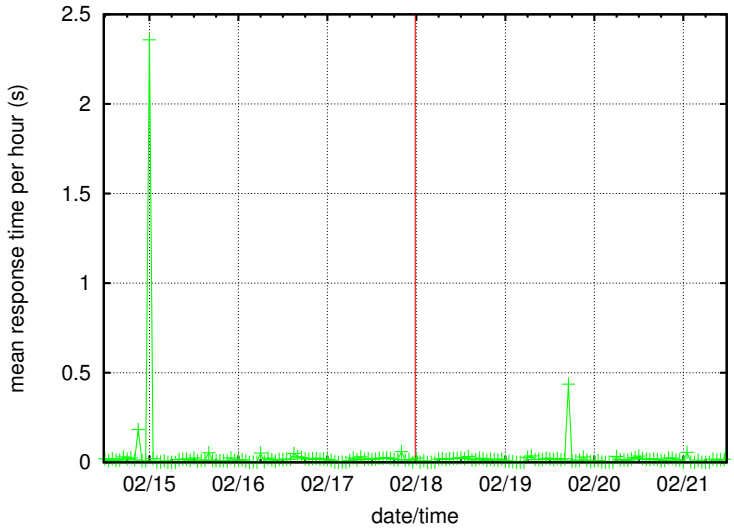


Figure 6.43: Average response time per hour for `kappa` server around the time of its only issue.

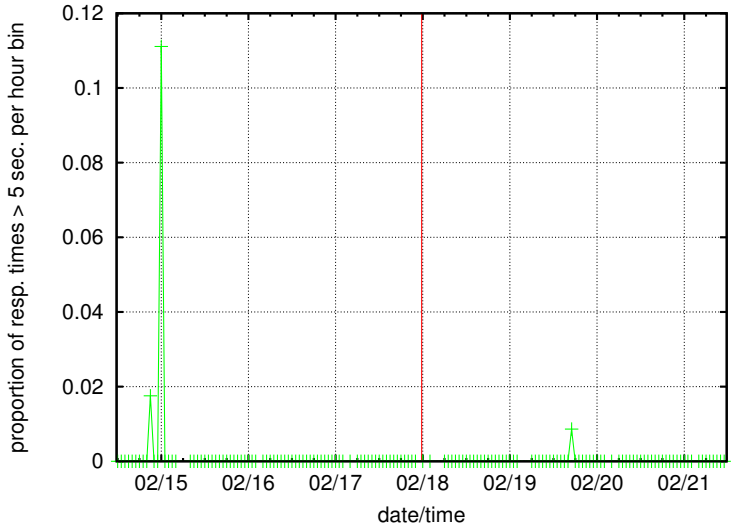


Figure 6.44: Proportion of observed response times over the five second threshold per hour bin for `kappa` server around the time of its only issue.

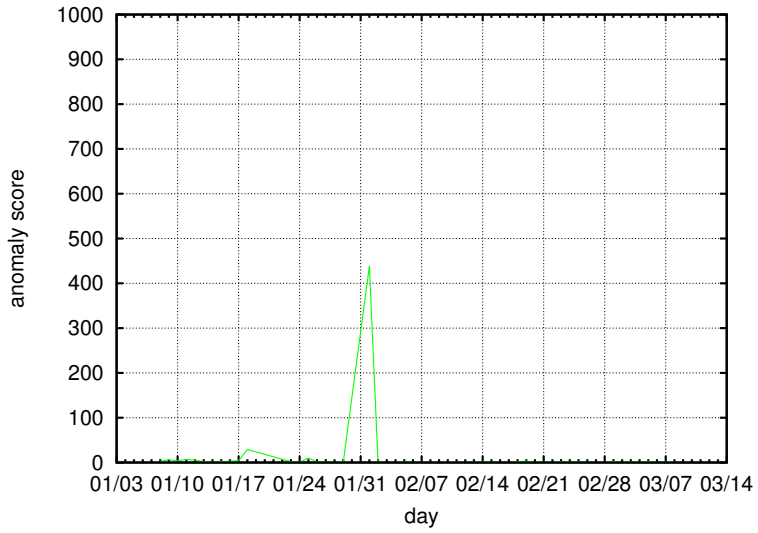


Figure 6.45: Daily anomaly score for the `kappa` server.

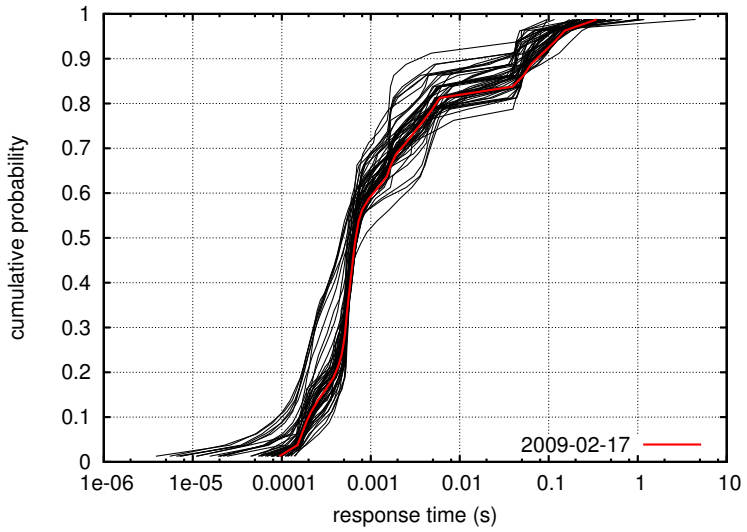


Figure 6.46: Response time distributions for the `kappa` server, with the distribution containing its only issue highlighted.

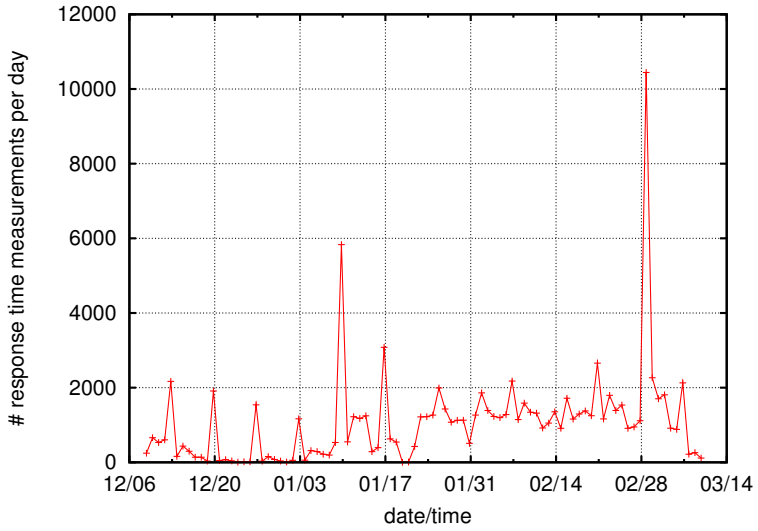


Figure 6.47: Count of response time measurements per day for `lambda` server.

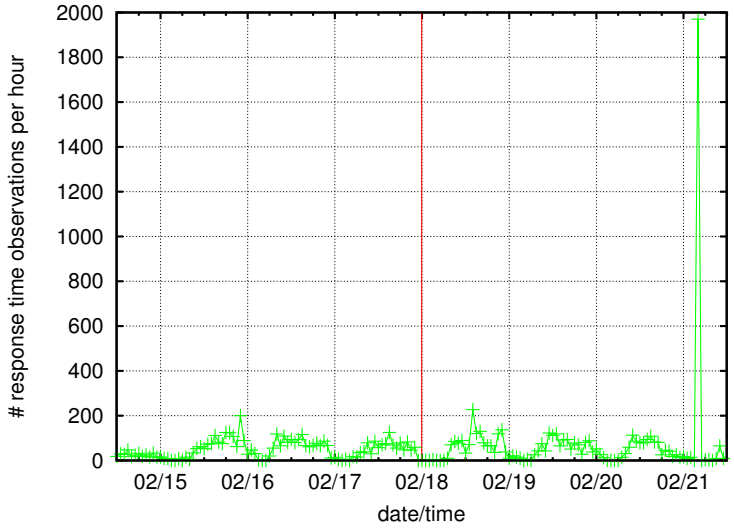


Figure 6.48: Count of response time measurements for `lambda` server around the time of its two (nearly simultaneous) issues.

Table 6.5: unmatched anomalies identified by my methods

#	server name	date
1	zeta	Sat 2009-02-28
2	zeta	Sun 2009-03-01
3	mu	Sat 2009-01-03
4	mu	Sat 2009-01-10
5	mu	Sat 2009-01-17
6	mu	Sat 2009-02-07
7	mu	Sat 2009-02-14
8	mu	Sat 2009-02-28
9	nu	Thu 2009-02-26
10	nu	Sat 2009-02-28
11	omicron	Sat 2009-02-07
12	pi	Thu 2009-02-26
13	pi	Sun 2009-03-01
14	rho	Thu 2009-02-26
15	rho	Sat 2009-02-28
16	sigma	Fri 2009-01-09
17	sigma	Sun 2009-01-25
18	tau	Sat 2009-01-03
19	tau	Sat 2009-01-10
20	tau	Sat 2009-02-07
21	tau	Sat 2009-02-28

`adudump` observed no response times during the hour of the issue (see Figure 6.48), so I conclude that this is another issue with the vantage point not seeing all the traffic.

Analyzing unmatched anomalies

There were also 21 anomalies (on 8 server address/port pairs) identified by my methods, and none of them corresponded with a Nagios performance alert. Table 6.5 lists the anomalies.

The first two anomalies corresponded with server `zeta` identified above. As shown in Figure 6.49, there were two days which had anomaly scores well above the anomaly threshold. The question is why my methods identified an anomaly when Nagios did not. The answer is found in Figure 6.50, which shows that the anomalous distributions had a shape clearly different than the normal basis, yet which converge back with the normal basis by the Nagios threshold. In other words, the days in question are indeed anomalous for much of their range, but the range that Nagios cares about (response times greater than five seconds) were typical. Nagios simply is not concerned about performance issues that do not result in response times above a certain

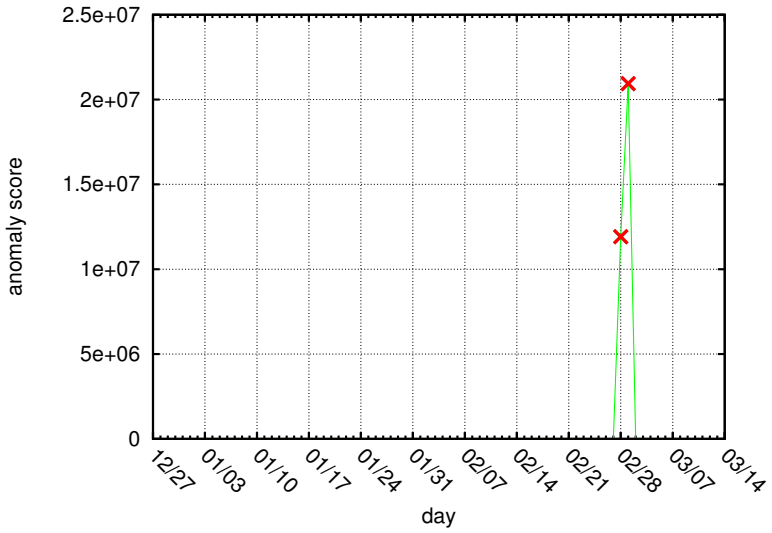


Figure 6.49: Daily anomaly score for the `zeta` server.

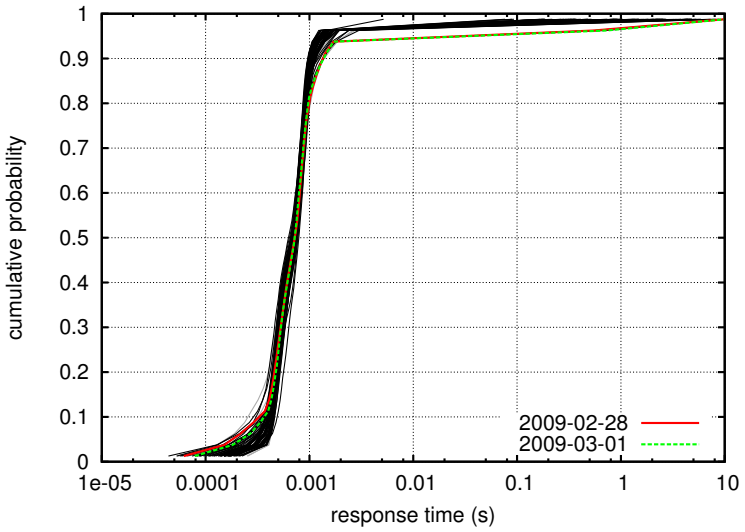


Figure 6.50: Response time distributions for the `zeta` server, with the two anomalous distributions highlighted.

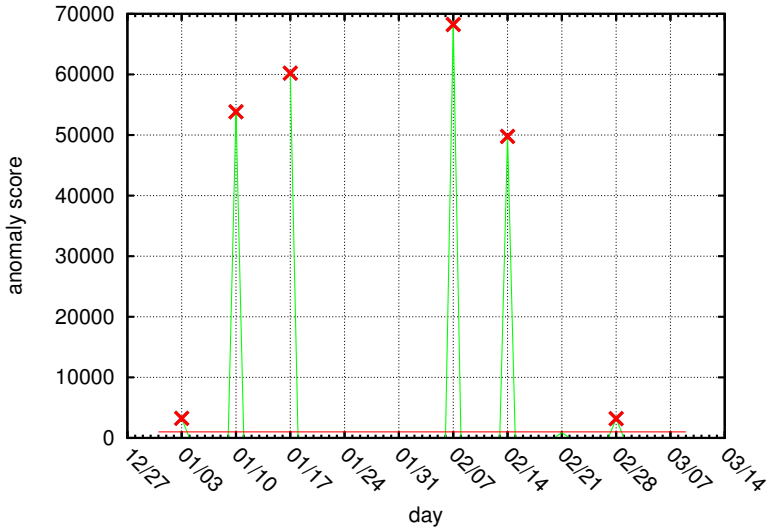


Figure 6.51: Daily anomaly score for the `mu` server.

threshold.

Anomalies 3 through 8 were for server `mu`, and the anomaly scores are shown in Figure 6.51. Figure 6.52 shows the first anomalous distribution (from January 3, 2009) highlighted along with the distributions comprising its normal basis. The tail of the distribution is heavier, with more response times greater than five seconds. The other five anomalies have the same cause, and their distributions are shown in Figures 6.53 through 6.57. Figure 6.58 shows the proportion of observed response greater than five seconds, and the anomalous days have the highest values. (December 13 and 20 were found to be anomalous versus the other days in the training set, and were excluded from the initial normal basis.) However, recall that these anomalies have no trace in the Nagios logs. This is probably because, although the proportion of response times greater than five seconds increases, the Nagios threshold is set to something higher, such as twenty seconds. The proportion of observed response times greater than twenty seconds is about the same for the anomalous days, as shown in Figure 6.59.

Anomalies 9 and 10 were for server `nu`, and the anomaly scores are plotted in Figure 6.60. The first anomaly, which occurred on February 26, 2009, is highlighted in Figure 6.61, and it is significantly different from its basis. The second anomaly, from February 28, 2009, is highlighted in Figure 6.62. It is also different from its basis, but it is not as significant a difference as the first anomaly. Its anomaly score is 1348, barely above the threshold of 1000. Figure 6.63 shows

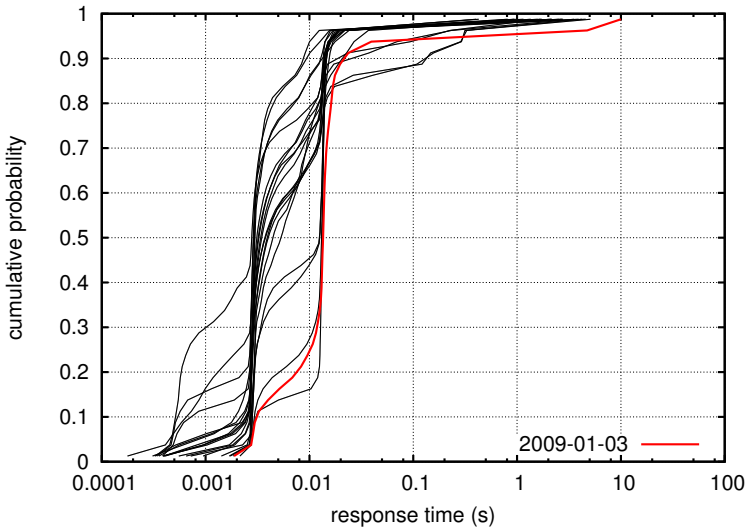


Figure 6.52: Response time distributions for the `mu` server, with the first anomalous distribution highlighted.

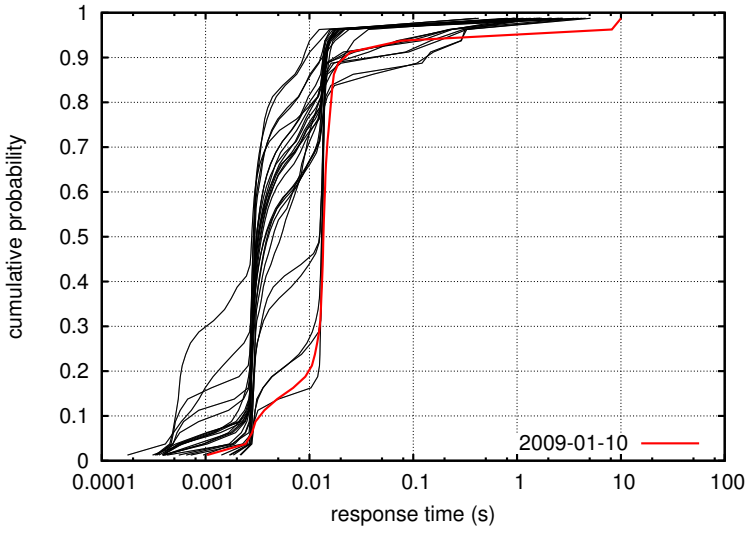


Figure 6.53: Response time distributions for the `mu` server, with the second anomalous distribution highlighted.

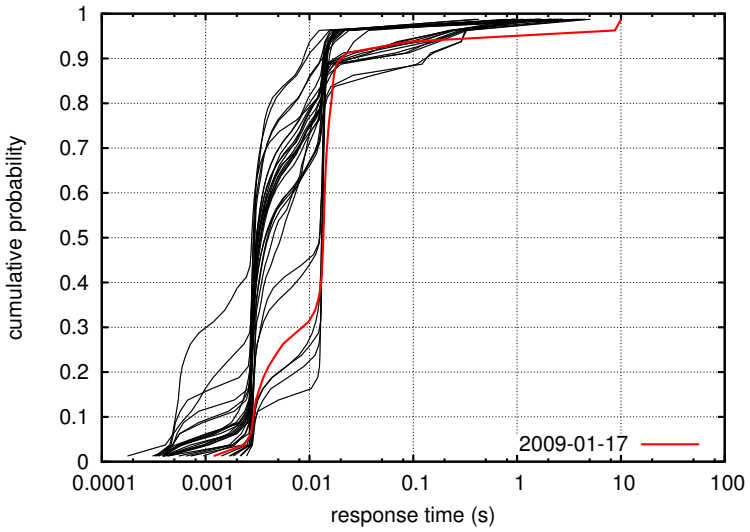


Figure 6.54: Response time distributions for the `mu` server, with the third anomalous distribution highlighted.

