

© 2008 Shuyi Chen

LINK GRADIENTS: PREDICTING THE IMPACT OF LINK LATENCY ON
MULTI-TIER APPLICATIONS

BY

SHUYI CHEN

B.S., Peking University, 2006

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Adviser:

Professor William H. Sanders

ABSTRACT

Geographically dispersed deployments of large and complex multitier enterprise applications introduce many challenges, including those involved in predicting the impact of network latency on end-to-end transaction response times. Here, a measurement-based approach to quantifying this impact using a new metric called the *link gradient* is presented. A nonintrusive technique for measuring the link gradient in running systems using delay injection and spectral analysis is presented, along with experimental results on PlanetLab that demonstrate that the link gradient can be used to predict end-to-end responsiveness, even in new and untested application configurations.

ACKNOWLEDGMENTS

I thank my girlfriend Lide Zhang for her support, her love. I could not have done it without her.

I thank my family for their love, support, and encouragement. Thanks to my friends for their friendship.

I thank Kaustubh Joshi, Matti Hiltunen and Richard Schlichting. They gave me useful advice on my thesis work.

I thank Jenny Applequist. She helped me proofread my thesis.

This work was completed with invaluable help from my adviser, Professor William H. Sanders. I am grateful for his advice, pointers, discussions, and teaching.

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-04-06351 and CNS-06-15372. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

CHAPTER 1	Introduction	1
CHAPTER 2	Technical Approach	5
2.1	The Link Gradient	5
2.2	Link Latency Perturbation	8
2.3	Spectral Link Gradient Computation	12
CHAPTER 3	Architecture and Implementation	14
3.1	Monitoring Framework	14
3.1.1	The Sniffer Daemon	15
3.1.2	The Delay Daemon	15
3.1.3	The Central Coordinator	19
3.2	Measurement Algorithm	20
3.2.1	Training Phase	20
3.2.2	Measurement Phase	23
CHAPTER 4	Micro Benchmarks	24
4.1	Accuracy of Delay Injection	25
4.2	Sampling Points per Period	27
4.3	Total Number of Sampling Points	29
4.4	Data Points per Bin	31
4.5	Delay Scale Factor	32
4.6	Standard Deviation of Response Time	33
4.7	Simple Application Scenarios	34
CHAPTER 5	Evaluation: RUBiS on PlanetLab	39
5.1	Experimental Setup	39
5.2	Link Gradient Computation	40
5.3	Predictive Power	42
5.4	Communication Pattern Variations	46
5.5	Per-Transaction Optimization	49
5.6	Optimization for Multiple Applications	50
CHAPTER 6	Limitations	53

CHAPTER 7	Related Work	55
CHAPTER 8	Conclusion and Future Work	57
REFERENCES	58

CHAPTER 1

Introduction

A global network infrastructure, coupled with the emergence of utility computing services such as cloud computing and new software paradigms such as service-oriented architectures (SOAs) and mashups, has led to applications that are geographically distributed across multiple data centers around the world. While offering many benefits, such distributed applications introduce significant challenges, not the least of which is the management of end-to-end responsiveness in a highly dynamic environment. In fact, even quantifying the effects of changing network characteristics on end-to-end performance is not easy. For a given deployment of a multitier enterprise application, for example, this relationship depends both on application factors such as execution logic, workloads, configuration settings, and transaction types, and on network factors such as latency, bandwidth, and loss rate.

Here, we focus on the impact of *logical link latency*—that is, the network latency from one application component to another, possibly across multiple physical links—on the end-to-end performance of application transactions in multitier applications. While this link latency, as noted above, is not the only factor, it is a significant element in many kinds of distributed applications, especially those deployed across wide geographical areas as is often the case with enterprise applications or cross-enterprise SOAs [1, 2]. Our specific goal is to be able to predict accurately the impact on transaction performance of changes in logical link latencies, such as might occur when a distributed application is reconfigured (e.g., a component is moved).

