Passive, Streaming Inference of TCP Connection Structure for Network Server Management

Jeff Terrell[1], Kevin Jeffay[1], F. Donelson Smith[1], Jim Gogan[2], and Joni Keller[2]

[1] Department of Computer Science
[2] ITS Communication Technologies
University of North Carolina
Chapel Hill, NC 27599
{jsterrel,jeffay,smithfd}@cs.unc.edu, {gogan,hope}@email.unc.edu

Abstract. We have developed a means of understanding the performance of servers in a network based on a real-time analysis of passively measured network traffic. TCP and IP headers are continuously collected and processed in a streaming fashion to first reveal the application-layer structure of all client/server dialogs ongoing in the network. Next, the representation of these dialogs are further processed to extract performance data such as response times of request-response exchanges for all servers. These data are then compared against archived historical distributions for each server to detect performance anomalies. Once found, these anomalies can be reported to server administrators for investigation.

Our method uncovers nontrivial performance anomalies in arbitrary servers with no instrumentation of the server nor even knowledge of the server's function or configuration. Moreover, the entire process is completely transparent to servers and clients. We present the design of the tools used to perform this analysis, as well as a case study of the use of this method to uncover a significant performance anomaly in a UNC web portal.

1 Introduction

Monitoring the performance of servers in a network is a challenging and potentially expensive problem. Common approaches are to purchase and install monitoring software on the server, or to use an active monitoring system that generates service requests periodically and measures the response time. Both approaches, while effective, typically require extensive customization to work with the specific server/service at hand.

We are developing an alternate approach based on passive collection of packet header traces, and real-time analysis of the data to automatically construct an empirical model of the requests received by servers and the responses generated. These models can be constructed for arbitrary servers with no knowledge of the functions performed by the server or the protocols used by the server. Given these models, we can easily compute important performance measures such as the response times for a server. Using statistical methods originally developed for medical image processing, distributions of these variables can be

M. Papadopouli, P. Owezarski, and A. Pras (Eds.): TMA 2009, LNCS 5537, pp. 42–53, 2009.

compared to archived historical distributions for each server to detect performance anomalies.

The approach works for arbitrary servers because it relies solely on properties of TCP. Using knowledge of the TCP protocol, packet header traces (consisting of only TCP/IP headers and no application layer headers or payloads) are processed in a streaming fashion to construct a structural model of the application dialog between each server and each client in real-time. This model is an abstract representation of the pattern of application-data-unit (ADU) exchanges that a client and server engaged in at the operating system's socket interface. For example, if a web browser made a particular request to a web server that was 200 bytes long, and the server generated a response of 12,000 bytes, then by analyzing the headers of the sequence of TCP/IP packets flowing between the client and server, we would infer this request/response structure and represent this dialog as consisting of a single exchange wherein 200 bytes were sent from client to server and 12,000 bytes were sent from server to client. We intentionally ignore transport-layer effects such as segmentation, retransmission, etc. The model can be augmented to include both server-side and client-side "think times" which can be inferred from the arrival time of the packets. We refer to the server-side think times as *response times*, and they are our primary performance metric.

Using off-the-shelf hardware, we have constructed a network monitoring server that is capable of tracing the 1 Gbps link connecting the 40,000 person UNC campus to its upstream ISP, and performing the above analysis continuously, in real-time, for all servers on the UNC campus. We have been continuously tracing the UNC campus and gathering response time performance data for all servers for a period of over six months. During this period we have processed approximately 70 terabytes of packet headers. However, because our representation of client/server dialogs is relatively compact, the complete activity of the UNC servers during this 6-month period requires only 3 terabytes of storage (600 gigabytes, compressed). By mining these data for performance anomalies, we were able to discover a significant performance anomaly that occurred to a major UNC web portal. Over a period of three days in April 2008, the server experienced a performance issue in which the average response time increased by 1,500%. This discovery was made without any instrumentation of the server or even a priori knowledge of the server's existence.