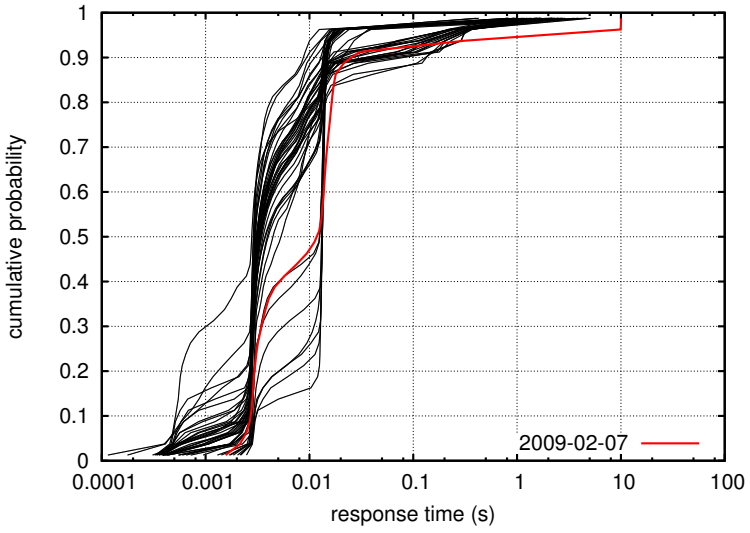


Figure 6.55: Response time distributions for the `mu` server, with the fourth anomalous distribution highlighted.

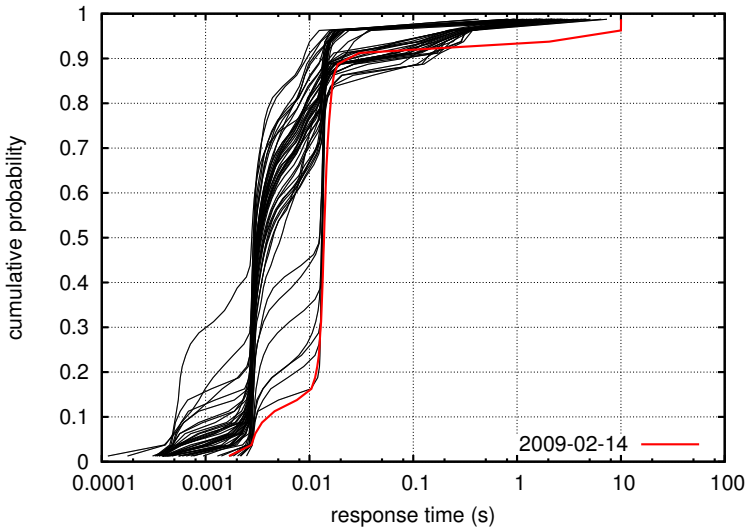


Figure 6.56: Response time distributions for the `mu` server, with the fifth anomalous distribution highlighted.

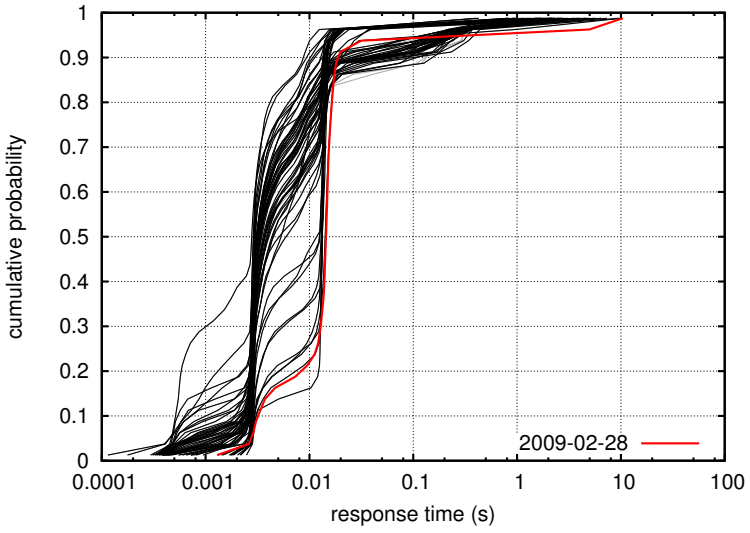


Figure 6.57: Response time distributions for the `mu` server, with the sixth anomalous distribution highlighted.

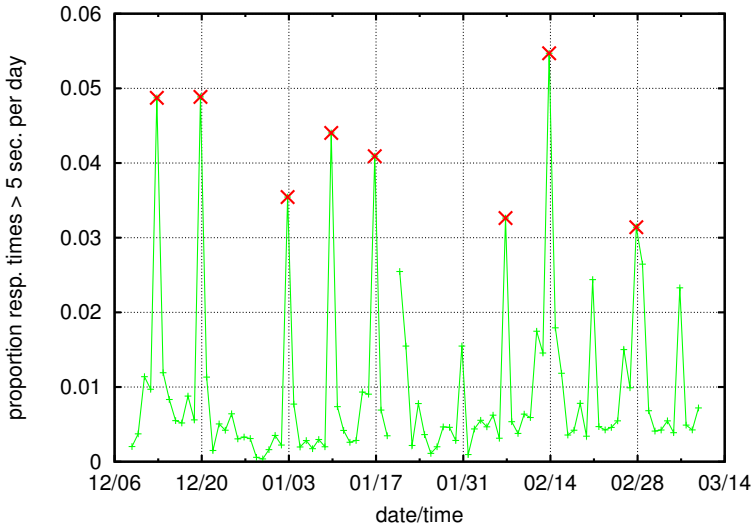


Figure 6.58: Proportion of observed response times over the five second threshold per day for `mu` server, with anomalous days highlighted.

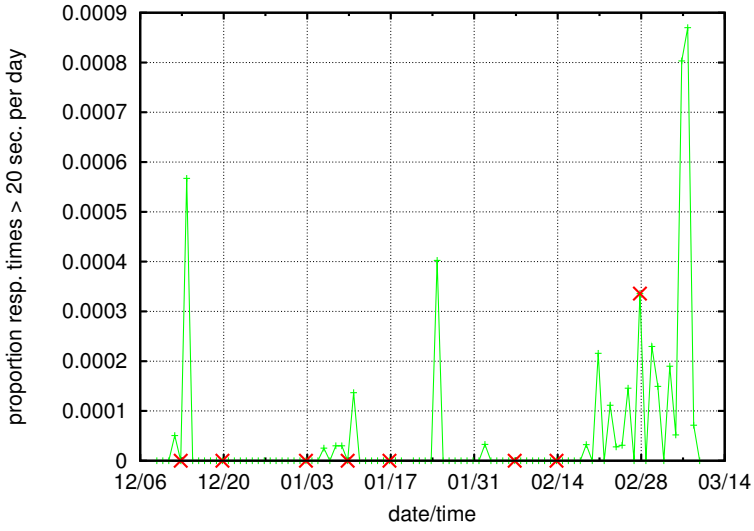


Figure 6.59: Proportion of observed response times over a twenty second threshold per day for `mu` server, with anomalous days highlighted.

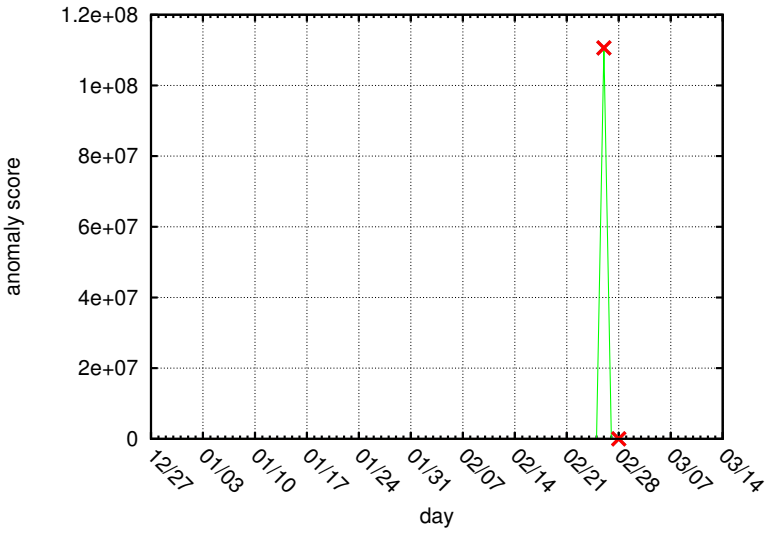


Figure 6.60: Daily anomaly score for the `nu` server.

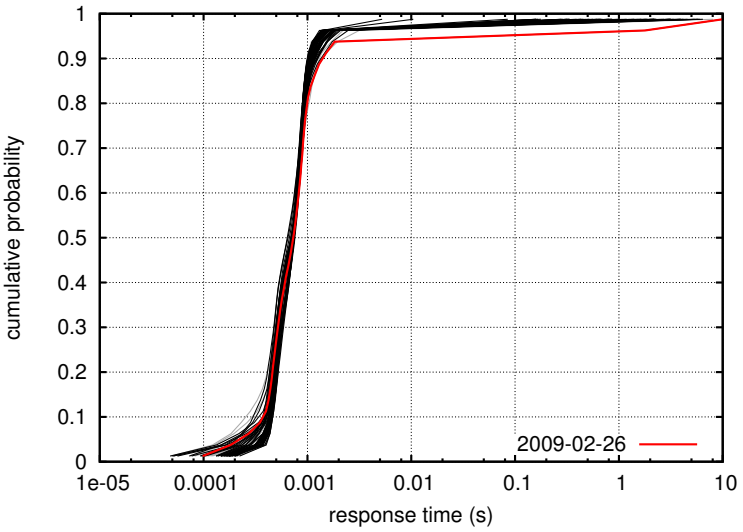


Figure 6.61: Response time distributions for the `nu` server, with the first anomalous distribution highlighted.

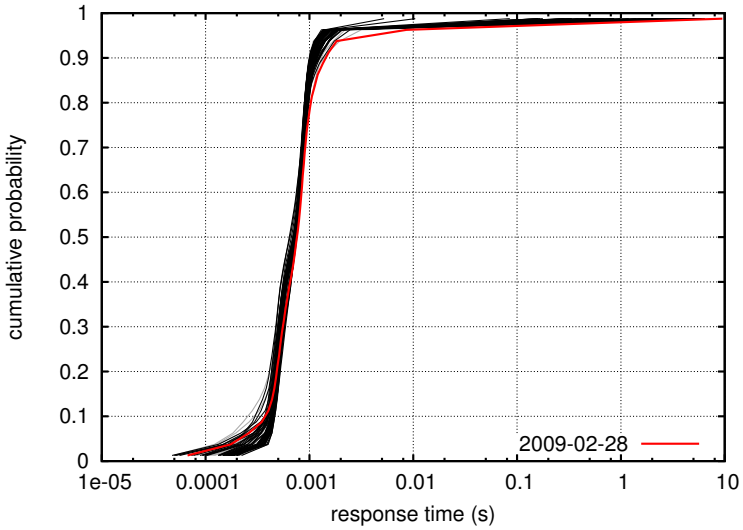


Figure 6.62: Response time distributions for the `nu` server, with the second anomalous distribution highlighted.

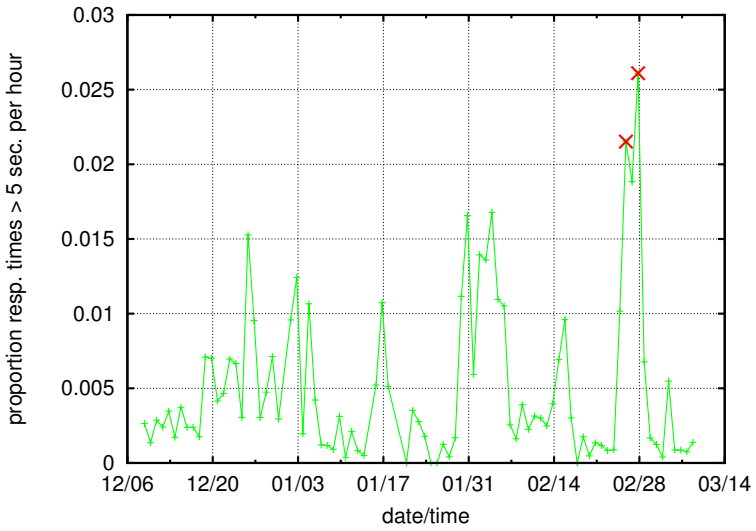


Figure 6.63: Proportion of observed response times over the five second threshold per day for `nu` server.

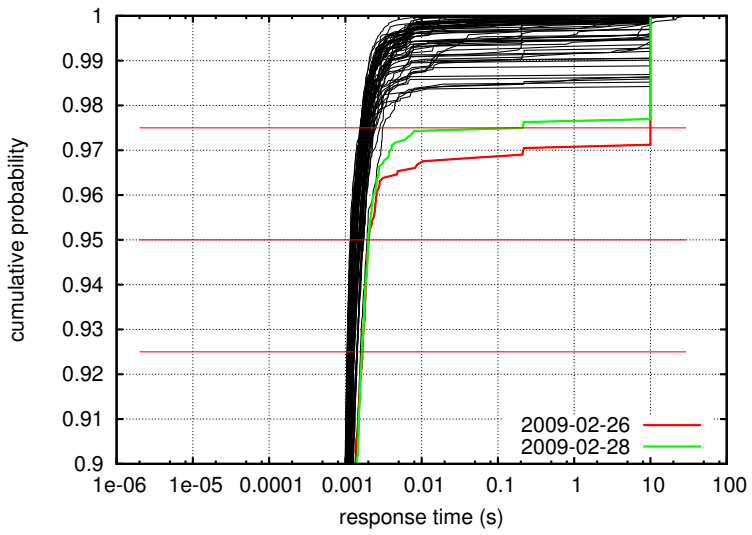


Figure 6.64: Cumulative distribution function (CDF), zoomed, for `nu` server, with anomalous distributions highlighted. Bin boundaries for a 40-bin QF are plotted in red.

the proportion of observed response times greater than five seconds. The two days identified by my methods correspond with the maximum values.

One puzzling feature of the first issue is that the anomaly score does not seem to be commensurate with the increase in the proportion of five-plus-second response times. In other words, the proportion is only about twenty percent greater on February 26 than on February 28, but the anomaly score is nearly 100,000 times greater. To understand why, consider Figure 6.64. So far, I have been plotting the discrete (or summarized) quantile function (QF) with 40 bins, with the X and Y axes swapped to imitate the style of a CDF. However, this figure is the actual, unsummarized CDF. Recall from Section 4.2 that each bin of the 40-bin QF is an average of the values in a distribution between two quantiles. For example, the last bin is the average of all values between the .975 quantile and the maximum. From the figure, the variation of the last bin among the distributions forming the normal basis (black lines) is fairly large, with several values of at least five seconds, so the values of the last bins of the two anomalous distributions are not extreme outliers. However, the normal distributions all have small values (less than 0.0032) for their 39^{th} bins, and the variation is small, so the 39^{th} bins of the anomalous QF distributions (0.0084 and 1.787, respectively) are quite anomalous. This is easier to see in Figure 6.65, which is plotted without a logarithmically-scaled X-axis. Generally speaking, I claim that the adaptability to different variations is a strength of my method, because some

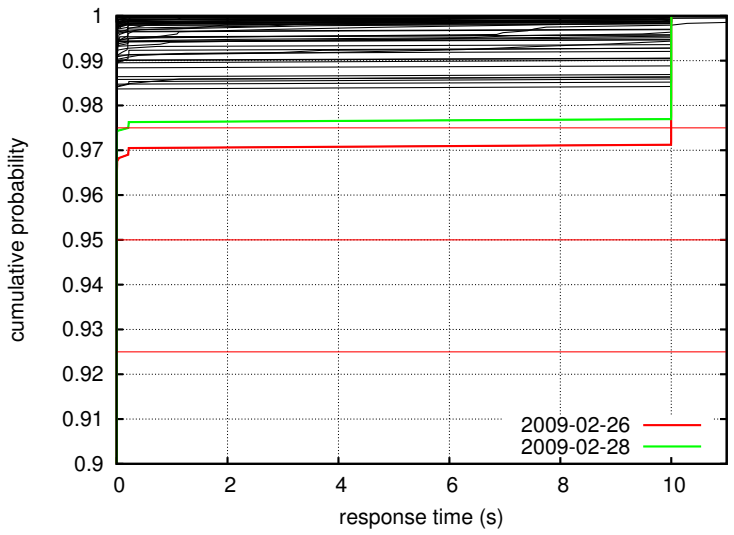


Figure 6.65: Cumulative distribution function (CDF), zoomed, on a log-scaled X-axis, for `nu` server, with anomalous distributions highlighted. Bin boundaries for a 40-bin QF are plotted in red.

servers typically have higher variation of performance even under normal operation. However, this appears to be a situation in which such adaptability is a weakness.

The eleventh anomaly was for server `omicron`, and the anomaly scores are shown in Figure 6.66. The anomaly occurred on February 7, 2009, and the distribution of response times for this day was noticeably different (Figure 6.67). The proportion of response times greater than five seconds is shown in Figure 6.68, and the value for February 7 is near the maximum. The actual maximum is from an incomplete day (January 21) during which `adudump` only observed 169 response times because of an outage, and the anomaly score was not computed for that day.

Anomalies 12 and 13 were for server `pi`, and the anomaly scores are shown in Figure 6.69. These anomalies exhibits the same characteristics as the anomaly for server `zeta`: although the body of the anomalous distributions diverge from the normal basis, they converge back to the normal basis by the point at which response times are high enough for Nagios to notice (Figure 6.70). This server also exhibits the effect seen for server `nu`, in which the anomaly score is inflated because of low variance in the 39^{th} bin.

Server `rho` has the same explanation for anomalies 14 and 15, as shown in Figures 6.71 and 6.72.

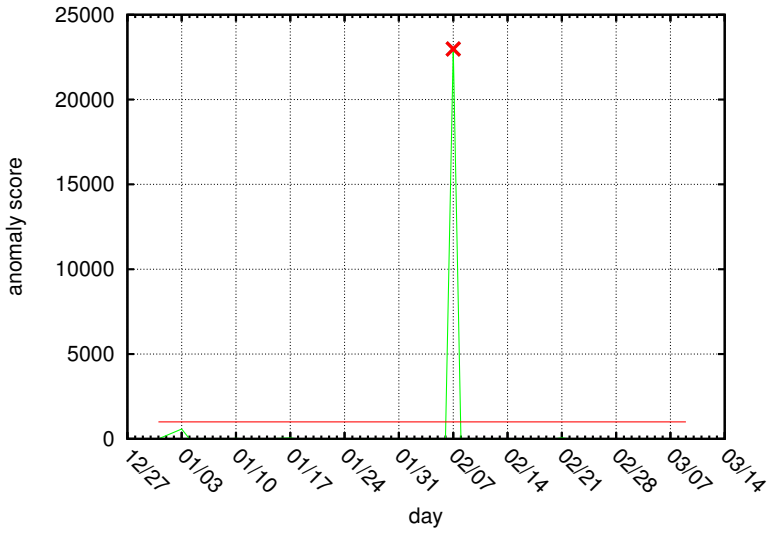


Figure 6.66: Daily anomaly score for the `omicron` server.

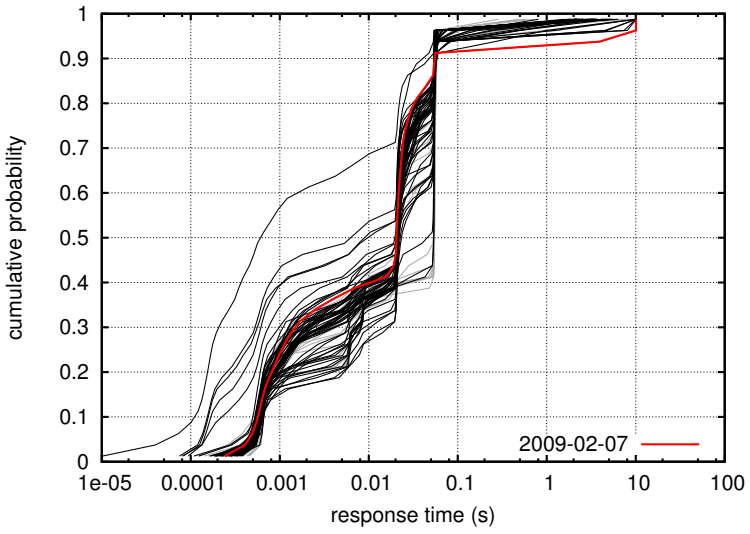


Figure 6.67: Response time distributions for the `omicron` server, with the anomalous distribution highlighted.