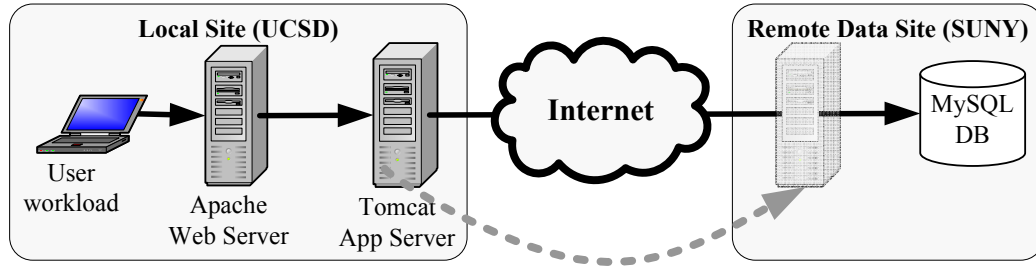


Figure 1.1: Sample RUBiS configuration

To illustrate the difficulty of this task, consider a 3-tier deployment of the RUBiS web service benchmark [3] on PlanetLab, as shown in figure 1.1. This configuration reflects a simple scenario in which a company might keep its back-end database server at its local site (in this case, New York), but deploy geographically dispersed front-ends (including some in California) for load balancing and improved latency to the end users. Table 1.1 shows the impact on transaction end-to-end response time associated with moving the application server from being colocated with the web server (*local*) to being colocated with the database (*remote*). As can be seen, even on this simplest of configurations, the impact of this change in link latency is significant, and varies dramatically based on the transaction type. Not surprisingly, predicting the impact is even more difficult for more complicated and realistic multitier enterprise applications that may have, for example, replicated web, application, and authentication servers and databases, as well as load balancers, caches, firewalls, and traffic filters.

Transaction	Local (s)	Remote (s)	Change
PutBidAuth	0.03	0.17	466%
SearchItemsByReg.	5.70	0.16	-97%
SellItemForm	0.04	0.14	250%
ViewBidHistory	2.31	0.19	-92%
ViewUserInfo	4.13	0.20	-95%

Table 1.1: Average response time for RUBiS transactions

In this thesis, we present a measurement-based approach to quantifying the impact of changes in logical link latency on the per-transaction end-to-end response time for dis-

tributed applications, with a special focus on multitier enterprise applications. This approach is based on the *link gradient*, a new metric that captures this impact. The collection of link gradients for an application, one per link per transaction type, can then be used to predict end-to-end responsiveness even in new and untested configurations. We demonstrate the accuracy of these predictions experimentally on PlanetLab using RUBiS and the bulletin-board benchmark RUBBoS [4].

In addition to the metric itself, we also describe an innovative technique for measuring the link gradient in a running system based on delay injection and spectral analysis. This approach has the following advantages:

- The use of spectral techniques ensures that intrusiveness is kept to a minimum, thereby allowing link gradient measurements to be performed continuously, even on running production systems.
- The link gradient for each transaction can be isolated from other transactions and applications even if they use the same hardware resources, making it particularly useful in highly shared virtual machine environments.
- It does not require information about system internals, makes no assumptions about system architecture (e.g., synchronous vs. asynchronous communication or caching policies), does not require modification of the target system, and does not need synchronized clocks.
- The metric is simple, is compactly expressed, and can easily be used in conjunction with other system metrics.

Knowing the link gradients of a given application provides a cornerstone that can be used in a variety of ways to configure and manage the system. For example, it can be used by designers to ascertain server placements that improve availability while ensuring that responsiveness remains within a tolerable threshold, in SOAs to help in a choice between

competing services, or by network operators to evaluate the cost versus benefit trade-offs of planned network upgrades.

The rest of this thesis is organized as follows. Chapter 2 defines the link gradient metric and develops our proposed spectral measurement technique. Chapter 3 describes the architecture and implementation of our approach. Chapter 4 demonstrates the accuracy of the delay injection mechanisms and provides micro-benchmarks that explore the sensitivity of the measurement technique to various input parameters. Chapter 5 provides experimental results indicating the accuracy, usefulness, and predictive power of the measured link gradients using an example 3-tier e-commerce system running on PlanetLab. Chapter 6 discusses the strengths and limitations of the approach. Chapter 7 provides an overview of related work. Finally, we conclude the paper and outline future work in chapter 8.