In this paper we present an overview of our method of capturing and modeling client/server dialogs and its validation. The dialogs are represented using a format we call an *a-b-t connection vector* where a represents a request size, b represents a response size, and t represents a think time. We present the an overview of a tool we have developed called `adudump` that processes TCP/IP packet header traces in real-time to generate *a-b-t* connection vectors for all client/server connections present in the network. We then present some results from an on-going case study of the use of `adudump` to generate connection vectors

for servers on the UNC campus network and the mining of these data to understand server performance. The tools used in this study and the data obtained will be publicly available for non-commercial use.

2 Related Work

Inferring the behavior of applications from analyses of underlying protocols is not new. For example, several schemes for monitoring web systems via an analysis of HTTP messages have been reported. Feldmann's BLT system [1] passively extracts important HTTP information from a TCP stream, but, unlike our approach, BLT is an off-line method that requires multiple processing passes and fundamentally requires information in the TCP payload (i.e., HTTP headers). This approach cannot be used for continuous monitoring or monitoring when traffic is encrypted. In [2] and [3], Olshefski *et al* introduce ksniffer and its improved sibling, RLM, which passively infer application-level response times for HTTP in a streaming fashion. However, both systems require access to HTTP headers, making them unsuitable for encrypted traffic. Furthermore, these approaches are not purely passive. ksniffer requires a kernel module installed on the server system, and RLM places an active processing system in the network path of the server. In contrast, our methods will work for any application-layer protocol and we can monitor a large collection of arbitrary servers simultaneously.

Commercial products that measure and manage the performance of servers include the OPNET ACE system[1]. ACE also monitors response times of network services but requires an extensive deployment of measurement infrastructure throughout the network, on clients, servers, and points in between. Fluke's Visual Performance Manager[2] is similar and also requires extensive configuration and integration. Also similar is Computer Associates Wily Customer Experience Manager[3]. CEM monitors the performance of a particular web server, and in the case of HTTPS, it requires knowledge of server encryption keys in order to function.

3 Measurement

The `adudump` tool generates a model of ADU exchanges for each TCP connection seen in the network. The design of the tool is based on earlier approaches for passive inference of application-level behavior from TCP headers (Smith *et al* [4], Weigle *et al* [5], Hernandez-Campos *et al* [6,7]). However, while these approaches build application-level models from packet headers in an offline manner, we have extended these techniques to enable *online* (real-time) inference (*i.e.* analyzing

[1] http://www.opnet.com/solutions/application_performance/ace.html
[2] http://www.flukenetworks.com/fnet/en-us/products/Visual+Performance+Manager/Overview.htm
[3] http://www.ca.com/us/performance-monitoring.aspx

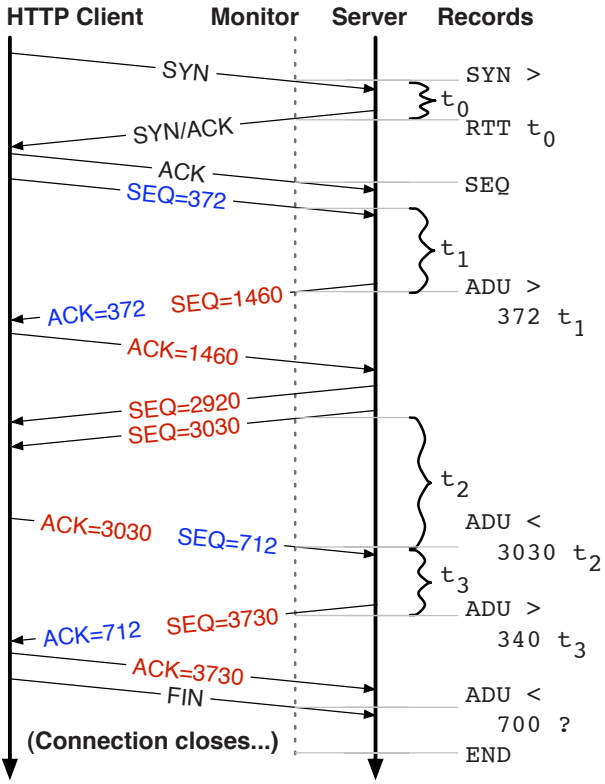


Fig. 1. *a-b-t* inference example

packets as they are seen at a monitor in a single pass). This affords the capability for *continuous* measurement of application-level data.