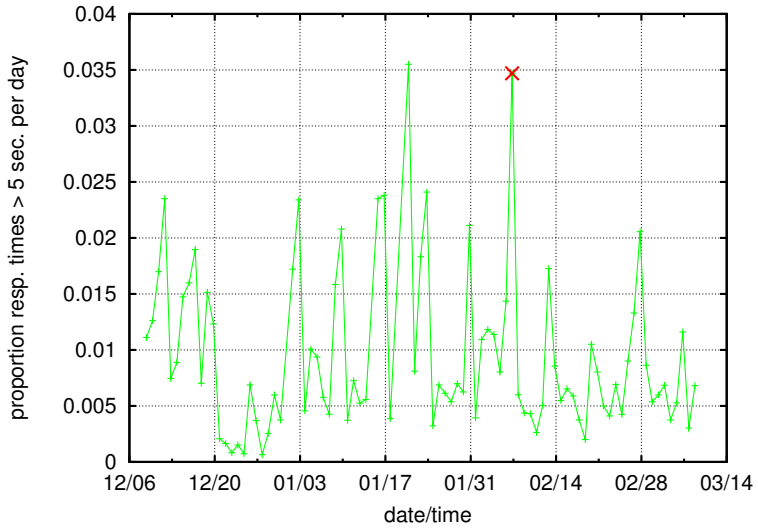


Figure 6.68: Proportion of observed response times over the five second threshold per day for `omicron` server.

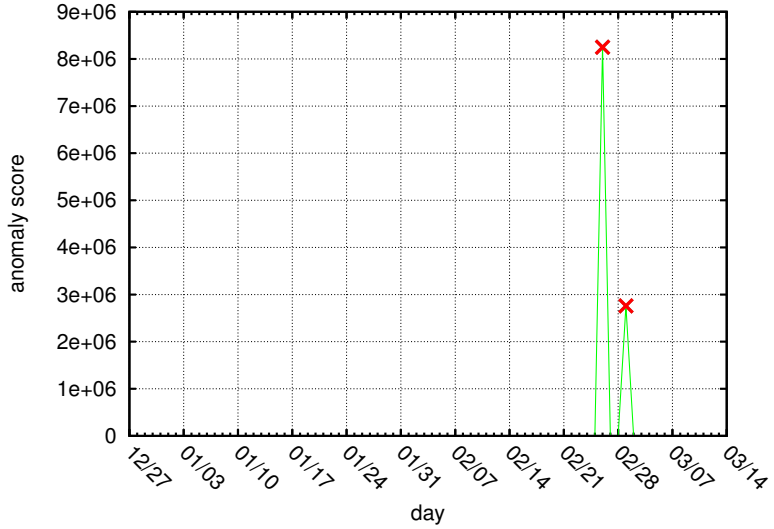


Figure 6.69: Daily anomaly score for the `pi` server.

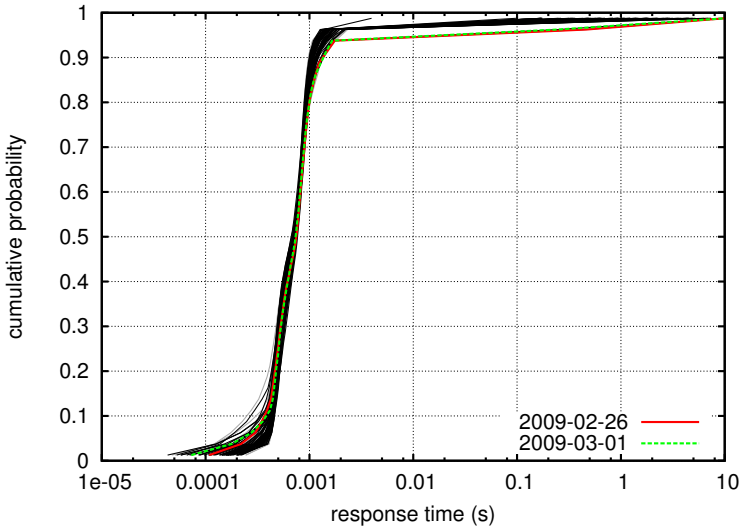


Figure 6.70: Response time distributions for the `pi` server, with the anomalous distributions highlighted.

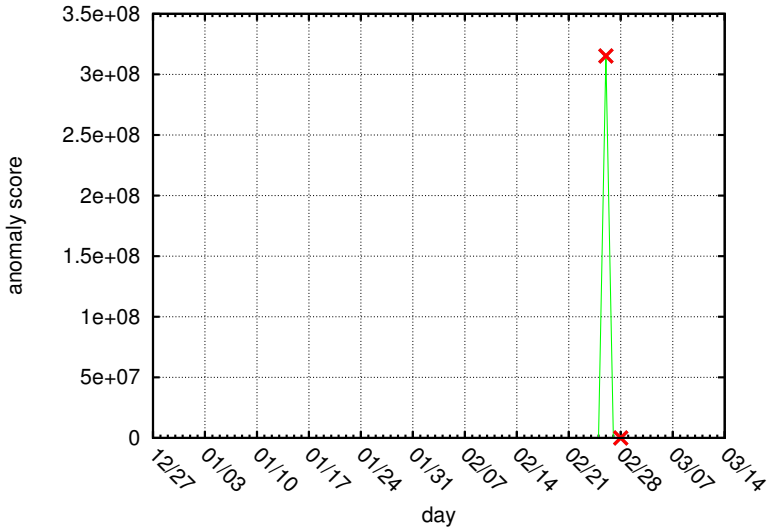


Figure 6.71: Daily anomaly score for the `rho` server.

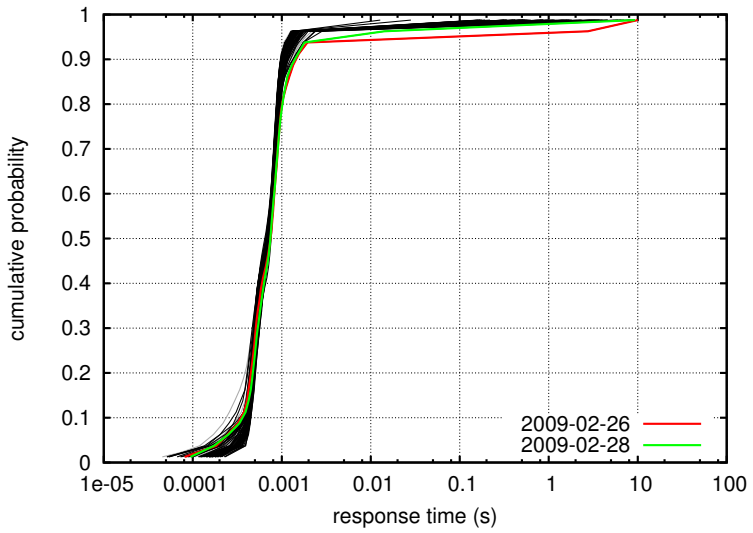


Figure 6.72: Response time distributions for the `rho` server, with the anomalous distributions highlighted.

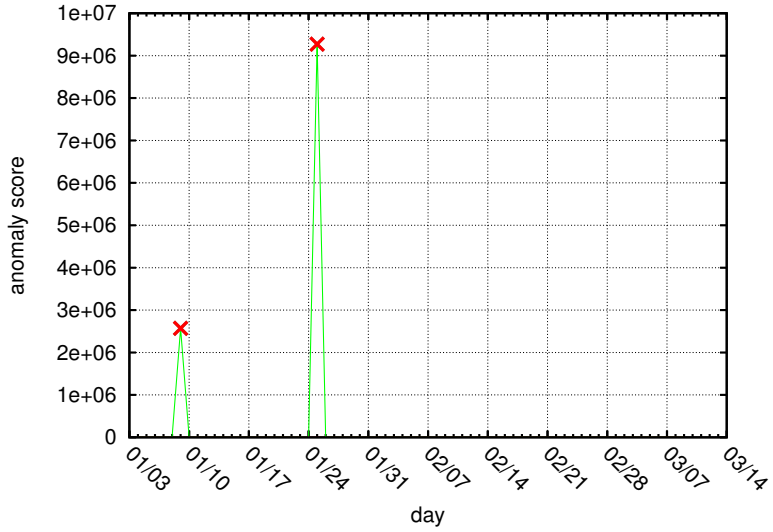


Figure 6.73: Daily anomaly score for the `sigma` server.

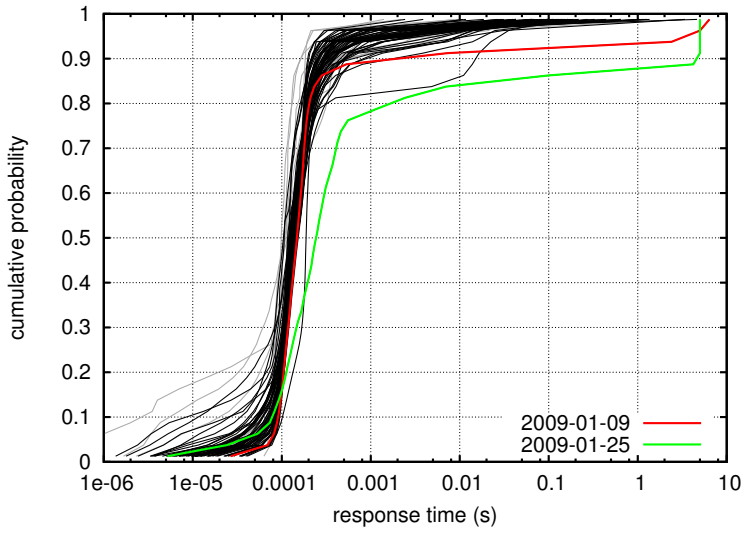


Figure 6.74: Response time distributions for the `sigma` server, with the anomalous distributions highlighted.

Anomalies 16 and 17 are for server `sigma`, and the anomaly scores are shown in Figure 6.73. Figure 6.74 shows that there were legitimate performance issues on these days. The distribution for January 25, 2009 (anomaly 17) shows that there is a timeout setting for the server of five seconds, which means that Nagios will not detect any performance issues regardless of the severity unless its threshold is set below five seconds.

Anomalies 18–21 are for server `tau`, which is actually on a different port of the same system as server `sigma`. The anomaly scores are shown in Figure 6.73. The distributions do not look very anomalous when plotted with a log-scaled X-axis in Figure 6.76. However, Figure 6.77 shows that, again, the 39^{th} bin (at Y=0.9625) again has very low variance in the normal basis.

6.5 Conclusions

This chapter compared a set of performance issues identified by a Nagios instance monitoring the web systems group of UNC against a set of anomalies identified by my methods over the same systems. There was very little agreement between Nagios and my methods, and most of this chapter was an explanation for why this was the case. I will summarize the primary observations here.

First, in many cases, `adudump` observed very little traffic for many servers. This could

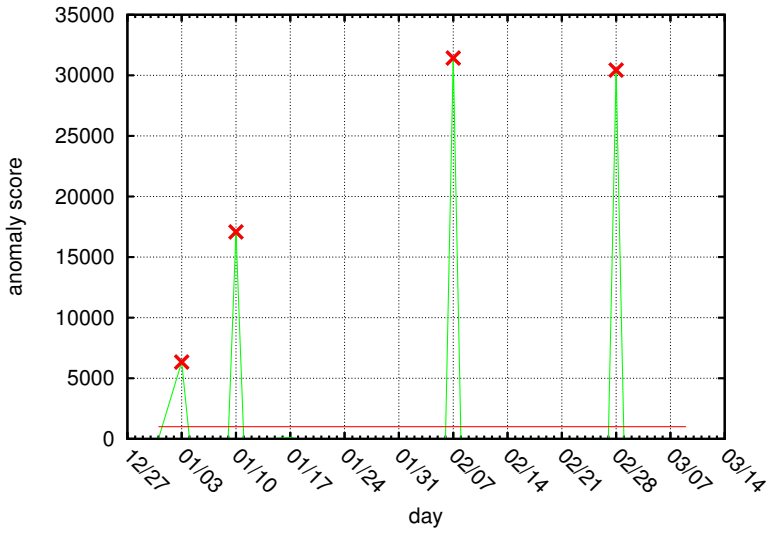


Figure 6.75: Daily anomaly score for the `tau` server.

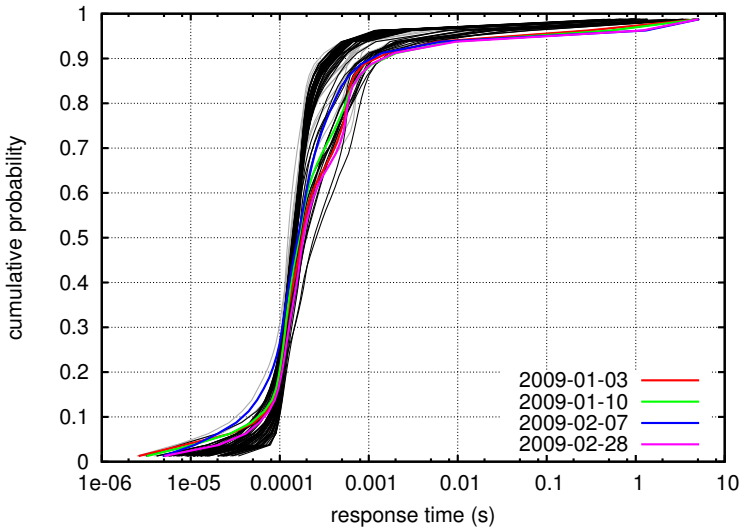


Figure 6.76: Response time distributions for the `tau` server, with the anomalous distributions highlighted.

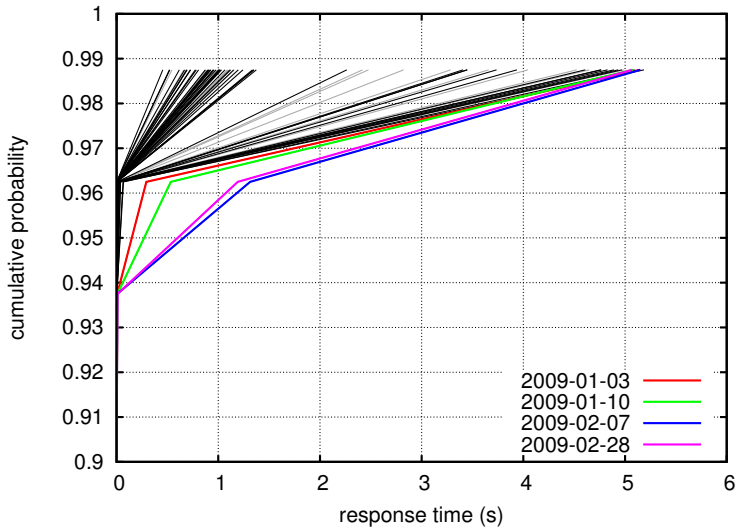


Figure 6.77: Last four bins of the response time distributions for the `tau` server, with the anomalous distributions highlighted.

happen because the server was not in the server VLAN, or the server's IP address could have changed between the monitor and the server (*e.g.* with a NAT box or load balancer), or most of the traffic could have originated within the server VLAN, or there might simply have been very little traffic to the server. Each of these possibilities illustrates the important things to keep in mind when choosing a vantage point. The case of having little traffic identifies a fundamental weakness of passive measurement in general.

Second, in many cases, Nagios appears simply to be unfortunate in its measured response times in that it happens to have multiple improbably large response times. Even when the response time distributions measured by `adudump` are not unusual, if there is a possibility of seeing a response time over the threshold, Nagios can still declare a performance issue if it happens to measure three such response times in a row. These cases are often accompanied by a relatively high number of "soft" Nagios issues that do not finally result in "hard" issues. Related to this case were many cases in which I simply could not explain why Nagios identified a performance issue.

Third, I observe that, in some cases, the anomaly score seems to be disproportionately high. The distribution being tested against the normal basis is indeed anomalous, but the anomaly score seems extreme (*e.g.* in the millions). This occurs when (a) the distributions comprising the normal basis exhibit low variation in all bins except for the last one, and (b) the test

distribution has a value well off of the envelope of distributions in a particular bin—usually the 39^{th} bin. This is a concern with my methods that I would like to explore for future work.

Lastly, I also want to note that an `anom_thres` setting of 1000 seems to offer more reasonable results, with fewer anomalies, than the default setting of `50` that I used in previous chapters. This is also a topic for future work. Perhaps, because anomaly scores seem to grow multiplicatively, it would be better to threshold based on the logarithm of the anomaly score.

In general, I argue that, if a good vantage point can be found, then my methods give better results than the probing approach adopted by Nagios. Nagios is more prone to sampling error than my methods, which aggregate many response times together. Furthermore, my methods do not, by their measurement, affect the servers themselves, like a probing approach does. However, my methods could be augmented by a probing approach for those servers that see very little traffic.

Chapter 7

Conclusions

This dissertation has introduced a novel approach to managing the performance of servers within a large network with a completely passive approach. I will discuss the contributions and insights of this dissertation in the order they were presented.

In Chapter 3 I introduced `adudump`, which builds a model of the application-level dialog of a TCP connection based merely on its sequence numbers. Because of its general approach, `adudump` can measure request sizes, response sizes, think times, and other application-level aspects of TCP connections regardless of the application protocol carried over the connection. The measurements are also unhindered by encryption. `adudump` is also efficient, and can process at least 600 Mbps of traffic, even on relatively old hardware. The server-side think time, or response time, is a reflection of the performance of a server: the higher the response time, the worse the server is performing. Thus, `adudump` provides a metric for the server performance. When used on the border link or other aggregation point of a large network, it measures server performance for a broad set of servers within the network. Another advantage is that `adudump` reports response time and other metrics as soon as they occur.

Chapter 4 described my performance anomaly detection methods, which were adapted from a medical image analysis application. At a high level, the methods aggregate server response times into distributions, build a profile for a particular server from a set of the server's response time distributions, and then compare new distributions to the profile to determine whether the new distribution is anomalous. Thus, given these methods, I can detect cases in which the performance of a server is not up to its typical level—again, all from a passive approach relying

only on TCP sequence numbers and packet arrival timestamps. The methods are subject to a number of parameters, which enable a network manager to tune the methods to a particular network or circumstance. One major advantage of the methods is that they maintain the generality of `adudump`'s measurement, working well for a wide variety of servers, regardless of the typical range or variance of response times.

Chapter 5 explored several case studies of applying the performance anomaly detection methods to real servers in UNC's network. One major contribution is that the methods detected legitimate performance issues that were of interest to network managers for a variety of server types including HTTP, SMTP, and POP3. Furthermore, the case studies highlighted the richness of the data measured by `adudump` and showed the usefulness of the data in diagnosing the cause of the performance issues. Again, it is remarkable that performance issues can be not only detected but also diagnosed with a purely passive approach that works for many different types of servers. The chapter also provided evidence for a reasonable default setting for the parameter values.