CHAPTER 2

Technical Approach

In this chapter, we define the link gradient of a system, and describe how it can be used to predict the effect of link latency changes on a transaction’s response time. Then, we explain how spectral analysis can allow the computation of a link gradient with only small perturbations to the system and derive the required formulae.

2.1 The Link Gradient

Consider a multitier application consisting of multiple software components and let $C = \{c_1, c_2, \dots, c_n\}$ be the set of n logical one-way communication links between them¹. Each link c_i is parameterized by a link latency l_i , which in most cases is the same for the forward and backward links and can be measured by a ping test between the two components. Finally, let the system’s performance metric be quantified using its end-to-end mean response time \bar{rt} . If the system provides a number of different services (e.g., an e-commerce site with multiple transactions, such as browse and buy), then the response time can be either a per-transaction response time, or the system’s overall mean response time².

Then, the *link gradient* $\vec{\nabla} \bar{rt}$ quantifies how a change in the link latency for each link affects the response time, and is defined as a vector $\vec{\nabla} \bar{rt} = \left(\frac{\delta \bar{rt}}{\delta l_1}, \dots, \frac{\delta \bar{rt}}{\delta l_n} \right)$, where each element $\nabla \bar{rt}_i = \frac{\delta \bar{rt}}{\delta l_i}$ is the link gradient of link c_i . Intuitively, the link gradient of a link is a partial derivative that specifies the rate at which the system’s response time changes per

¹Two-way communications are represented by two independent links, one for each direction.

²Although we focus on end-to-end response time, other system metrics whose dependency on link latency is to be quantified can also be used without much modification to the algorithm.

unit change in the link latency of communication link c_i , assuming that the latencies of all other links remain constant.

The link gradient can be used to approximate how the response time of the system would be affected by a change in link latencies (e.g., due to reconfiguration of the system). Specifically, if the vector $\vec{\Delta}l = \{\Delta l_1, \dots, \Delta l_n\}$ represents the amounts by which each link latency changes, the new response time of the system can be approximated by its Taylor expansion:

$$\bar{rt}(\vec{l} + \vec{\Delta}l) \approx \bar{rt}(\vec{l}) + \vec{\nabla} \bar{rt} \cdot \vec{\Delta}l + O(\|\vec{\Delta}l\|^2) \quad (2.1)$$

This equation makes two simplifying assumptions. First, it assumes that the response time (as a function of link latency) does not have any discontinuities. Second, the equation only captures first-order effects. If a system's response time has nonlinear dependencies on the link latency, the equation is accurate only as long as the change in latency Δl is small and the higher-order terms can be ignored. For linear relationships, the higher-order terms vanish, making the equation exact. Although there are some exceptions, we argue that many systems have large regions of continuous linear relationships in which the higher-order effects can be ignored.

To make the argument, we examine an approximate interpretation of link gradients in terms of message crossings. Consider a node a that calls another node b over a link c that has a latency of l . The amount by which the mean response time of node a would change if the link latency changes to $l + \Delta l$ is dictated to a large extent by the type of communication. For instance, if a sends a message to b and waits for a reply before continuing, the response time could be expected to increase by Δl , and the link gradient would be $\lim_{\Delta l \rightarrow 0} \frac{\bar{rt}(l+\Delta l) - \bar{rt}(l)}{\Delta l} = 1$. The link gradient would remain the same if a sent m messages to b in a pipelined fashion (e.g., over a TCP link) before waiting for a response. However, if a were to send m messages in series such that it waited for a response from b before sending the next message, the increased latency would affect the response time for each

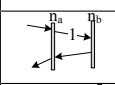
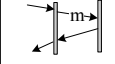
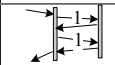
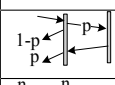
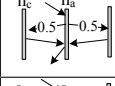
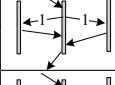
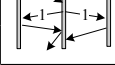
Pattern	Descriptions	Link Gradient
	Single request-response	1
	m pipelined messages	1
	m sequential messages	m
	Cached (miss prob. p)	p
	Symm. Load Balanced	0.5
	Concurrent (wait for all)	Prob[Server is slowest]
	Concurrent (wait for first)	Prob[Server is fastest]