For a given connection, the core inference method is based on an analysis of TCP sequence numbers. As explained in [4,5], sequence numbers provide enough information to reconstruct the application-level dialogue between two end points. Figure 1 details the inferences that `adudump` draws for an example connection. `adudump` not only reports the size of the ADUs, but the application-level *think-time* between ADUs. A variety of contextual information is also printed, as shown in Table 1. Table 2 also gives an example of the data format.

To understand the inference, consider Figure 1. The connection "begins" when the three-way handshake completes. This event is marked with a `SEQ` record. The monitor sees a data segment sent from the client (in this case a web browser) to the server and makes a note of the time it was sent. The next segment is another data segment, sent in the opposite direction and acknowledging the previous data. Thus, `adudump` infers that the previous ADU (of 372 bytes) is completed, and generates a record with the ADU's size, direction, and subsequent think-time. The next segment, a pure acknowledgement (i.e. a segment without a

Table 1. 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
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

Table 2. `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
```

payload) is not used in determining ADU boundaries. In general, `adudump` ignores pure acks. Next, the server continues its response. Again, note that `adudump` generates no record until it infers that the ADU is complete. Also, note that the think-times that `adudump` reports are relative to the position of the monitor in the network. In other words, the think-times necessarily include a component of network delay as well. This is discussed in more detail in Section 4.

Note that this simple example assumes that the client and server take turns sending data. Such cases are called "sequential connections." "Concurrent connections," wherein both endpoints transmit simultaneously, can also be analyzed. Examples of such applications that employ concurrent connections include HTTP with pipelining enabled, peer-to-peer applications such as BitTorrent, and most interactive applications such as the secure shell. Connections are assumed to be sequential (i.e. non-concurrent) by default, until concurrency is detected by the existence of unacknowledged data in both directions simultaneously. Although concurrent applications do have a think-time, it is not possible to unambiguously determine the think-time without instrumenting the application. In our data,

ADUs from concurrent connections constitute approximately 5% of the connections, 25% of the ADUs seen, and 30% of the size in bytes.

4 Data

We have used `adudump` to generate a six-month data set of records of all TCP connections entering the UNC campus from the Internet, which we will make available through DatCat[4]. It is this data set that we will use for the remainder of this paper. Overall, we collected over three terabytes of data, modeling about 4 billion connections. Table 3 lists the individual collections, which were punctuated by measurement faults such as power outages and full disks. The records were captured by running `adudump` on a fiber split of the 1 Gbps link between the University of North Carolina and its commodity Internet uplink. Both directions of the link were tapped and fed to a machine with a 1.8 GHz Intel Xeon processor, 1.25 GB of RAM, and an Endace DAG card for packet capture. For privacy reasons, only inbound connections (*i.e.* those for which the initial SYN was sent to the UNC network) were captured. The collection process experienced very infrequent bouts of packet drops; the relatively old machine was able to keep up even when the link burst to its full 1 Gbps capacity.

Table 3. Data collection. All times local (EDT); all dates 2008. Data for Monday, March 17, was lost. Days are in MM/DD format. Durations are listed as days:hours:minutes.