Chapter 6 compared my methods to Nagios, an existing network management tool in use at UNC. Unlike my approach, Nagios generates and sends requests to specific application services (*e.g.* HTTP) to measure the performance, which has three disadvantages. First, the measurement affects the service itself. Second, the measurement is specific to a particular application, and cannot be applied to other types of servers. Third, it is very difficult to determine whether the requests Nagios generates are representative of the other requests the server receives in the ways that matter, *e.g.* in terms of the work required by the server. In addition to these drawbacks, Nagios does a simple threshold approach on the measured response times. The comparison showed that the issues identified by Nagios often do not correspond with atypical response time distributions; in other words, Nagios reports an issue when there does not appear to be one. Furthermore, because the approach is based on single response time measurements and static thresholds, Nagios can easily miss important performance issues. My approach to performance anomaly detection is an improvement to the approach adopted by Nagios in all of these ways.

In summary, I have shown that it is possible to detect and, to an extent, diagnose server performance issues in a large network providing various types of services without perturbing

those services, by accessing only the information available in the TCP/IP headers of the packets the server receives and sends.

Future Work

I now consider a variety of ideas for future work.

Methodological improvements

The discussion in this dissertation has identified various improvements and additions that could be made to my performance anomaly detection methods. I list them here.

- upgrade `adudump` to report distinct records for ADU records containing response times, client-side think-times, and quiet-times (Section 3.3)
- allow more than two exclusions from the training set (Section 4.6)
- identify situations in which other representations (*e.g.* aPDF, aCDF) are better choices (Section 4.2)
- explore basis divergence (Section 5.6) and propose a solution
- weight quarter-hour bins equally in the rolling-day analysis, so that newer or older quarter-hours influence the overall distribution by a standard amount (Section 5.5)
- automatically identify which quarter-hour bins contribute to an anomaly (Section 5.9)
- automatically diagnose performance issues (Section 5)
- quantify the effect of sampling error and apply performance anomaly detection to servers with fewer observations (Section 6)

Measurement

I also have several ideas for extensions I could make to `adudump`, either in terms of the data it collects, the type of connections it analyzes, or the measurement context.

- infer A–B–T connection vectors from a single direction of TCP traffic using sequence numbers as well as acknowledgement numbers

- infer performance from concurrent connections (when possible)
- infer performance from non-TCP connections (when possible)
- create a lightweight, possibly inline `adudump` monitor system
- instrument kernel(s) or driver(s) with application-level socket information, to supplement and validate `adudump` inference algorithms

Miscellaneous

Lastly, I have some ideas for other directions to explore.

- group related servers together (*e.g.* the four campus SMTP servers from Section 5.5), and leverage correlations
- automatically identify candidate groups of servers based on connection structure
- design a generic "session" structure that identifies groups of related connections between two IP addresses, *e.g.* simultaneous HTTP connections, and use this session structure to further diagnose anomalies
- explore new notions of response time, *e.g.* the time elapsed from the beginning of a request to the end of a response
- incorporate organizational knowledge into my methods, *e.g.* network maps, dependency graphs, and load-balanced systems
- enable manual overrides to anomaly detection methods, and automatically suggest parameter settings to accommodate the overrides
- create real-time (minimal delay) performance anomaly detection methods

Appendix A

Full Results

This appendix contains the results of doing the per-day anomaly test over the corpus of Section 4.7. Two server sockets were excluded because they had too few days to create a basis for normality (*i.e.* the number of days is less than `tset_size`). The other 227 server sockets are represented with the anomaly score timeseries plot. Incomplete days due to outages are represented with black points, and are the same for all servers. Incomplete days due to a lack of data for the server are represented with gray points.

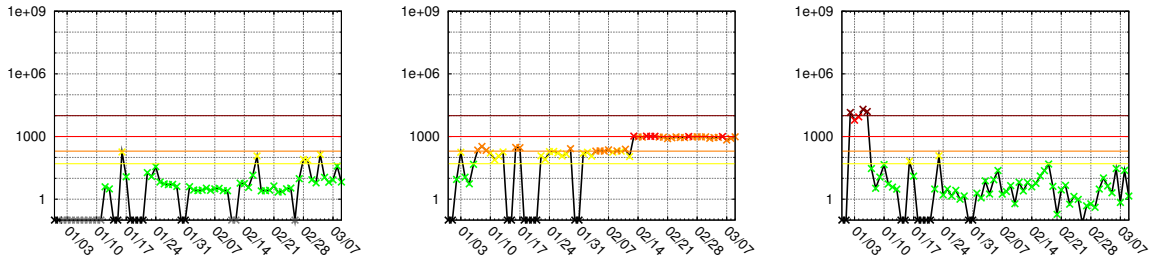
I use the standard parameter settings given in Section 4.7: `tset_size` = 21, `basis_error` = 0.01, `anom_thres` = 50, `window_size` = 60, and `num_bins` = 40. In some cases, the `anom_thres` setting was too restrictive, resulting in a basis that could not adapt to a slow shift. I list those cases as they are encountered.

Note that there are many servers with a sustained burst of *error* and *critical* anomalies. This finding argues for at least one of the following cases:

- The anomalies accurately reflect performance issues in the network, which are quite common.
- The anomaly threshold is overly sensitive, and should be increased.
- The anomaly threshold is overly sensitive *in some cases* (such as the non-linear variation of Server 191), and should be increased in those cases.

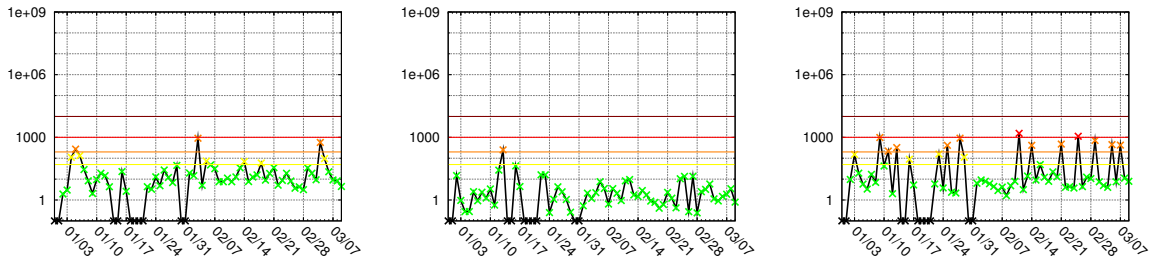
In addition, I also note that some servers (such as Server 125) have significant anomalies, yet the response times stay relatively small (*e.g.* below 150 milliseconds). For non-interactive applications, users are much more tolerant of delays in this range, so such anomalies probably will not be of interest to ITS staff. Therefore, I note for future work that it might be useful, if interactive applications can be distinguished from non-interactive applications, to augment

the anomaly score with an absolute response time threshold, so that anomalous yet tolerable response times are less likely to catch the attention of the network manager.



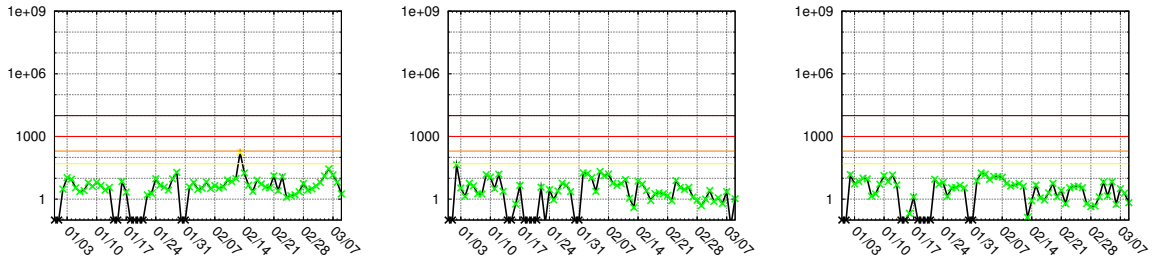
(a) server 1 (b) server 2 (c) server 3

Figure A.1: Anomaly score timeseries for servers 1–3. Server 2 is used as an example in Section 4.7.5 of a server sensitive to the setting of `anom_thres`. Server 3 is a portal web server for one of the colleges on campus (not the one mentioned in Chapter 5). During Jan 2–6, much fewer response times are less than 100 milliseconds than normal; however, the distribution is the same for response times greater than half a second, so it is likely that few users noticed a significant difference, despite the `error` and even `critical` anomaly scores.



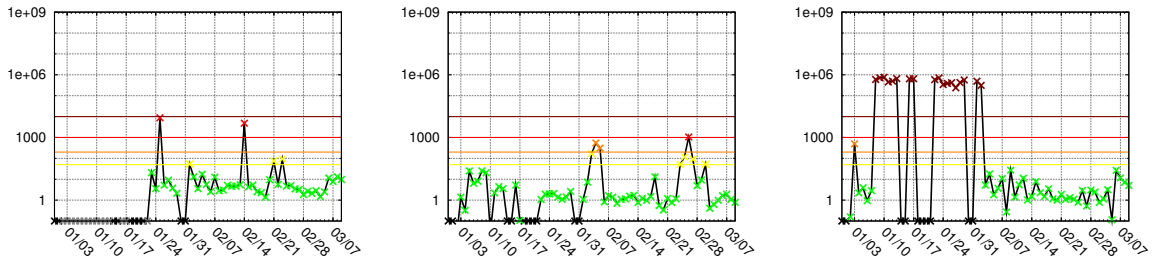
(a) server 4 (b) server 5 (c) server 6

Figure A.2: Anomaly score timeseries for servers 4–6



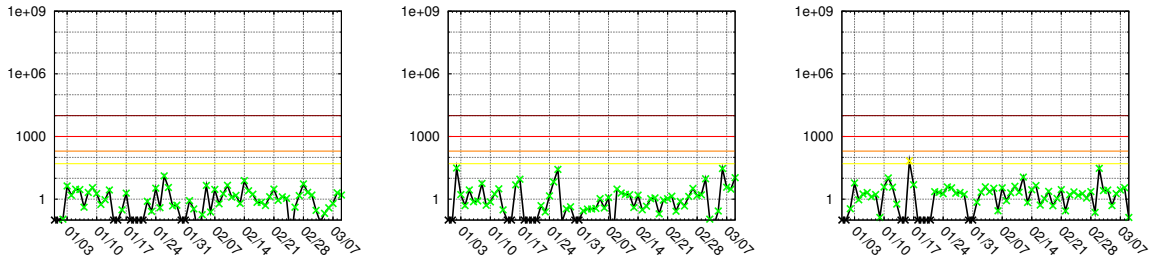
(a) server 7 (b) server 8 (c) server 9

Figure A.3: Anomaly score timeseries for servers 7–9



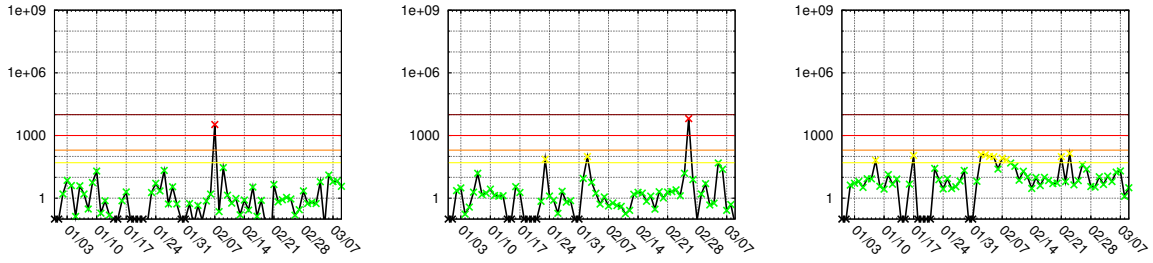
(a) server 10 (b) server 11 (c) server 12

Figure A.4: Anomaly score timeseries for servers 10–12. Server 10 is an example of a server having too few response times to form a 40-bin QF for many of the days measured. As a result, bootstrapping occurs later, which is why many points show up as grey/black in the plot: the bootstrapping has not yet occurred, so a normal basis has not yet been formed. The last QF bin of server 12 is typically at most two seconds, yet between Jan 8–Feb 2, the last two QF bins are both between 2–3 seconds.



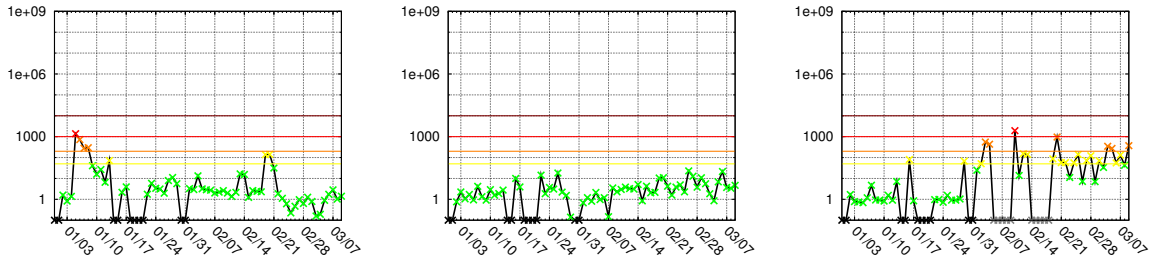
(a) server 13 (b) server 14 (c) server 15

Figure A.5: Anomaly score timeseries for servers 13–15



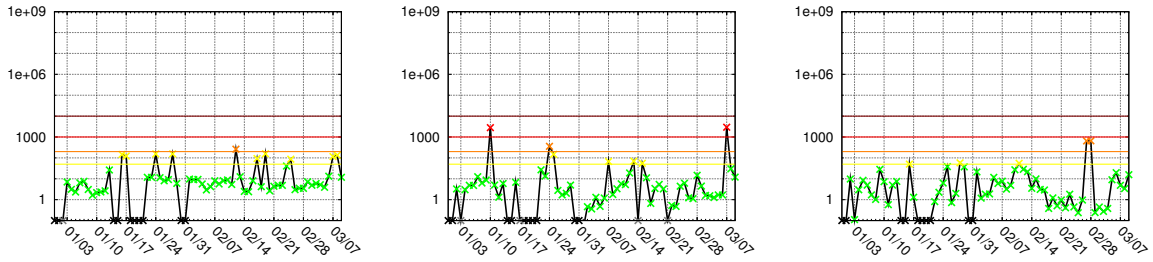
(a) server 16 (b) server 17 (c) server 18

Figure A.6: Anomaly score timeseries for servers 16–18



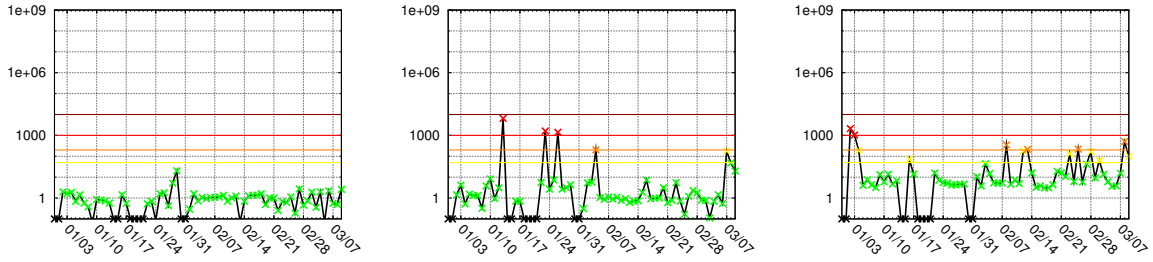
(a) server 19 (b) server 20 (c) server 21

Figure A.7: Anomaly score timeseries for servers 19–21



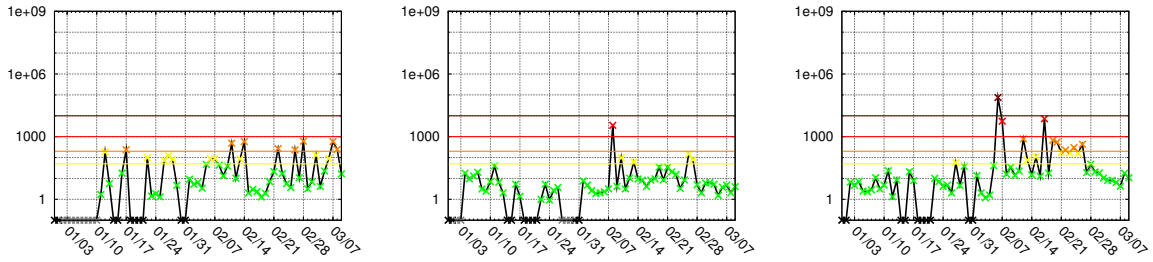
(a) server 22 (b) server 23 (c) server 24

Figure A.8: Anomaly score timeseries for servers 22–24



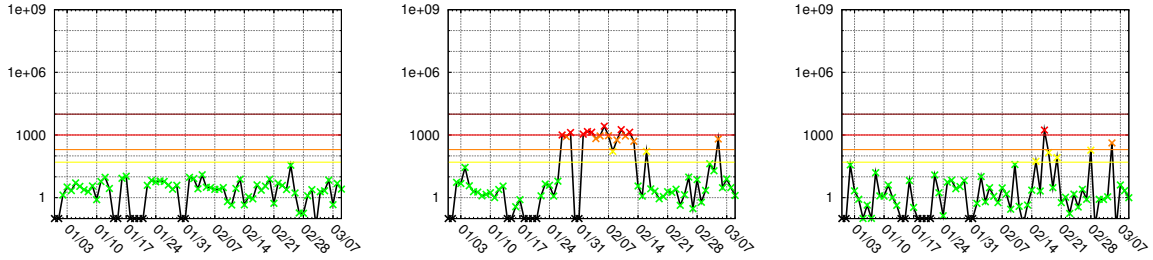
(a) server 25 (b) server 26 (c) server 27

Figure A.9: Anomaly score timeseries for servers 25–27



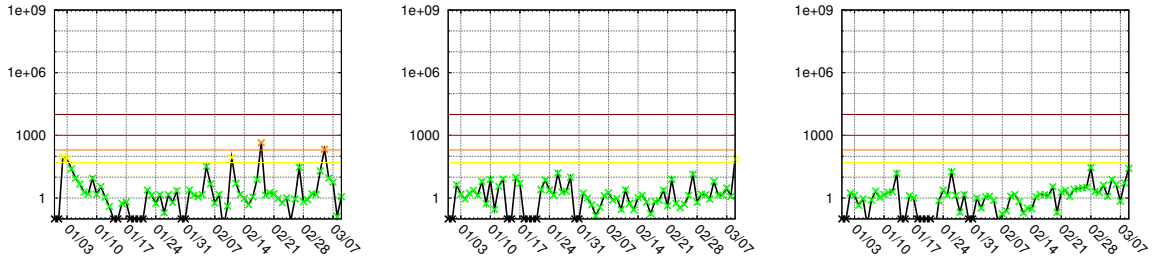
(a) server 28 (b) server 29 (c) server 30

Figure A.10: Anomaly score timeseries for servers 28–30



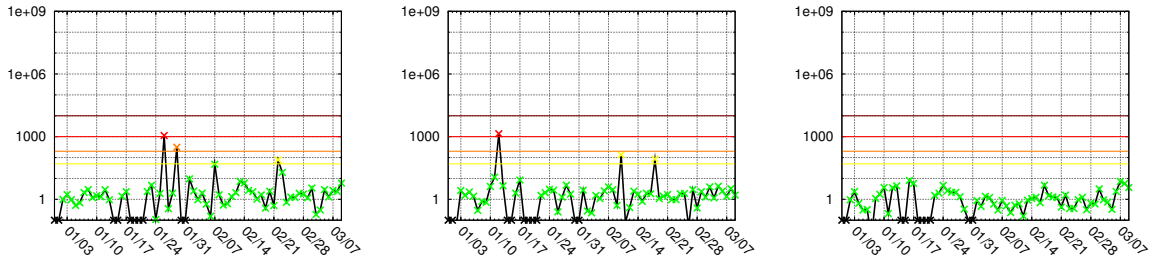
(a) server 31 (b) server 32 (c) server 33