Table 2.1: Link Gradient for Common Message Patterns

of the m messages, and the link gradient would be m . Conversely, if a did not wait for a reply from b , increase in link latency would not affect the response time at all, and the link gradient would be 0. Drawing upon these observations, we can loosely interpret the link gradient as the “mean number of message crossings in the critical path of the system response,” which, for many communication patterns is a constant function of node behavior, and thus gives rise to a linear relationship between response time and latency. However, in reality, factors such as timeouts and nonlinearity due to queuing affect this argument when very large latency changes are considered. We outline ways to tackle nonlinearity in more detail in chapter 6, but nevertheless show in chapter 5 that the linearity assumption holds quite well in practice.

Table 2.1 shows common communication patterns and corresponding (ideal) link gradients. For the first five communication patterns in the table, the mean number of message crossings of a link in the critical path is independent of the actual latency of the link as long as the behavior of n_a does not change (e.g., due to a timeout). Barring that possibility, the dependency between response time and link latency is linear, and the higher-order terms in Equation 2.1 can be ignored. The last two patterns (which are commonly used in systems

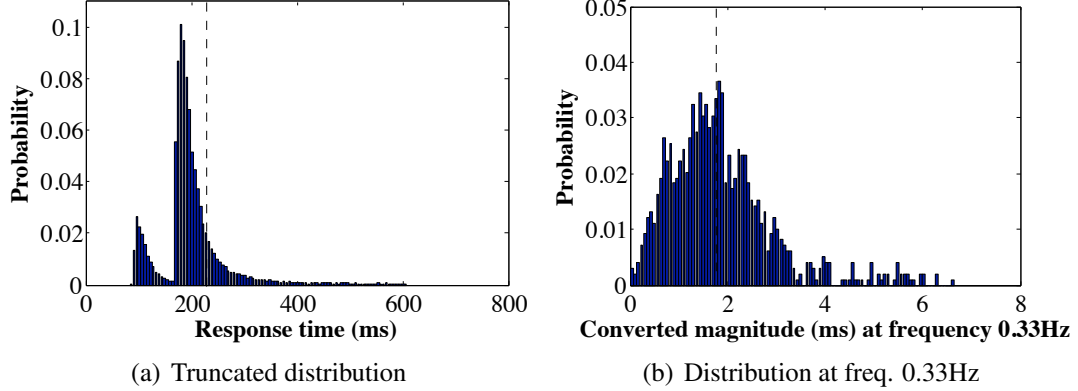


Figure 2.1: Response Time Distributions

with replicated state) represent concurrent communications in which n_a waits for either all responses, or the first one. For those patterns the link gradient is equal to the probability that n_b is the slowest or fastest server respectively, and is thus in the critical path of the system response. Since these probabilities can vary as the link latency changes (and the server in question appears faster or slower in relation to the other servers), the higher-order terms in Equation 2.1 can only be ignored if the change in link latency is very small.

2.2 Link Latency Perturbation

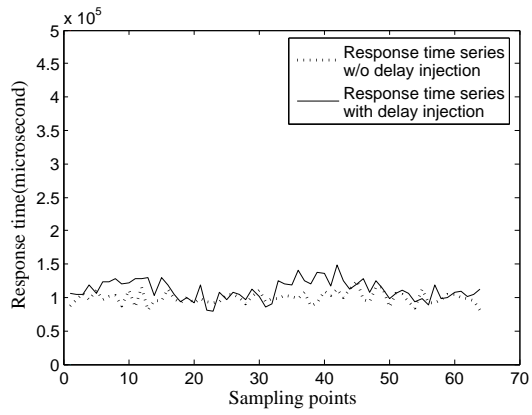
Conceptually, link gradient can be easily computed at run-time by active delay injection. A target communication link can be chosen, and a delay Δl_i can be systematically injected in the packets traversing the link. The end-to-end response time of the system $\bar{r}t'$ can then be measured while the delay is being injected and compared with the nominal response time $\bar{r}t$. The ratio $\frac{\bar{r}t' - \bar{r}t}{\Delta l}$ is then an approximation to the link gradient $\vec{\nabla} \bar{r}t_i$. As long as the relationship between response time and link latency is linear, the approximation is exact.