#	begin	end	duration	outage	size	records	ADUs	conns
1	Fr 03/14 22:25	Th 04/17 03:50	33:05:25	1:14:11	813 GB	11.8 B	8.8 B	820 M
2	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
3	We 04/23 11:14	Th 04/24 03:00	0:15:46	0:07:38	16 GB	234 M	161 M	19 M
4	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
5	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
6	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
7	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
8	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
9	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
10	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

Think-times reported by `adudump` are with respect to the monitor's vantage point, and think-times include an unknown component of network delay. However, note that since our monitor is at the edge of the UNC network, it is relatively close to the UNC servers. Since the UNC network is generally well-provisioned and well-designed, it is rare to see a intra-campus round-trip-time of more than a millisecond. For this reason, we only consider server-side think-times for the analysis, as these can be accurately inferred from our monitoring vantage point.

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

5 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, we instead constructed 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 runtime 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-read/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. We captured the packets traversing the link, saving the trace as a pcap file which we then fed as input to `adudump`, producing the *measured* data.[5] In this way, we can determine how correctly `adudump` functions.

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

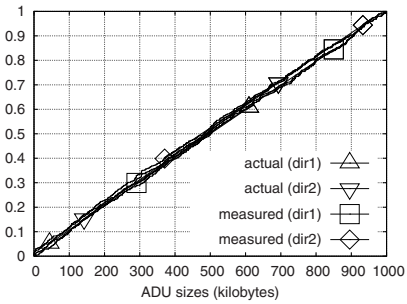
5.1 Sequential Validation

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 we use plausible randomized network path conditions to test `adudump`, but we do not claim realism.

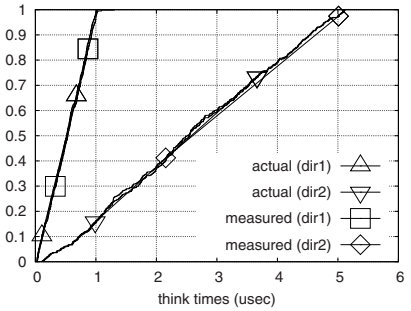
Figure 2(a) plots the actual and measured per-direction ADU size distributions. The distributions are nearly identical. The slight differences are because, in the case of TCP retransmission timeouts (with sufficiently long RTT), `adudump` splits the ADUs, guessing (incorrectly in this case) that the application intended them to be distinct. The default *quiet time threshold*, which governs this behavior, is 500 milliseconds, so RTTs shorter than this threshold do not split the ADU. We chose 500 ms as a reasonable trade-off between structural detail and solid inference of application intent.

Similarly, Figure 2(b) plots the actual and measured think-time distributions. Note that the actual distributions were different for each direction. Note also

[5] `adudump`, which uses CAIDA's CoralReef library, works equally well analyzing offline traces.

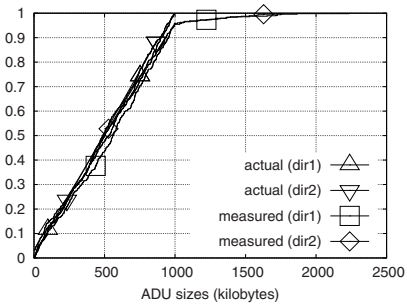


(a) CDF of actual vs. measured ADU size distributions, for either direction.

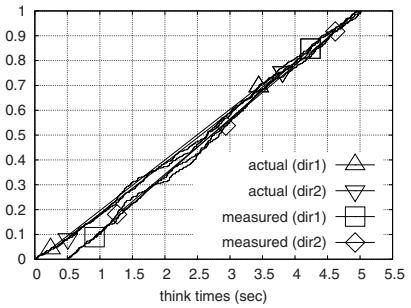


(b) CDF of actual vs. measured think-time distributions, for either direction.

Fig. 2. 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.

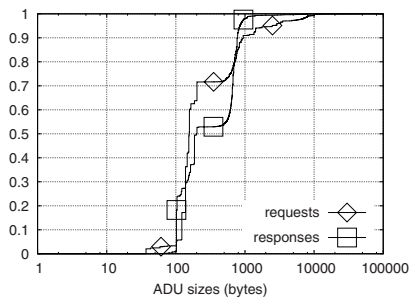
Fig. 3. Concurrent validation results

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 very close to the ground truth.

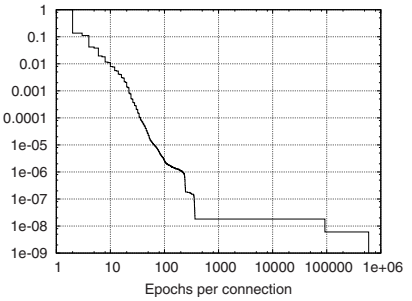
5.2 Concurrent Validation

In the concurrent tests, each application instance sends multiple ADUs with interspersed think times without synchronizing on the ADUs they receive. We did not introduce packet loss in this case.

Figure 3(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



(a) CDF of ADU sizes



(b) Complementary CDF of exchanges per connection

Fig. 4. Connection structure information inferred by `adudump` for example server

the two ADUs into one that is larger. This is a type of inference error that is unavoidable because we do not have access to applications running on end systems. Nevertheless, this is a sacrifice we gladly make, because it enables generic access to the behavior of any server without any server instrumentation.