Figure A.11: Anomaly score timeseries for servers 31–33



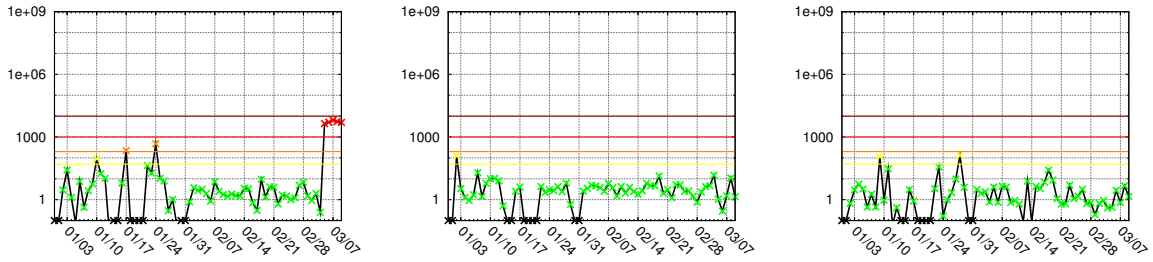
(a) server 34 (b) server 35 (c) server 36

Figure A.12: Anomaly score timeseries for servers 34–36



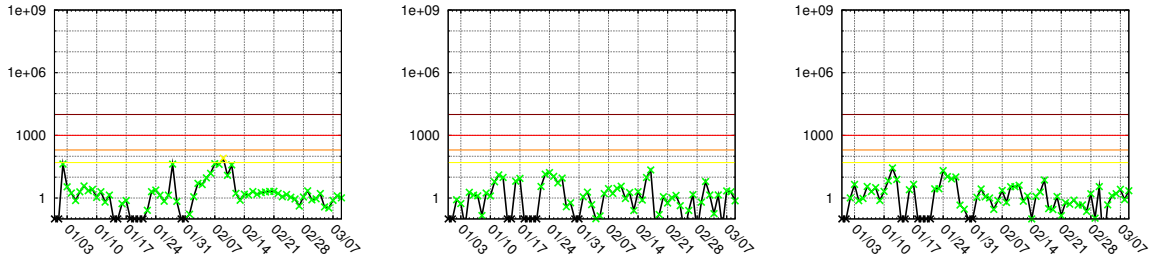
(a) server 37 (b) server 38 (c) server 39

Figure A.13: Anomaly score timeseries for servers 37–39



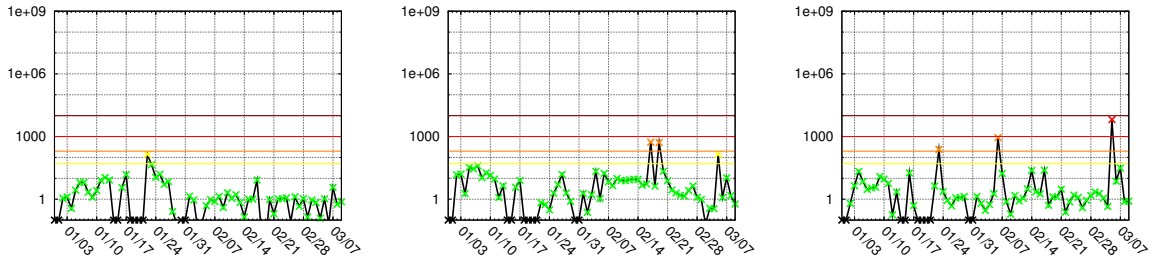
(a) server 40 (b) server 41 (c) server 42

Figure A.14: Anomaly score timeseries for servers 40–42



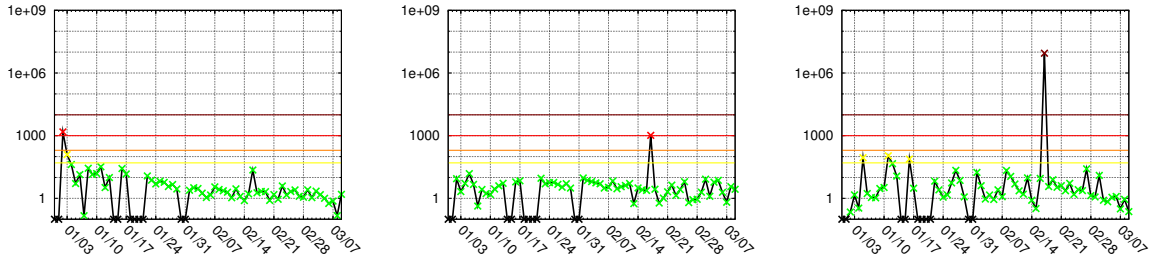
(a) server 43 (b) server 44 (c) server 45

Figure A.15: Anomaly score timeseries for servers 43–45



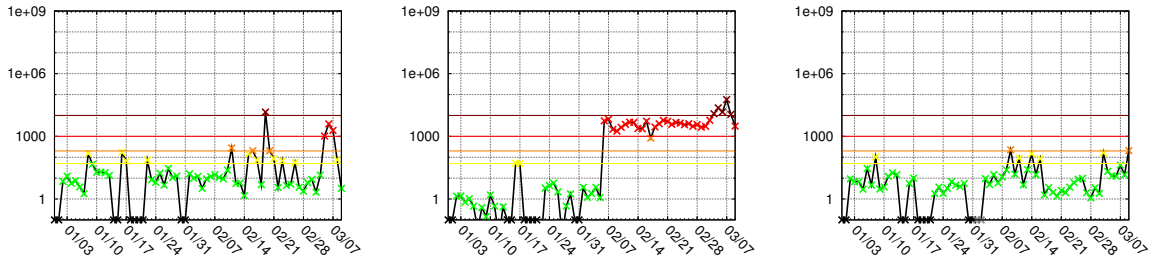
(a) server 46 (b) server 47 (c) server 48

Figure A.16: Anomaly score timeseries for servers 46–48



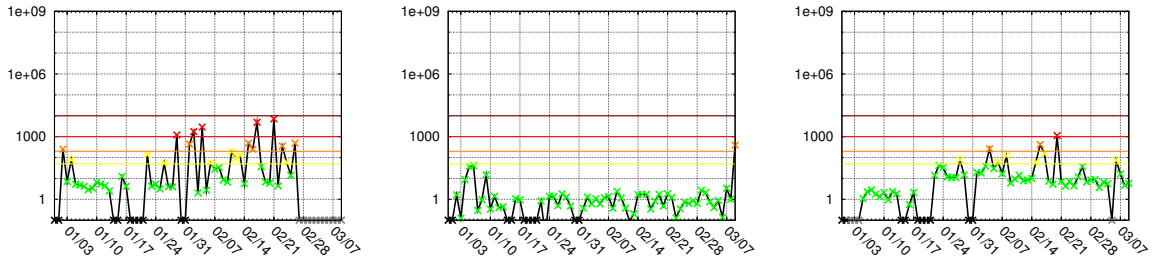
(a) server 49 (b) server 50 (c) server 51

Figure A.17: Anomaly score timeseries for servers 49–51. Server 51 had a severe `critical` anomaly (with a score of over 8.8 million) on Feb 17, when the last QF bin jumped from its typical value of around 20–100 milliseconds to nearly 20 seconds. ITS reported that the cause of this problem was with the authentication component of the service, which had issues after a DNS server outage.



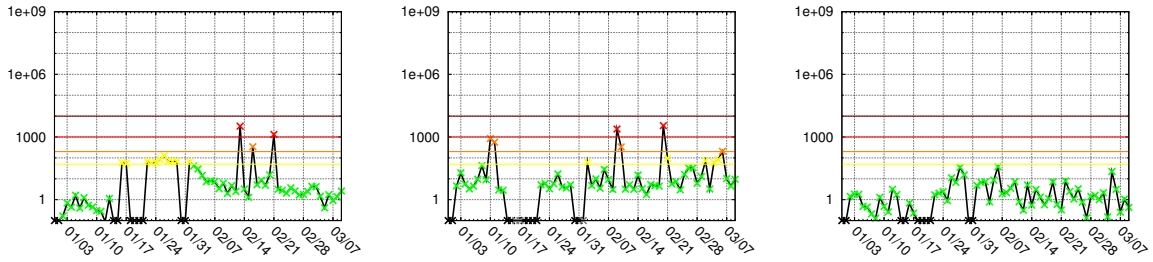
(a) server 52 (b) server 53 (c) server 54

Figure A.18: Anomaly score timeseries for servers 52–54. Server 53 experienced a sustained performance anomaly starting on Feb 6, and lasting at least through March 9.



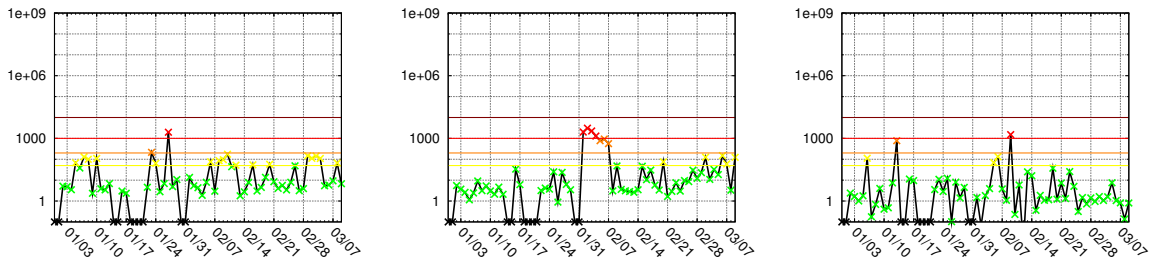
(a) server 55 (b) server 56 (c) server 57

Figure A.19: Anomaly score timeseries for servers 55–57



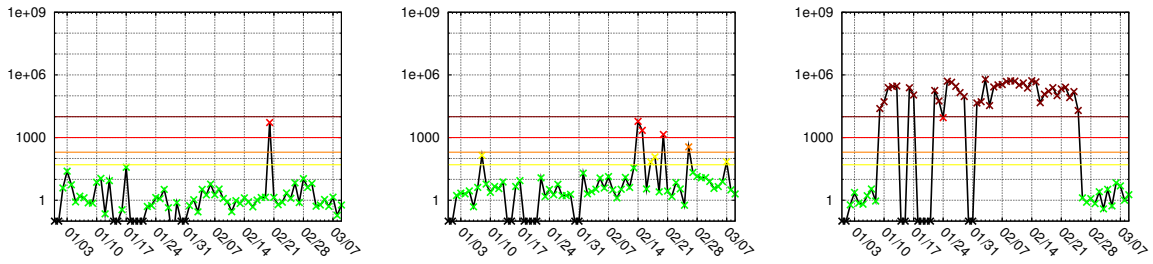
(a) server 58 (b) server 59 (c) server 60

Figure A.20: Anomaly score timeseries for servers 58–60



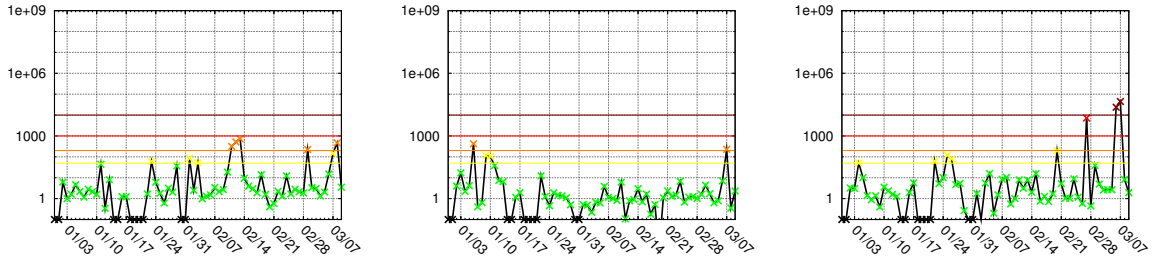
(a) server 61 (b) server 62 (c) server 63

Figure A.21: Anomaly score timeseries for servers 61–63



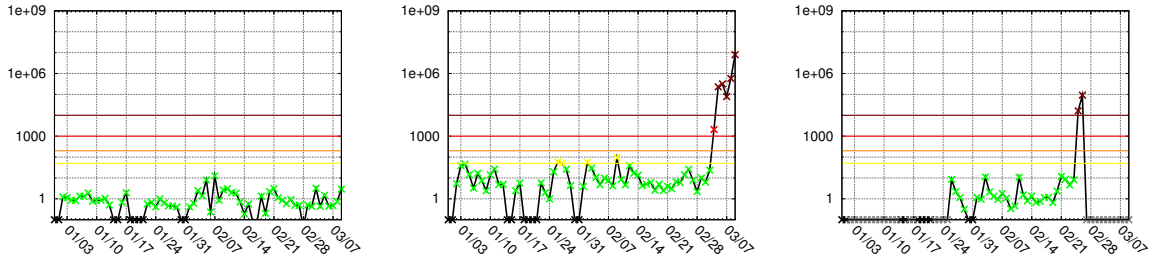
(a) server 64 (b) server 65 (c) server 66

Figure A.22: Anomaly score timeseries for servers 64–66. Server 66 experienced a sustained performance issue from Jan 9–Feb 25, but it is a dynamically-assigned address.



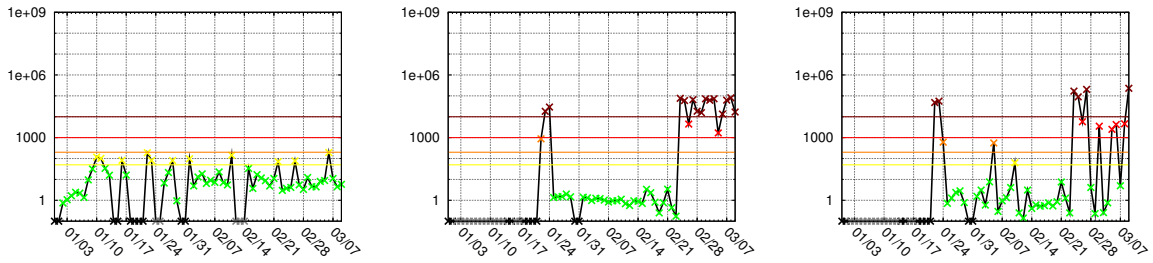
(a) server 67 (b) server 68 (c) server 69

Figure A.23: Anomaly score timeseries for servers 67–69



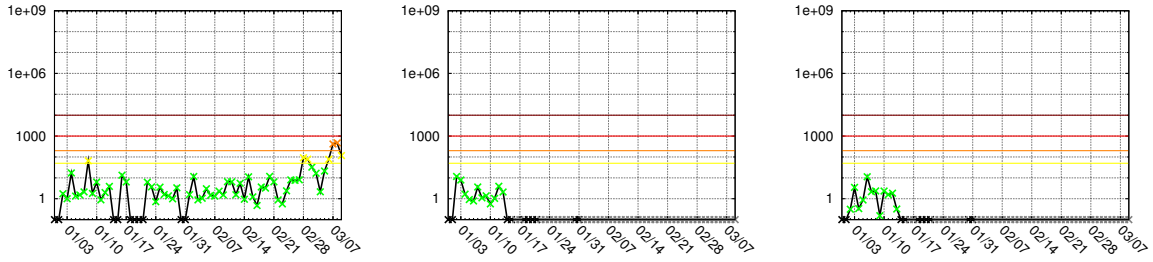
(a) server 70 (b) server 71 (c) server 72

Figure A.24: Anomaly score timeseries for servers 70–72. Server 71 experienced a performance issue starting on Mar 4 and continuing at least through Mar 9. Server 72 is one of many that were apparently shut down at some point prior to Mar 9.



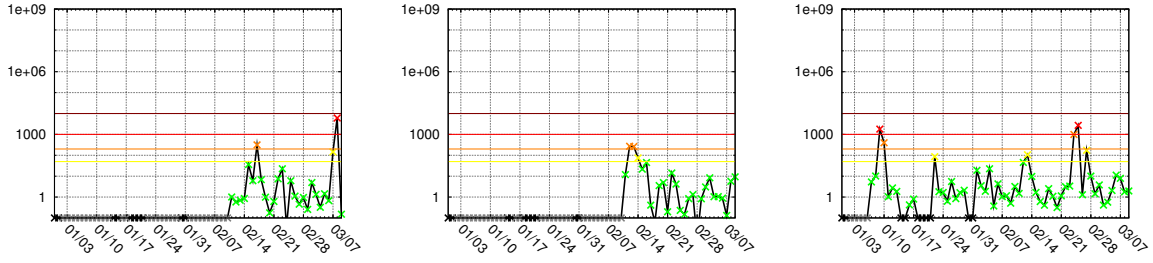
(a) server 73 (b) server 74 (c) server 75

Figure A.25: Anomaly score timeseries for servers 73–75. Server sockets 74 and 75 are actually the same DHCP-assigned IP address.



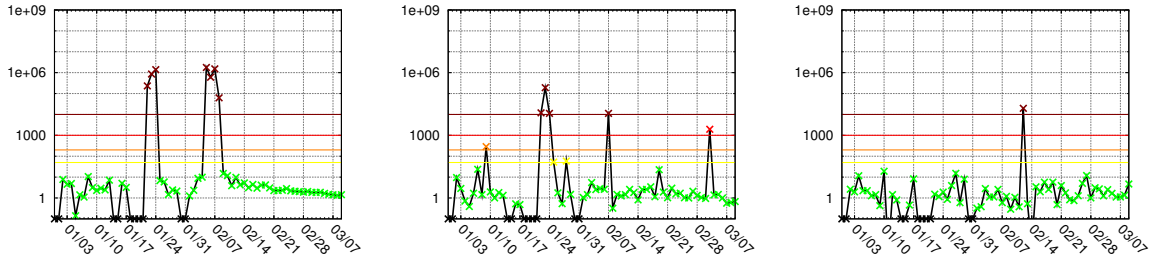
(a) server 76 (b) server 77 (c) server 78

Figure A.26: Anomaly score timeseries for servers 76–78



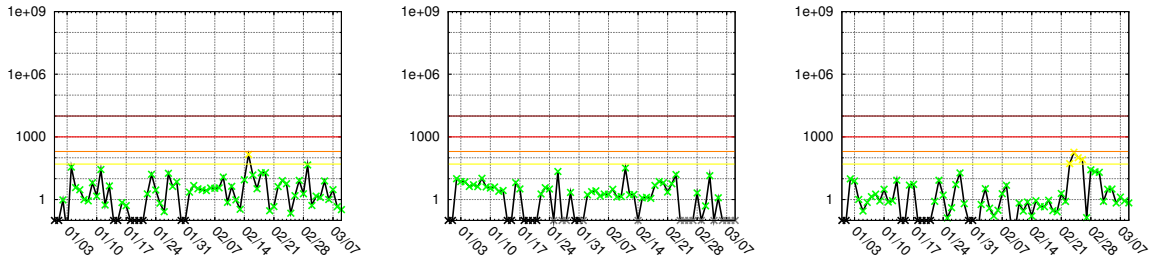
(a) server 79 (b) server 80 (c) server 81

Figure A.27: Anomaly score timeseries for servers 79–81



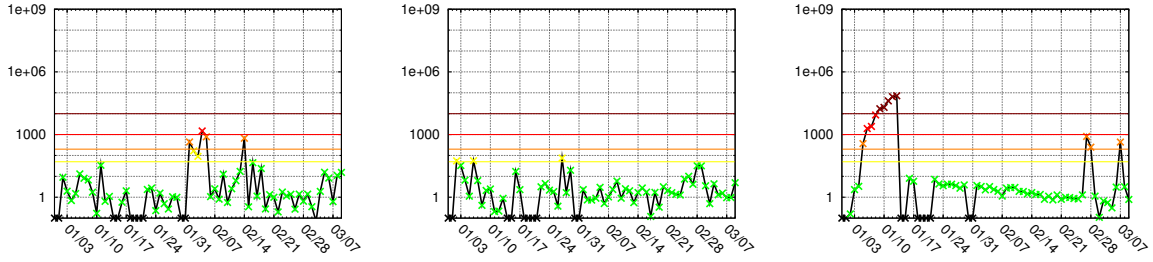
(a) server 82 (b) server 83 (c) server 84

Figure A.28: Anomaly score timeseries for servers 82–84. Server 82 saw two `critical` performance issues: one from Jan 22–24, and one from Feb 5–8. Server 83 experienced issues on mostly the same days. In fact, server (socket) 83 is the same IP address as server (socket) 82.