However, in practice, the complicating factor is that production systems, especially those running across wide-area networks and/or on shared resources, typically have response time distributions with long tails, high variances, non-stationary noise patterns due to periodic events such as garbage collection. For example, figure 2.1(a) shows a trun-

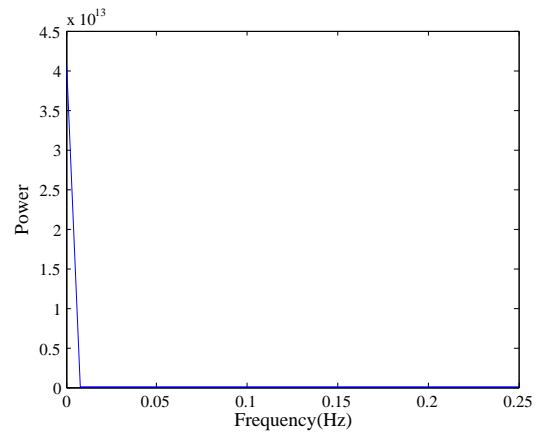
cated response time distribution for a single transaction of RUBiS (*PutBid*) running on PlanetLab. The distribution is constructed using 27648 samples, with a mean of 229 ms, a standard deviation of 269 ms, and has a long tail (not shown) reaching up to 12s. Even with these many samples, the 95% confidence interval for the mean is ± 3.17 ms, implying that to detect a change in the mean with 10% error, the mean would have to change by at-least 63.4 ms. Injecting very high delays into a running system is not only disruptive, but can also cause the system dynamics themselves to change, making the measurement inaccurate. Due to the large number of samples, one runs the risk of the system changing behavior during sample collection itself. Moreover, it is not trivial to inject low variance delays into communication links.

To solve this problem of excessive noise, we propose a unique approach based on the observation that most of the noise found in typical environments is not periodic. Moreover, if periodic noise does exist (e.g., garbage collection), it occurs only at a few narrow frequencies. Therefore, by injecting perturbations in the form of periodic waveforms at carefully chosen frequencies and performing spectral analysis to extract the effect on the system's response time, one can get precision that is significantly superior to a time domain method. To illustrate, consider figure 2.1(b) which shows a distribution of the magnitude of a randomly chosen frequency component (0.33Hz) in the same response time data as figure 2.1(a). However, at this frequency, not only is the mean response time much smaller (1.86 ms), but the distribution is much tighter (with no truncated tail), more symmetric, and has a standard deviation of 1.1 ms. The corresponding 95% confidence interval of ± 0.07 ms allows a change of 1.4 ms to be detected with 10% error.

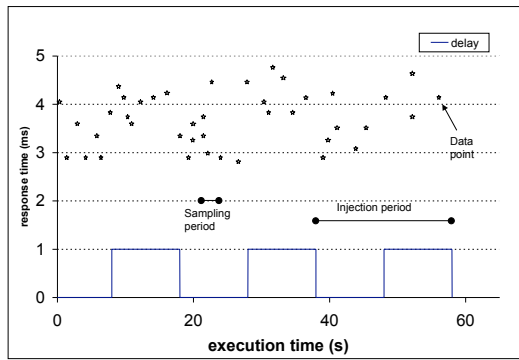
We perform such spectral analysis using the Fast Fourier Transform (FFT). The basic approach is illustrated in figure 2.2. The first graph, figure 2.2(a), shows two response time series for one of the transactions of a toy two-tier application that we built for experiments. The dashed time series was snapshot of transaction response time over a window of 128