Figure 3(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.

6 Example Use

The UNC dataset contains records for every server on campus that communicated with a client on the Internet. To demonstrate the usefulness of the data generated by `adudump`, we examine the data pertaining to one such server, the UNC webmail server. We selected this server more-or-less randomly from among many popular UNC servers, yet this example shows the breadth and depth of information available for any servers. Note that the information we present here only scratches the surface of what is available for the webmail server (let alone all UNC servers) and is presented merely to provide an example of the types of analyses that are enabled by `adudump` data.

We start by looking at the broad picture offered by our 6-month dataset. Figure 4(a) shows distributions of request sizes received by the webmail server and response sizes sent by the server. The requests in particular exhibit strong modality, with most of the distribution found in relatively few values. Because we know the identity and purpose of this server, we can conjecture that the smooth increase in response size between 500 and 1,000 bytes is because the size of email messages vary smoothly in that range. We can also make educated guesses about the behavior of these connections, given the (externally known) fact that we are dealing with a HTTPS server. However, without additional information (or system administrator knowledge), we cannot know for certain whether this is the cause. This weakness, however, is also a strength: the bluntness of the

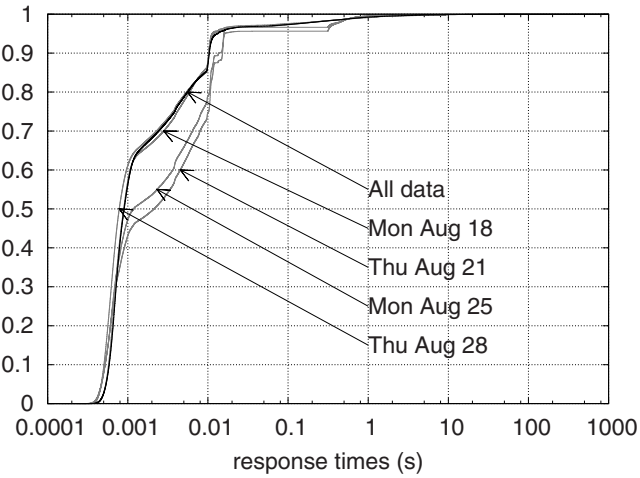


Fig. 5. CDF of response times for example server, both for the entire dataset and during selected days at the start of the semester

inferences that `adudump` makes also enables it to be more broadly applicable to *any* server operating over TCP.

Another structure-revealing metric is the number of request-response exchanges per connection. Figure 4(b) shows the distribution of exchanges per connection over the entire trace. 86% of connections have exactly two exchanges, 96% have four or fewer, and the distribution exhibits classic heavy-tailed behavior. This plot clearly suggests many avenues for additional analyses (which we are pursuing). Our point in this paper is that `adudump` provides insight into interesting and important application-level behaviors without requiring any knowledge of what the application is or how it performs.

We also want to briefly explore the depth of data reported by `adudump`. Figure 5 compares the overall distribution of webmail response times gathered over the whole dataset with specific days. In general, this distribution is very stable from day to day, differing little even on weekends and holidays. However, we discovered a significant change during the beginning of the fall 2008 semester. Monday, August 18^{th} is the day before the start of the semester, and webmail response times for this day are typical. Over the next several days, however, the server takes longer to respond. Normal operation resumes by Thursday the 28^{th}. Although beyond the scope of this paper, we note that it is easy to drill down, looking at response time distributions per hour, differences in request size or response size distributions, or even a timeseries of the individual response time measurements.

7 Case Study

One challenge we face is in detecting when performance anomalies occur, given the quantitative and qualitative variation of response time distributions among

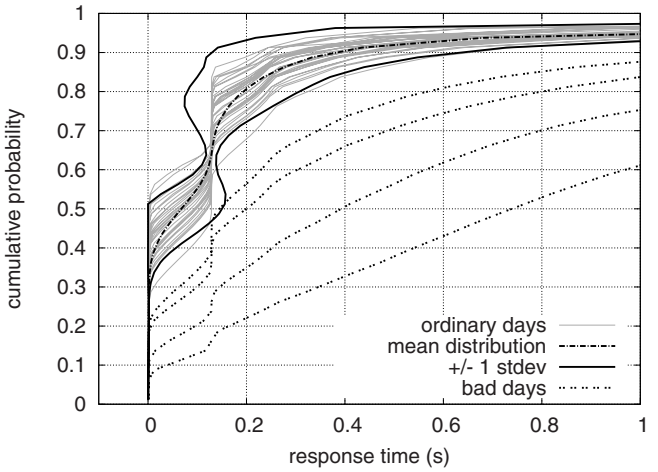


Fig. 6. Illustration of performance anomaly detection using data from a campus web server