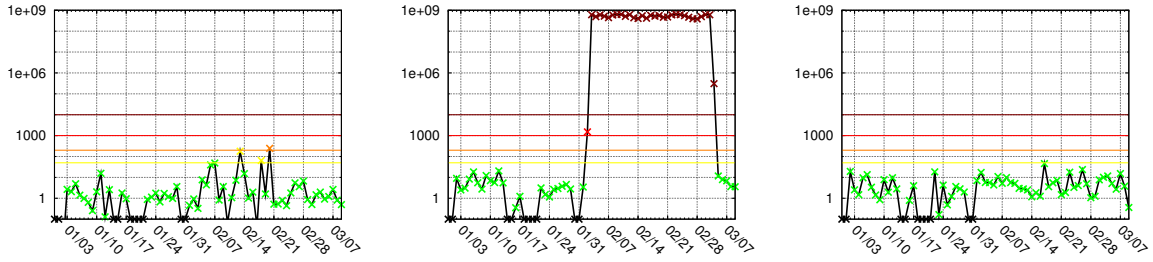
(a) server 85 (b) server 86 (c) server 87

Figure A.29: Anomaly score timeseries for servers 85–87



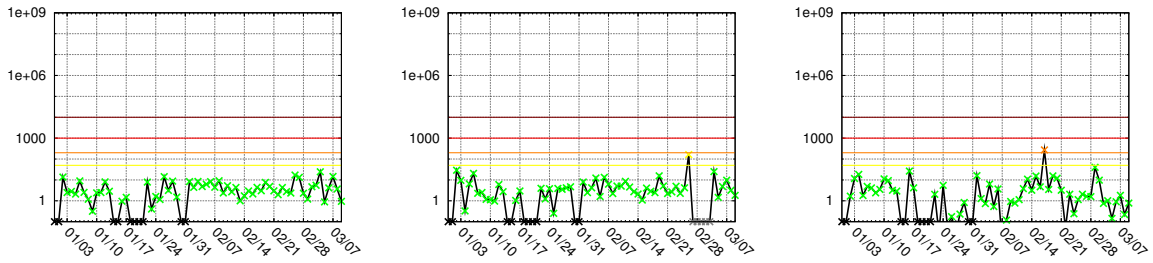
(a) server 88 (b) server 89 (c) server 90

Figure A.30: Anomaly score timeseries for servers 88–90. Server 90 is the registration web server discussed in Chapter 5.



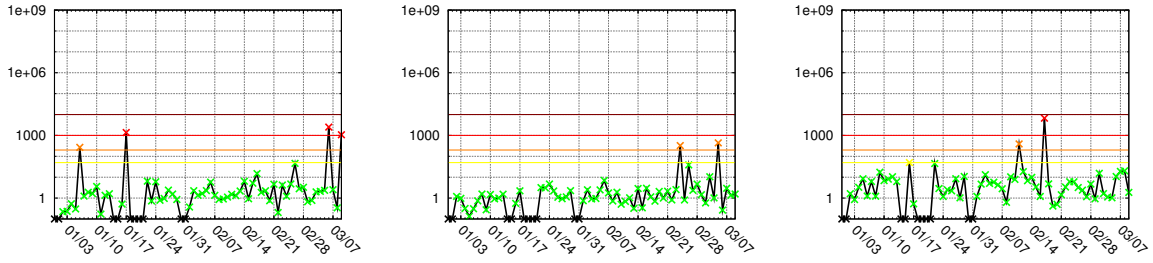
(a) server 91 (b) server 92 (c) server 93

Figure A.31: Anomaly score timeseries for servers 91–93. Server 92 has the most extreme anomalies in the entire corpus by a substantial margin. This server is the mail server discussed in Chapter 5.



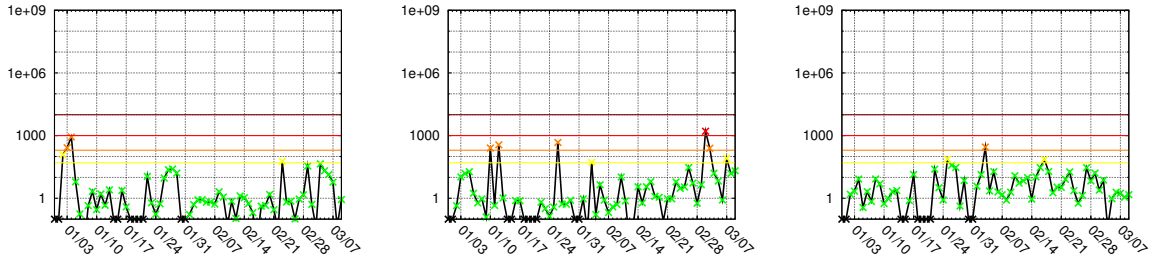
(a) server 94 (b) server 95 (c) server 96

Figure A.32: Anomaly score timeseries for servers 94–96



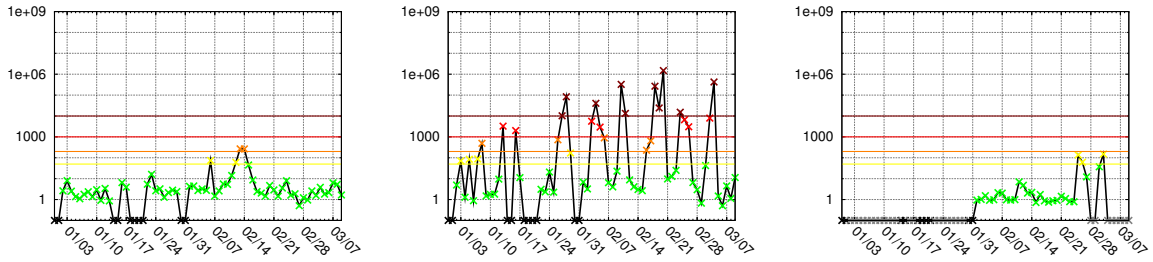
(a) server 97 (b) server 98 (c) server 99

Figure A.33: Anomaly score timeseries for servers 97–99



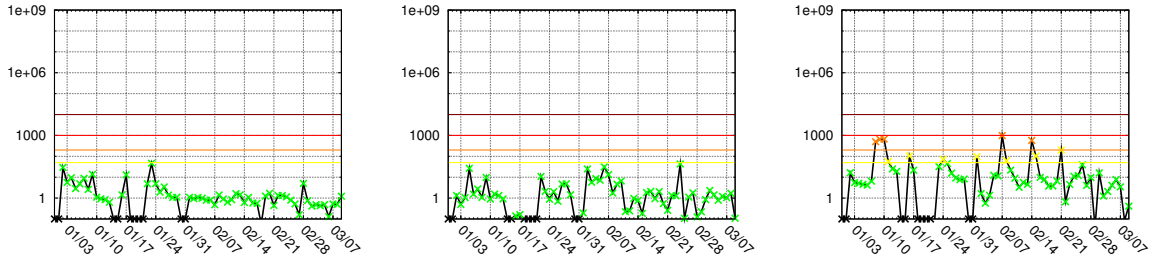
(a) server 100 (b) server 101 (c) server 102

Figure A.34: Anomaly score timeseries for servers 100–102



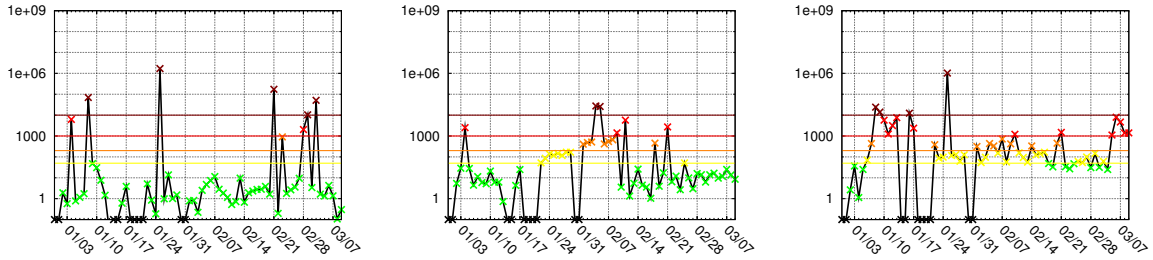
(a) server 103 (b) server 104 (c) server 105

Figure A.35: Anomaly score timeseries for servers 103–105. Server 104 is a video conferencing server; presumably, the anomalies (which all occur on weekdays) correspond to conferencing events and are part of the expected operation of the server.



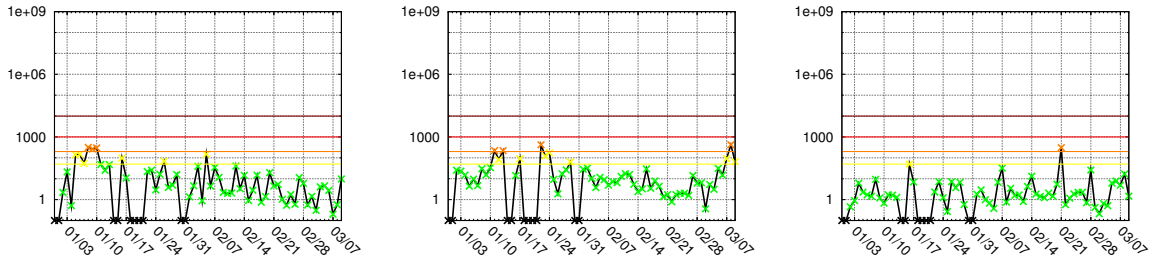
(a) server 106 (b) server 107 (c) server 108

Figure A.36: Anomaly score timeseries for servers 106–108



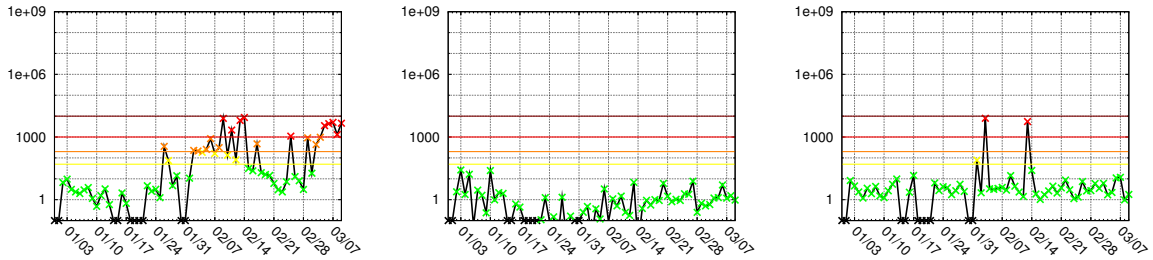
(a) server 109 (b) server 110 (c) server 111

Figure A.37: Anomaly score timeseries for servers 109–111. Server 109 has ten days (including two in the training set) in which the last QF bin is an outlier. Server 110 exhibits a slow ramp-up in anomaly scores like server 2, and changing the `anom_thres` parameter to a less strict setting of 200 reduces the anomaly count from 22 to 6. Server 111 also has a ramp up from Jan 26–Mar 5, and setting `anom_thres` = 200 yields zero anomalies in this date range. (Jan 25 remains a severe anomaly.)



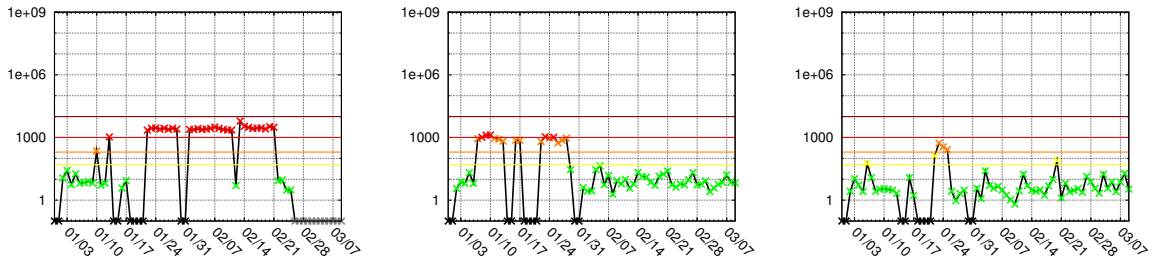
(a) server 112 (b) server 113 (c) server 114

Figure A.38: Anomaly score timeseries for servers 112–114



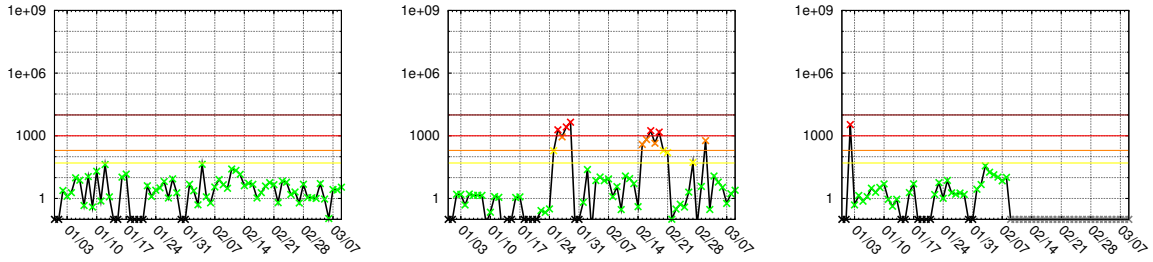
(a) server 115 (b) server 116 (c) server 117

Figure A.39: Anomaly score timeseries for servers 115–117



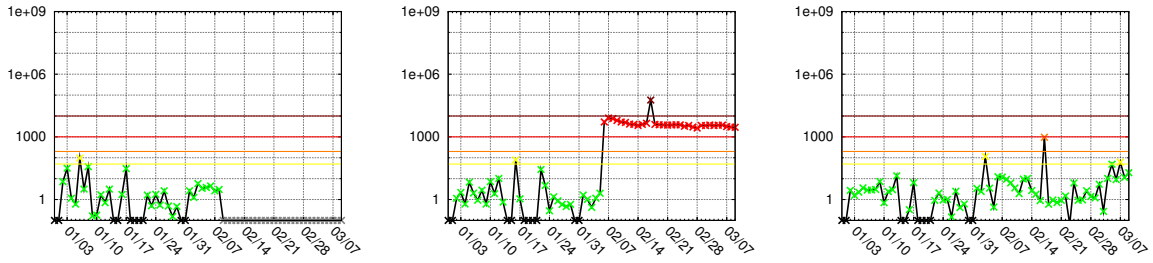
(a) server 118 (b) server 119 (c) server 120

Figure A.40: Anomaly score timeseries for servers 118–120. Server 118 sees a shift in the response time distributions starting Jan 22. It is a DHCP-assigned address and so probably not interesting to ITS staff.



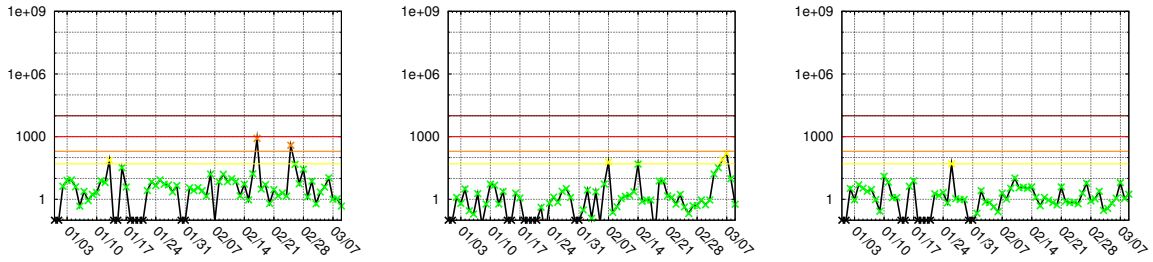
(a) server 121 (b) server 122 (c) server 123

Figure A.41: Anomaly score timeseries for servers 121–123



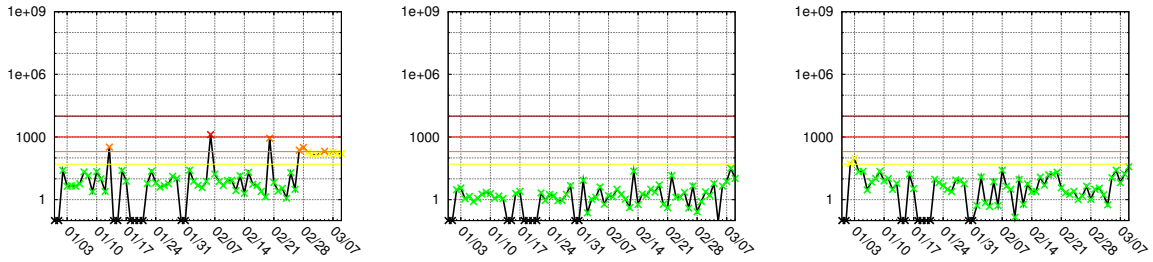
(a) server 124 (b) server 125 (c) server 126

Figure A.42: Anomaly score timeseries for servers 124–126. Server 125 had a sudden, pronounced shift in response time distributions starting on Feb 6. Except for the `critical` anomaly on Feb 17, the last QF bin is always less than 150 milliseconds. Because this server runs on port 443 (secure web), it is unlikely that the users would complain about such anomalies. This case argues that, if I can distinguish interactive use from non-interactive use, I might want to augment the anomaly score with an absolute metric.