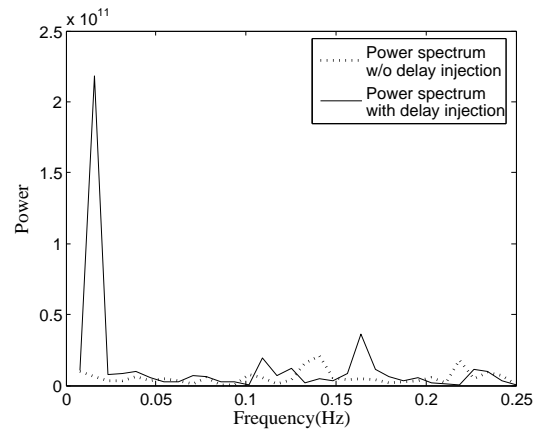
(a) Response Time Series



(b) Original Power Spectrum



(c) Delay Injection Pattern



(d) Power Spectrum with Delay

Figure 2.2: Power Spectra under Delay Injection

seconds. To examine how the time series looks in the frequency domain, we compute the power spectrum of the waveform using a Fast Fourier Transform (FFT). The power spectrum gives the amount of energy present in the waveform as a function of frequency. From the power spectrum, shown in figure 2.2(b), it can be seen that most of the energy in the waveform is present in the 0th frequency, which represents the non periodic portion of the waveform. For frequencies greater than 0, the energy in the spectrum is very low. Therefore, if the delay is injected in a periodic manner at any of those frequencies, the delay magnitude would also need to be very small.

Figure 2.2(c) shows how a periodic delay might be introduced into the link. The figure shows a square wave pattern. When the square wave is high, a constant delay is injected into all the messages that traverse the link. When it is low, no delay is injected. The (hypothetical) data points above the square wave show how the injected delay might affect the response time of the system. We then actually injected such a delay with a magnitude of 23 millisecond, and frequency of 0.015625 Hz into one of the links, measured response times and convert them into a response time series sampled at frequency 0.5Hz. The resulting response time series is shown by the solid line in figure 2.2(a). The corresponding power spectrum is shown in figure 2.2(d). Since it is very large, the 0 frequency term of the power spectrum is omitted. As can be seen from the response time series, the amount of delay injected is small compared to the original response time and its standard deviation. Nevertheless, the spike at the injection frequency of 0.015625 Hz in the power spectrum clearly shows that the effect of the injected delay dominates the other frequency components and allows an accurate measurement of the injected delay. Comparing the power spectrum with the original spectrum of figure 2.2(b), one can clearly see that the 0 frequency contains two orders of magnitude more power than the injection frequency component.

2.3 Spectral Link Gradient Computation

Next we develop formulae that allow compute now show how the height of the spike in the power spectrum can be used to compute the link gradient of the link. To do so, we consider some arbitrary response time series $x(t)$ of length N measured at uniform intervals of time ΔT_s , called the *sampling interval*. Using the values of the response time at the sampling intervals denoted $x(i \cdot \Delta T_s)$ or simply x_i , where $i = 0 \dots N - 1$, the power spectral density can be computed at all frequencies $f_k = \frac{k}{N \cdot \Delta T_s}$, $k = 0 \dots N - 1$ (i.e., those frequencies that fit an integral number k of cycles into the time series duration). That is done using the equation

$$\text{PSD}(f_k) = |\text{FFT}(f_k)|^2, \text{ where} \quad (2.2)$$

$$\text{FFT}(f_k) = \sum_{i=0}^{N-1} x_i e^{\frac{-2\pi j}{N} ik} \quad (2.3)$$

Now consider a delay added to each response time in a square-wave pattern with magnitude A_d and frequency $f_d = \frac{k_d}{N \cdot \Delta T_s}$ for some k_d . Then, we can show that the resulting Fourier Transform $\text{FFT}(f_k)$ is given by

$$\text{FFT}^d(f_k) = \frac{k_d A_d}{\sin(\frac{\pi}{2n})} e^{j(\frac{\pi}{2n} - \delta)} + \text{FFT}^0(f_k) \quad (2.4)$$