servers. First, we note that, for our purposes, it makes little sense to compare response time distributions for different servers. Even servers of the same type (e.g. HTTP) will often have substantially different response time distributions. Thus, we must compare a server's current operation (e.g., the distribution over the past hour or past day) to the same server's historical operation. The problem of performance anomaly detection reduces to determining the likelihood of the current distribution, given the historical distribution (both of which are empirical, or non-parametric).

Figure 6 illustrates this process using two sets of distributions. The lighter lines are CDFs of the response time distribution for a UNC web server for an "ordinary" day. We refer to these as the "training" set. The four dashed lines are response time distributions for days that were flagged by server administrators as corresponding to days when performance problems were noted. In addition, there is a line representing the mean distribution of the training set as well as two lines representing the mean plus or minus one standard deviation.

The standard deviations were calculated using a method introduced in [8]. Each distribution was represented as a 40-bin quantile function. A quantile function can be thought of as a summarized inverse CDF. The distribution is first evenly divided by quantile into 40 bins, so that, for example, the second bin contains all values between the 2.5 and 5^{th} percentiles. Each bin is then summarized as a mean. The result is a vector of length 40, which can be thought of as a point in 40-dimensional space. Principal components analysis (PCA) was then performed on the 40-dimensional "cloud" of points to determine the two (orthogonal) directions of greatest variation, as well as the standard deviation along these axes. Adding and subtracting these principal components (scaled by their respective standard deviations) from the mean gives us an idea of the "spread"

of the overall population of distributions. The resulting sum (and difference) give us the 1-standard-deviation "bounds", as shown in Figure 6. The days marked as "bad" fall well outside of the bounds, and thus are clearly anomalous. This provides evidence that anomalous response times can be automatically detected given a historical archive or response time distributions.

8 Conclusion

We have developed and validated a tool to passively infer the application-level dialog in a TCP connection, for all connections on a link, in a passive, online, streaming fashion, at gigabit speeds, on off-the-shelf hardware. Having acquired a multi-month dataset of all TCP connections entering the UNC campus, we have shown that it is possible to identify server response time performance anomalies without knowledge of the function or operation of the server. We believe our tools and methods enable a new paradigm of passive network and server management wherein high-level application performance data can be gleaned from low-level network measurements.

References

1. Feldmann, A.: BLT: Bi-Layer Tracing of HTTP and TCP/IP. In: Proc. of WWW-9 (2000)
2. Olshefski, D.P., Nieh, J., Nahum, E.: ksniffer: determining the remote client perceived response time from live packet streams. In: Proc. OSDI, pp. 333–346 (2004)
3. Olshefski, D., Nieh, J.: Understanding the management of client perceived response time. In: ACM SIGMETRICS Performance Evaluation Review, pp. 240–251 (2006)
4. Smith, F., Hernández-Campos, F., Jeffay, K.: What TCP/IP Protocol Headers Can Tell Us About the Web. In: Proceedings of ACM SIGMETRICS 2001 (2001)
5. Weigle, M.C., Adurthi, P., Hernández-Campos, F., Jeffay, K., Smith, F.: Tmix: a tool for generating realistic TCP application workloads in ns-2. ACM SIGCOMM CCR 36(3), 65–76 (2006)
6. Hernández-Campos, F.: Generation and Validation of Empirically-Derived TCP Application Workloads. Ph.D. dissertation, Dept. of Computer Science, UNC Chapel Hill (2006)
7. Hernández-Campos, F., Jeffay, K., Smith, F.: Modeling and Generation of TCP Application Workloads. In: Proc. IEEE Int'l Conf. on Broadband Communications, Networks, and Systems (September 2007)
8. Broadhurst, R.E.: Compact Appearance in Object Populations Using Quantile Function Based Distribution Families. Ph.D. dissertation, Dept. of Computer Science, UNC Chapel Hill (2008)