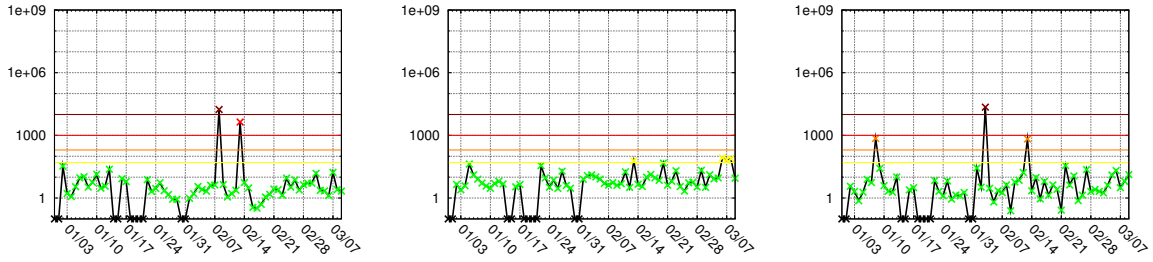
(a) server 127 (b) server 128 (c) server 129

Figure A.43: Anomaly score timeseries for servers 127–129



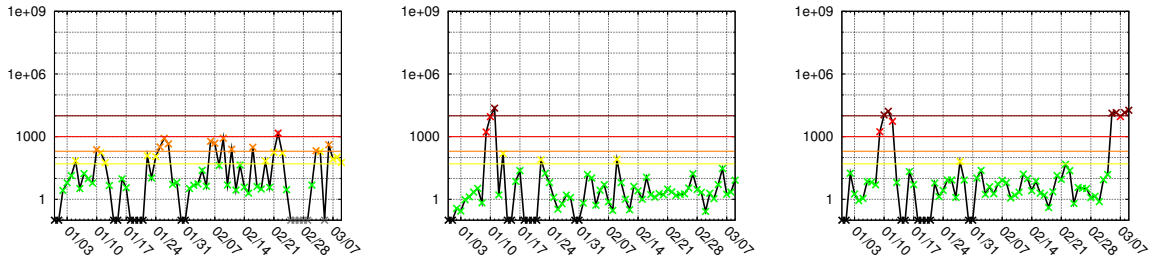
(a) server 130 (b) server 131 (c) server 132

Figure A.44: Anomaly score timeseries for servers 130–132



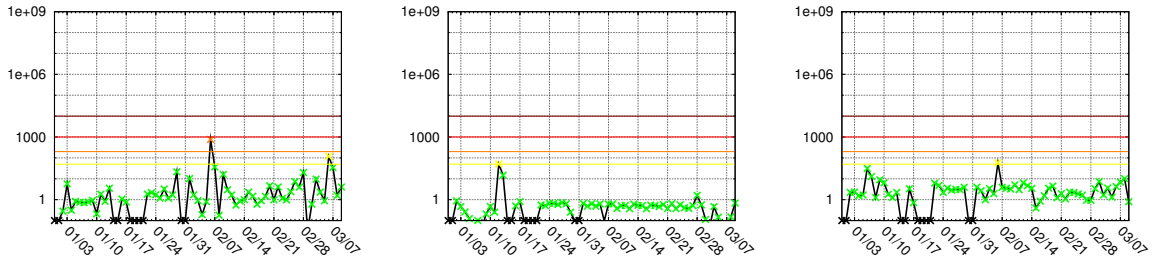
(a) server 133 (b) server 134 (c) server 135

Figure A.45: Anomaly score timeseries for servers 133–135



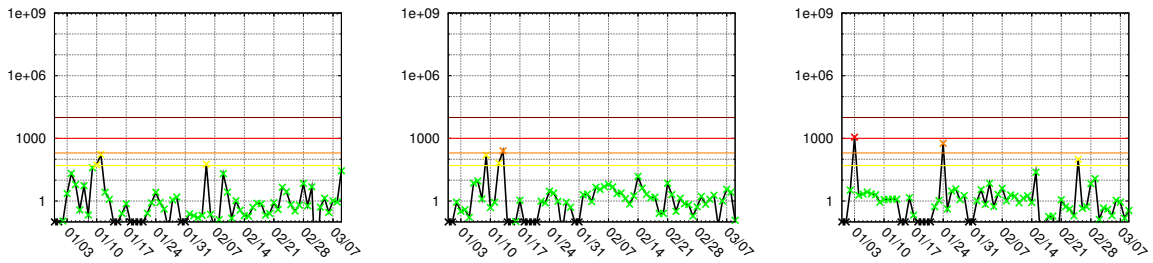
(a) server 136 (b) server 137 (c) server 138

Figure A.46: Anomaly score timeseries for servers 136–138



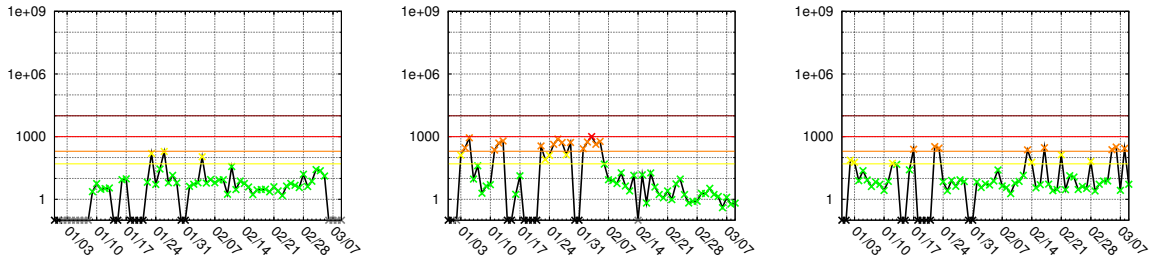
(a) server 139 (b) server 140 (c) server 141

Figure A.47: Anomaly score timeseries for servers 139–141



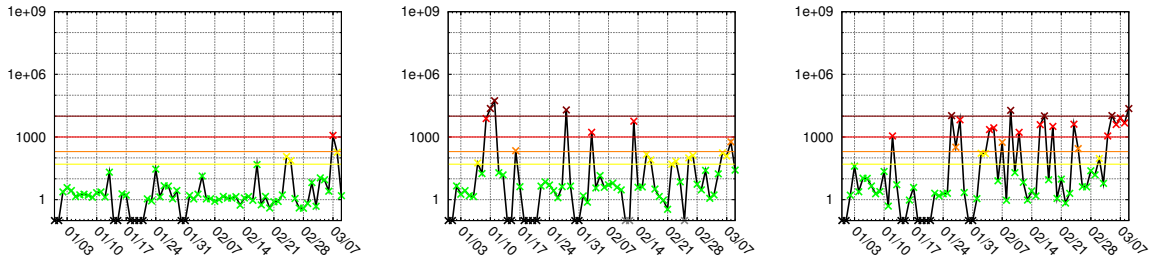
(a) server 142 (b) server 143 (c) server 144

Figure A.48: Anomaly score timeseries for servers 142–144



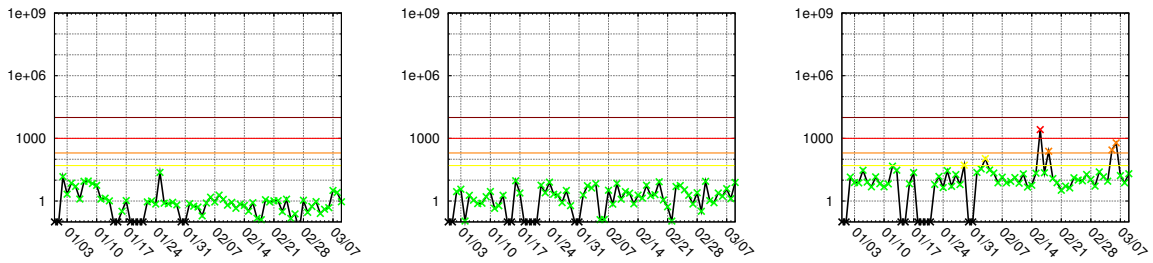
(a) server 145 (b) server 146 (c) server 147

Figure A.49: Anomaly score timeseries for servers 145–147



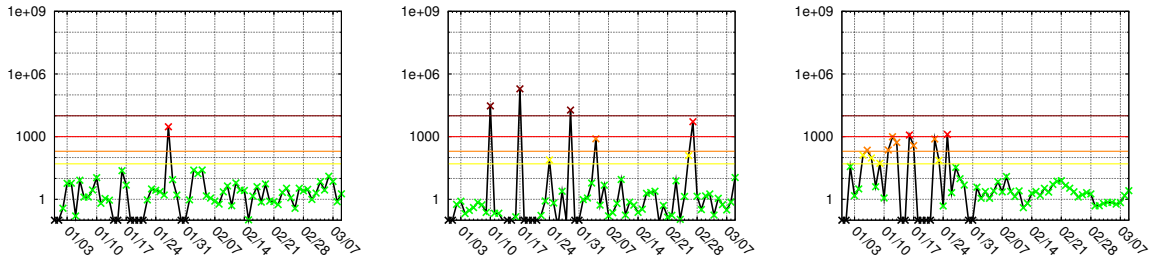
(a) server 148 (b) server 149 (c) server 150

Figure A.50: Anomaly score timeseries for servers 148–150



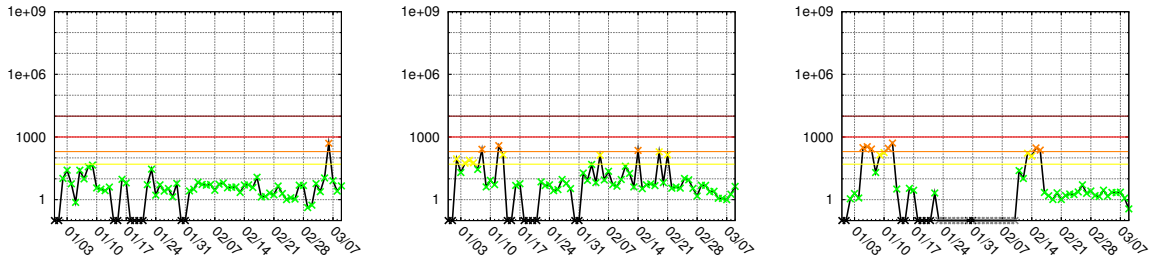
(a) server 151 (b) server 152 (c) server 153

Figure A.51: Anomaly score timeseries for servers 151–153



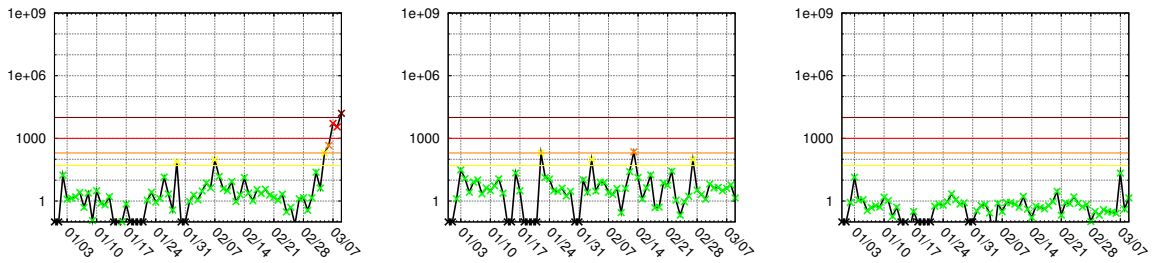
(a) server 154 (b) server 155 (c) server 156

Figure A.52: Anomaly score timeseries for servers 154–156



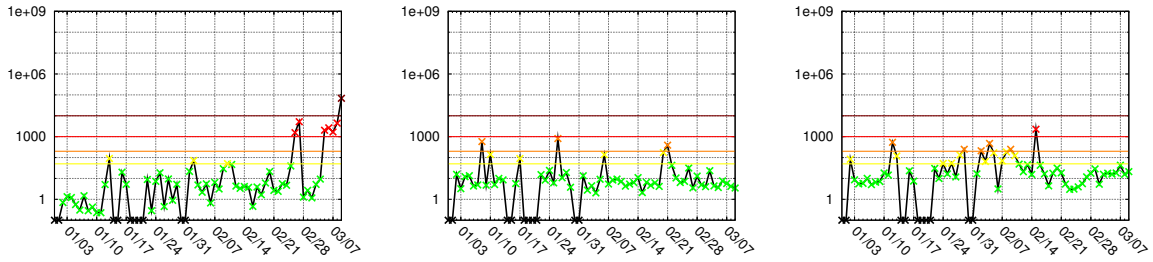
(a) server 157 (b) server 158 (c) server 159

Figure A.53: Anomaly score timeseries for servers 157–159



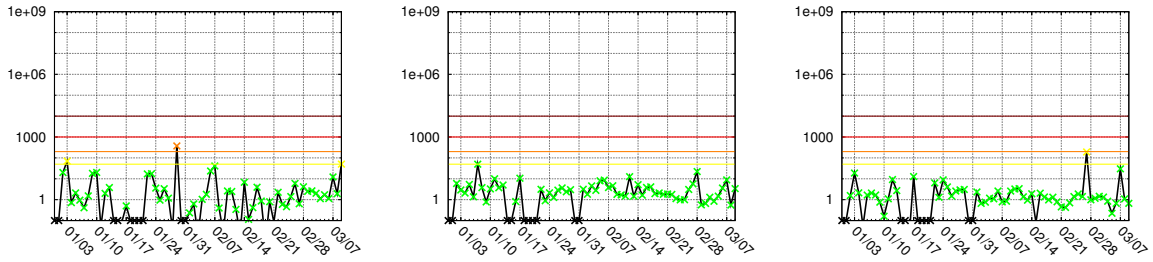
(a) server 160 (b) server 161 (c) server 162

Figure A.54: Anomaly score timeseries for servers 160–162



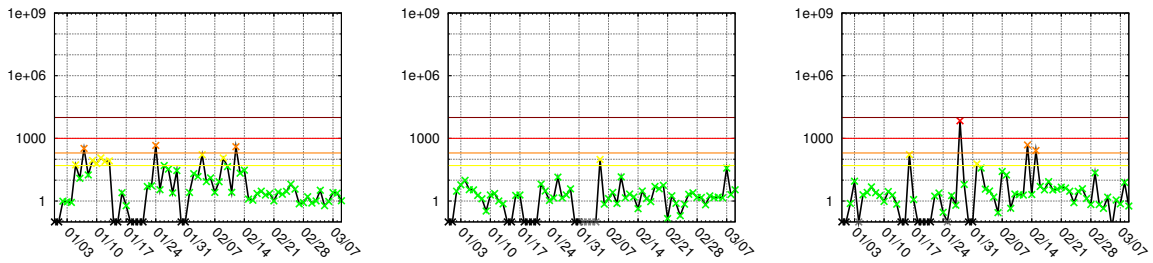
(a) server 163 (b) server 164 (c) server 165

Figure A.55: Anomaly score timeseries for servers 163–165



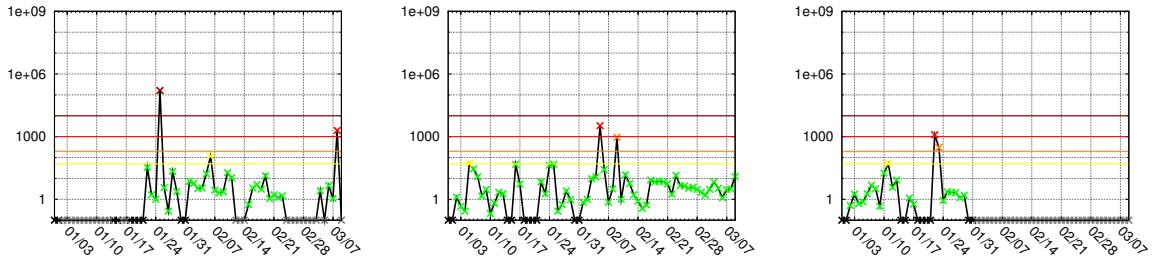
(a) server 166 (b) server 167 (c) server 168

Figure A.56: Anomaly score timeseries for servers 166–168



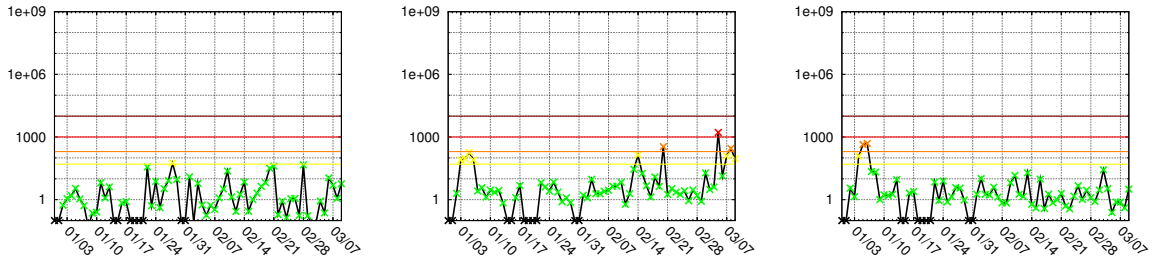
(a) server 169 (b) server 170 (c) server 171

Figure A.57: Anomaly score timeseries for servers 169–171



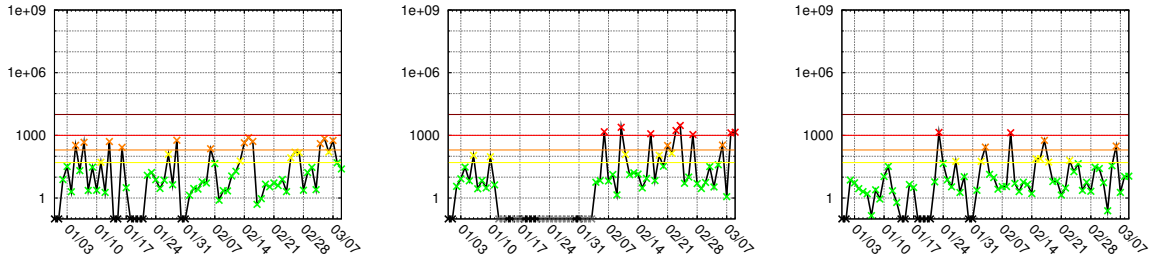
(a) server 172 (b) server 173 (c) server 174

Figure A.58: Anomaly score timeseries for servers 172–174



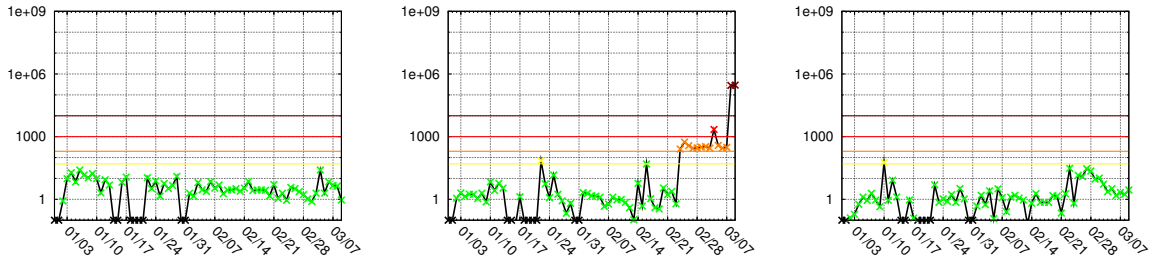
(a) server 175 (b) server 176 (c) server 177

Figure A.59: Anomaly score timeseries for servers 175–177



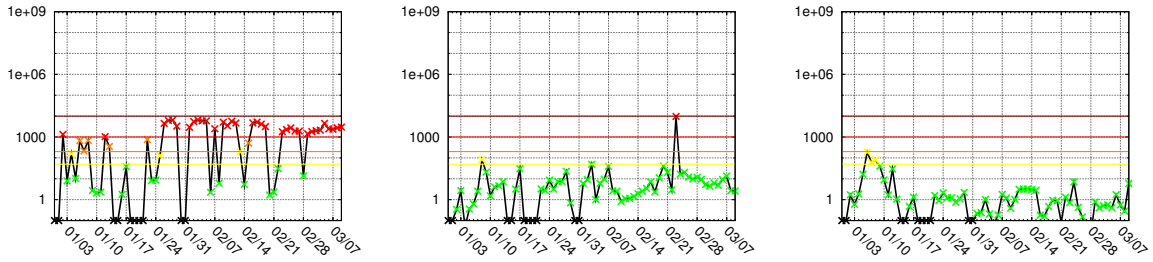
(a) server 178 (b) server 179 (c) server 180

Figure A.60: Anomaly score timeseries for servers 178–180



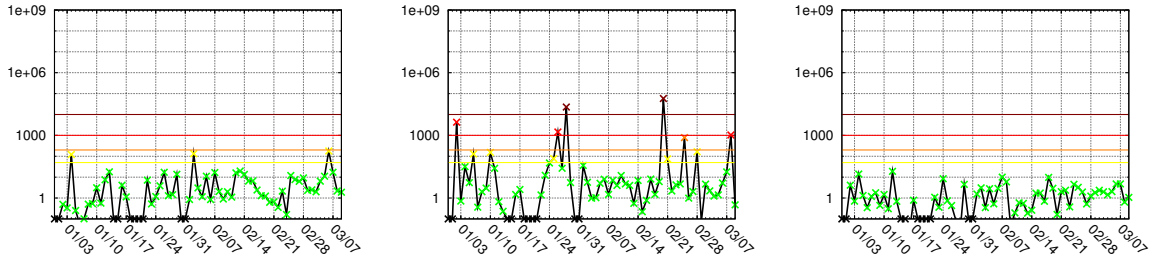
(a) server 181 (b) server 182 (c) server 183