In this equation, $\text{FFT}^0(f_k)$ is the complex-value Fourier Transform of the original response time series without any delay injection, δ is the phase shift of the square wave in comparison with the time series interval, and $2n = N/k_d$. In other words, $2n$ is the total number of data points in each square wave cycle. Because the magnitude of the complex number $e^{j(\frac{\pi}{2n} - \delta)}$ is one, by manipulating Equation 2.4, we obtain an expression for the

amount of delay introduced into the waveform:

$$A_d = \frac{|\text{FFT}^d(f_k) - \text{FFT}^0(f_k)| \cdot \sin(\frac{\pi}{2n})}{k_d} \quad (2.5)$$

However, we also know from Equation 2.1 that if the link latency l of a link is increased by Δl , then the change in response time is given by $\bar{r}t(l + \Delta l) - \bar{r}t(l) \approx \frac{\delta \bar{r}t}{\delta l} \cdot \Delta l$. Equating this change in response time to the delay A_d measured from the frequency spectrum and Equation 2.5, we obtain the link gradient $\frac{\delta \bar{r}t}{\delta l}$ as a function of the system response time series without and with a square wave delay injection of magnitude Δl :

$$\frac{\delta \bar{r}t}{\delta l} = \frac{|\text{FFT}^d(f_k) - \text{FFT}^0(f_k)| \cdot \sin(\frac{\pi}{2n})}{\Delta l \cdot k_d} \quad (2.6)$$

An important point to note is that the phase shift δ of the square wave does not appear in the equation. That means that the square wave injector does not need a clock that is synchronized with the response time measurement, given that the clock skew won't manifest itself during the short measurement period.

CHAPTER 3

Architecture and Implementation

Using the basic approach described in the previous section, we have implemented a distributed active monitoring framework that automatically calculates the link gradient graphs for a distributed application. In this chapter, we first present the architecture of the monitoring framework, and then describe the algorithm it uses to drive the link gradient measurement process.

3.1 Monitoring Framework

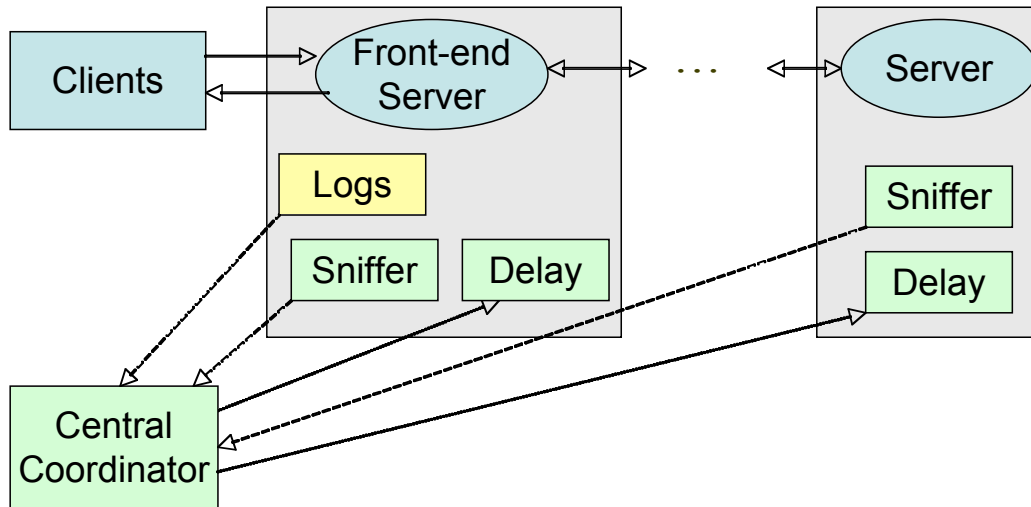


Figure 3.1: Monitoring architecture

The active monitoring framework can target one or more distributed applications deployed on a set of machines. The architecture of the framework on a sample target application is illustrated in figure 3.1. As seen in the figure, the framework consists of a central

co-ordinator and a set of *sniffer* and *delay* daemons, one for each physical host of the target applications.