Figure A.61: Anomaly score timeseries for servers 181–183



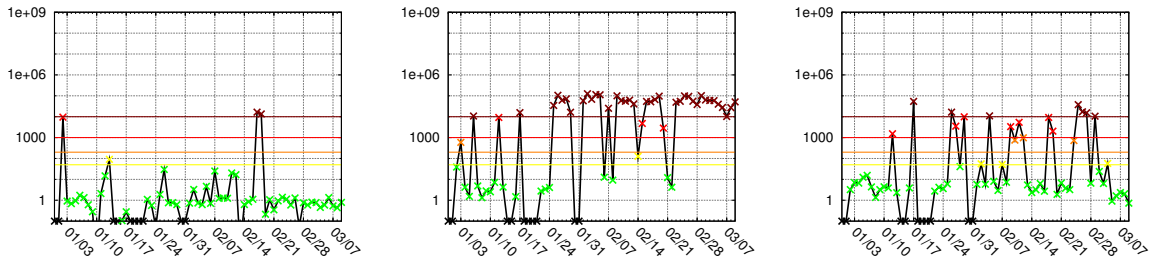
(a) server 184 (b) server 185 (c) server 186

Figure A.62: Anomaly score timeseries for servers 184–186



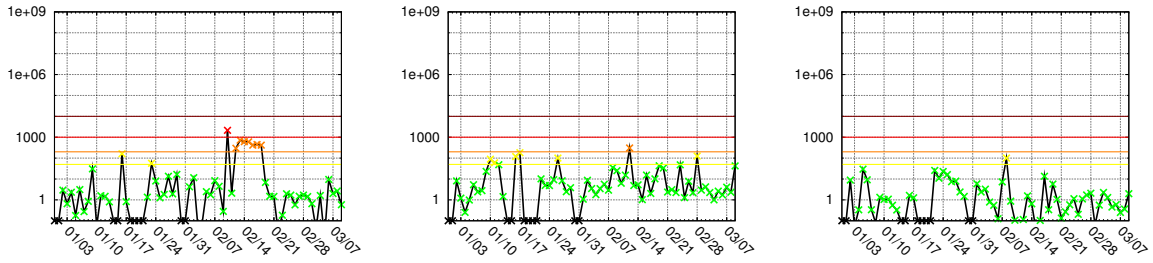
(a) server 187 (b) server 188 (c) server 189

Figure A.63: Anomaly score timeseries for servers 187–189



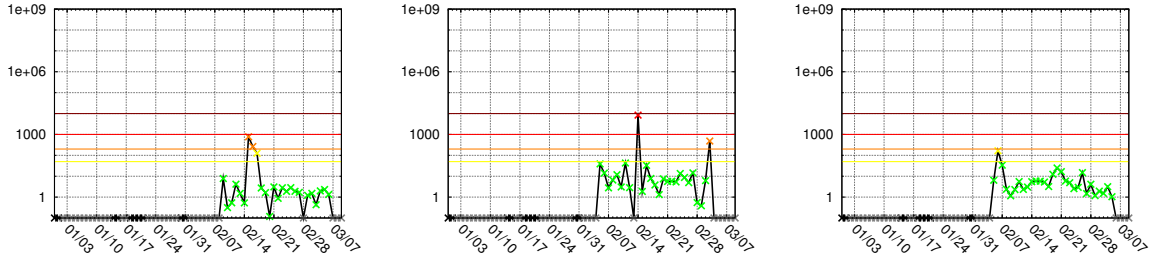
(a) server 190 (b) server 191 (c) server 192

Figure A.64: Anomaly score timeseries for servers 190–192. Server 191 has such strong anomalies because the variation in the QF space is horizontal (*i.e.* left-to-right variation, or variation along the independent axis) instead of vertical, which is a non-linear variation in the QF space. Thus, an anomaly has higher scores than a visual inspection of the distributions would suggest.



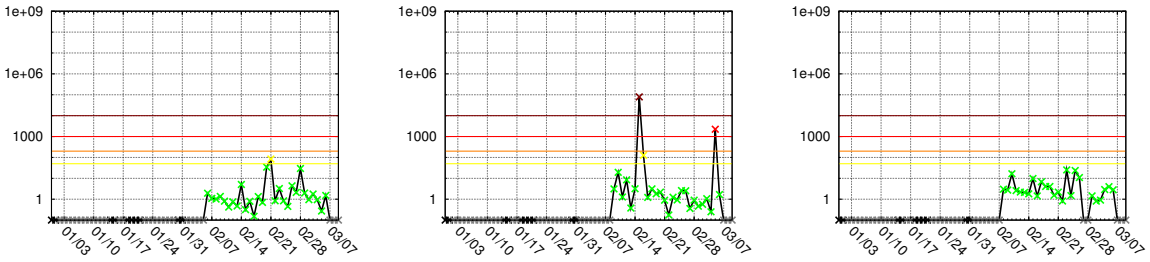
(a) server 193 (b) server 194 (c) server 195

Figure A.65: Anomaly score timeseries for servers 193–195



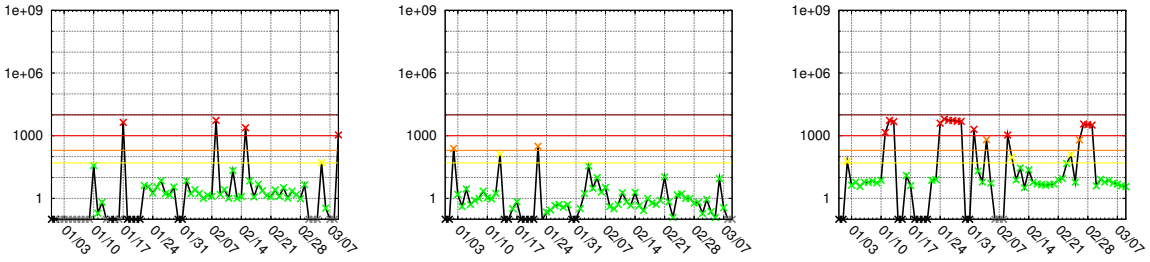
(a) server 196 (b) server 197 (c) server 198

Figure A.66: Anomaly score timeseries for servers 196–198



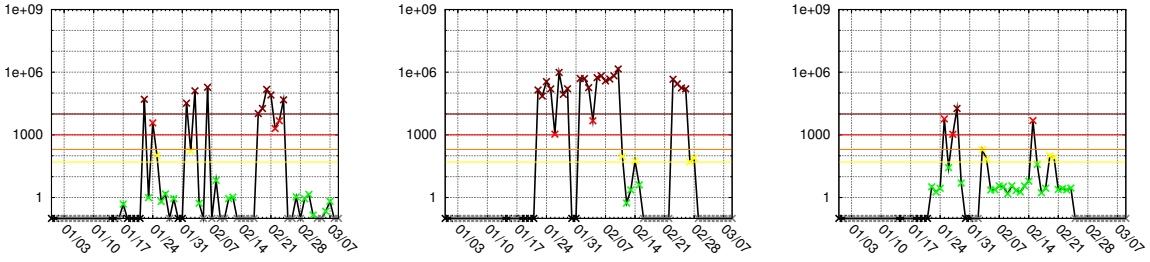
(a) server 199 (b) server 200 (c) server 201

Figure A.67: Anomaly score timeseries for servers 199–201



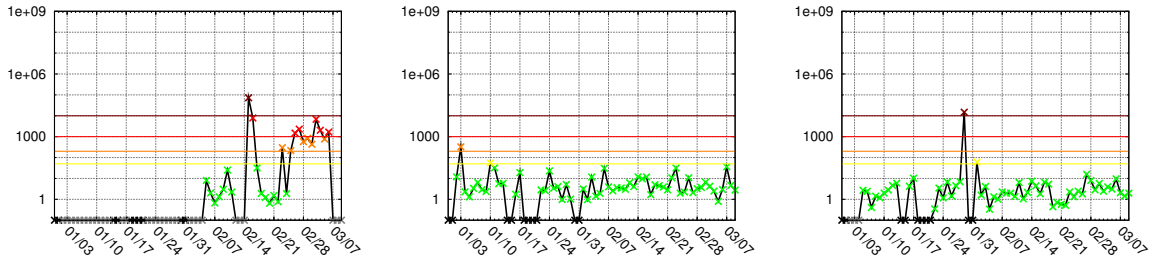
(a) server 202 (b) server 203 (c) server 204

Figure A.68: Anomaly score timeseries for servers 202–204



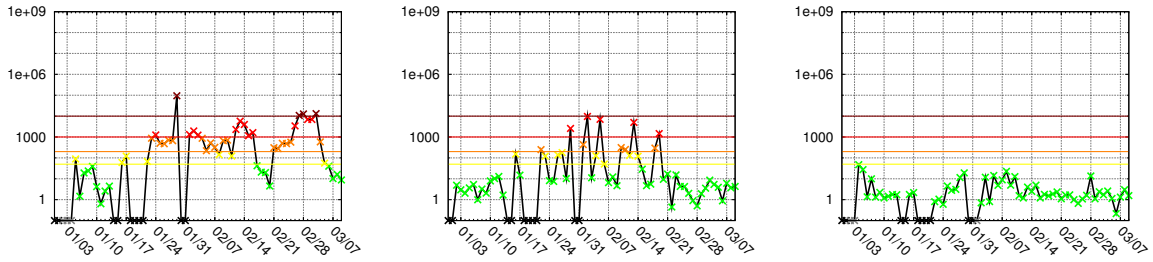
(a) server 205 (b) server 206 (c) server 207

Figure A.69: Anomaly score timeseries for servers 205–207. Servers 205 and 206 are in fact DHCP assigned addresses, as indicated by the presence of many gray points in the anomaly score timeseries.



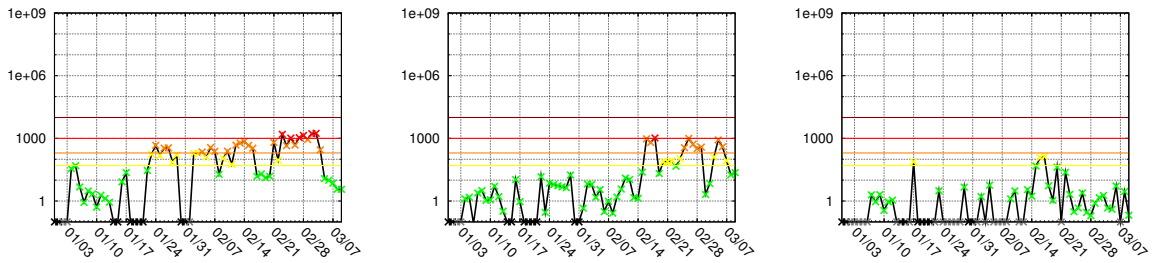
(a) server 208 (b) server 209 (c) server 210

Figure A.70: Anomaly score timeseries for servers 208–210



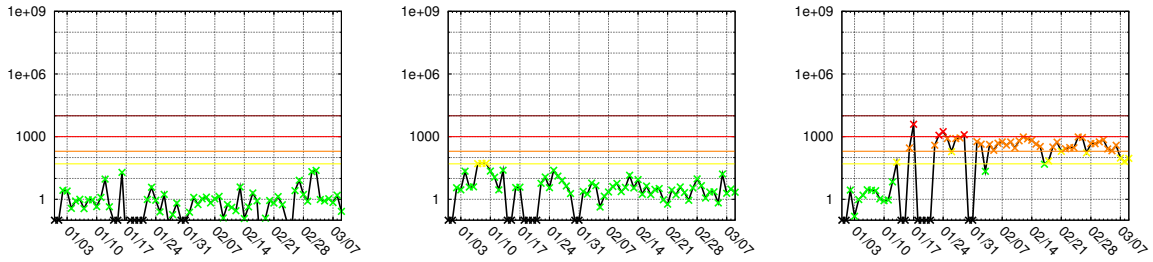
(a) server 211 (b) server 212 (c) server 213

Figure A.71: Anomaly score timeseries for servers 211–213



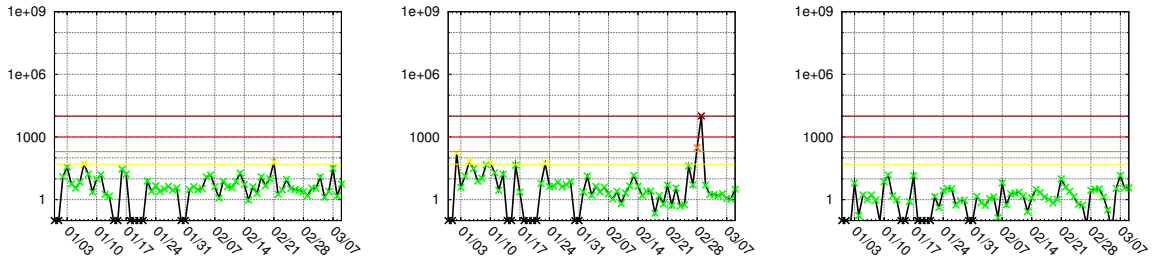
(a) server 214 (b) server 215 (c) server 216

Figure A.72: Anomaly score timeseries for servers 214–216



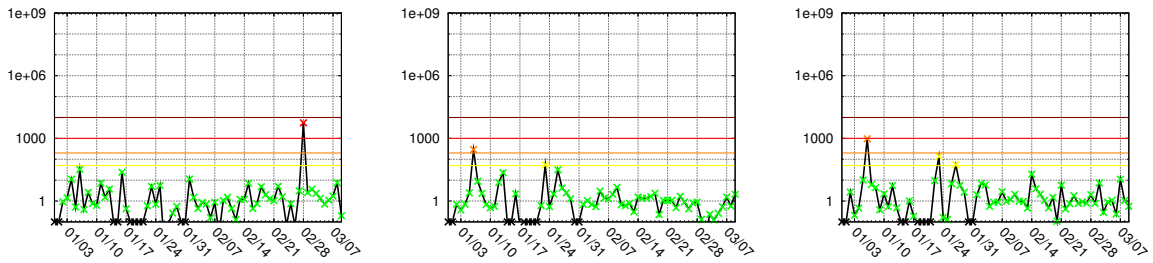
(a) server 217 (b) server 218 (c) server 219

Figure A.73: Anomaly score timeseries for servers 217–219



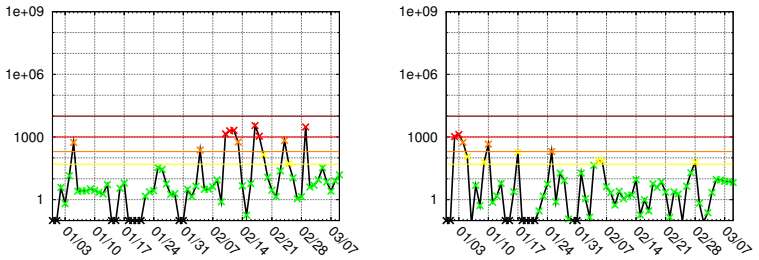
(a) server 220 (b) server 221 (c) server 222

Figure A.74: Anomaly score timeseries for servers 220–222



(a) server 223 (b) server 224 (c) server 225

Figure A.75: Anomaly score timeseries for servers 223–225



(a) server 226 (b) server 227

Figure A.76: Anomaly score timeseries for servers 226–227

Bibliography

[BC98] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, pages 151–160, 1998.

[BCG+07] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 13–24, New York, NY, USA, 2007. ACM Press.

[Bro08] Robert Elijah Broadhurst. *Compact Appearance in Object Populations Using Quantile Function Based Distribution Families*. PhD thesis, Dept. of Computer Science, UNC Chapel Hill, 2008.

[CKK01] Kenjiro Cho, Ryo Kaizaki, and Akira Kato. Aguri: An aggregation-based traffic profiler. In *COST 263: Proceedings of the Second International Workshop on Quality of Future Internet Services*, pages 222–242, London, UK, 2001. Springer-Verlag.

[ESV03] Cristian Estan, Stefan Savage, and George Varghese. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 137–148, New York, NY, USA, 2003. ACM Press.

[FDL+01] Chuck Fraleigh, Christophe Diot, Bryan Lyles, Sue Moon, Philippe Owezarski, Dina Papagiannaki, and Fouad Tobagi. Design and Deployment of a Passive Monitoring Infrastructure. In *Lecture Notes in Computer Science, 2170:556-567*, 2001.

[Fel00] Anja Feldmann. Blt: Bi-layer tracing of http and tcp&slash;ip. *Comput. Netw.*, 33(1-6):321–335, 2000.

[FVCT02] Yun Fu, Amin Vahdat, Ludmila Cherkasova, and Wenting Tang. Ete: Passive end-to-end internet service performance monitoring. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 115–130, Berkeley, CA, USA, 2002. USENIX Association.

[GKMS01] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Quicksand: Quick summary and analysis of network data dimacs technical report, 2001.

[HBP+05] Alefiya Hussain, Genevieve Bartlett, Yuri Pryadkin, John Heidemann, Christos Papadopoulos, and Joseph Bannister. Experiences with a continuous network tracing infrastructure. In *Proc. SIGCOMM MineNet*, pages 185–190, 2005.

[HC06] Félix Hernández-Campos. *Generation and Validation of Empirically-Derived TCP Application Workloads*. PhD thesis, Dept. of Computer Science, UNC Chapel Hill, 2006.

[HCJS07a] Felix Hernández-Campos, Kevin Jeffay, and F. Donelson Smith. Modeling and Generating TCP Application Workloads. In *Proc. IEEE Broadnets*, 2007.

[HCJS07b] Félix Hernández-Campos, Kevin Jeffay, and F.D. Smith. Modeling and Generation of TCP Application Workloads. In *Proc. IEEE Int'l Conf. on Broadband Communications, Networks, and Systems*, 2007.

[LH02] Kun-Chan Lan and John Heidemann. Rapid model parameterization from traffic measurements. *ACM Trans. Model. Comput. Simul.*, 12(3):201–229, July 2002.

[MJ98] Robert G. Malan and Farnam Jahanian. An extensible probe architecture for network protocol performance measurement. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 215–227, New York, NY, USA, 1998. ACM.

[ON06] David Olshefski and Jason Nieh. Understanding the management of client perceived response time. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 240–251, New York, NY, USA, 2006. ACM.

[ONA04] David Olshefski, Jason Nieh, and Dakshi Agrawal. Using certes to infer client response time at the web server. *ACM Trans. Comput. Syst.*, 22(1):49–93, February 2004.

[ONN04] David P. Olshefski, Jason Nieh, and Erich Nahum. ksniffer: determining the remote client perceived response time from live packet streams. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, page 23, Berkeley, CA, USA, 2004. USENIX Association.

[Pax99] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks 31(23-24)*, pages 2435–2463, 1999.

[Plo00] Dave Plonka. Flowscan: A network traffic flow reporting and visualization tool. In *LISA '00: Proceedings of the 14th USENIX conference on System administration*, pages 305–318, Berkeley, CA, USA, 2000. USENIX Association.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,623–656, July, October 1948.

[SHCJ01] F.D. Smith, Félix Hernández-Campos, and Kevin Jeffay. What TCP/IP Protocol Headers Can Tell Us About the Web. In *Proceedings of ACM SIGMETRICS '01*, 2001.

[Udu96] Divakara K. Udupa. In *Network Management System Essentials*. McGraw-Hill, USA, 1996.

[VV06] Kashi V. Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 111–122, New York, NY, USA, 2006. ACM.

[WAHC+06] Michele C. Weigle, P. Adurthi, Félix Hernández-Campos, Kevin Jeffay, and F.D. Smith. Tmix: a tool for generating realistic TCP application workloads in ns-2. In *ACM SIGCOMM CCR*, volume 36, pages 65–76, 2006.