3.1.1 The Sniffer Daemon

The sniffer daemon collects and periodically reports information about communication links to the central coordinator. During a reporting interval, the sniffer keeps track of the set of the communication links on which at-least one message has been exchanged, and the total number of messages exchanged during the reporting interval. Each communication link is defined using a 5-tuple $\langle \text{Protocol}, \text{SourceNode}, \text{SourcePort}, \text{DestinationNode}, \text{DestinationPort} \rangle$, where the protocol is either TCP or UDP, each node is represented using a tuple $\langle \text{HostIP}, \text{ProcessID} \rangle$, and the ports are represented by the corresponding TCP or UDP port number with one exception. When a node uses a port from the ephemeral range (port numbers 32768 to 61000 in Linux), we assume that it has been randomly assigned the port number by the operating system, and we replace that port by port number 0 so as to avoid spuriously creating multiple records when the same communication link is instantiated multiple times. The central co-ordinator uses the information provided by the sniffer to create an initial “communication graph”.

3.1.2 The Delay Daemon

The delay daemon implements delay injection, and starts and stops injection based on commands from the coordinator. We implement two versions of the delay daemons. The first version is based on the loadable packet filter kernel module in regular Linux kernel. The second version is specific for nodes on Planetlab, since they use a experimental Linux kernel which has the packet filter module disabled. We implement the second version using the `tap` device and UDP tunneling for planetlab machines.

Delay Daemon on regular Linux machines

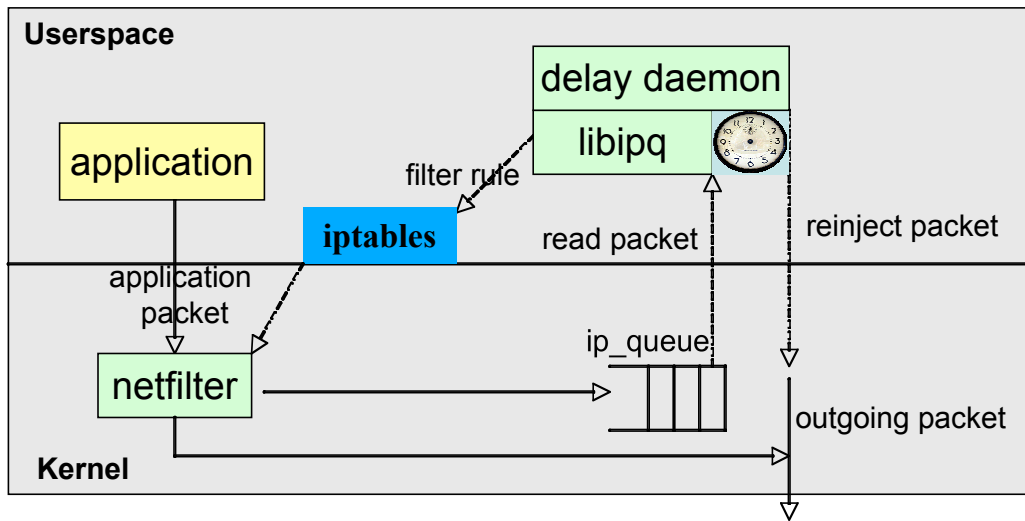


Figure 3.2: Implementation of delay daemon on regular Linux machines

The implementation of the delay daemon on regular Linux machines is illustrated in figure 3.2. To implement delay injection, it utilizes the `ip_queue` (`libnetfilter_queue` in Linux 2.6.14 or later) packet filter kernel module and `libipq`¹ to queue packets going through a link, delay them, and re-inject the packets back to the network stack. In all Linux kernels subsequent to the 2.4 series, every IP packet goes through the netfilter hook. The `ip_queue` loadable kernel module is provided by these kernels as a method to utilize the netfilter hook to queue packets in kernel space and allow their manipulation from user space. The delay daemon controls the `ip_queue` module by using `iptables`, a userspace command line program used to configure the Linux packet filtering ruleset.

The delay daemon can be instructed by the central coordinator to start delay injection with a specified delay magnitude and frequency on a specified communication link (specified using its 5-tuple), or to stop delay injection. The delay daemon starts or stops injection by installing (or removing) an `iptables` rule that forwards all packets from the designated

¹`libipq` is a development library for `iptables` userspace packet queuing that provides an API for communicating with `ip_queue`.

communication link to an in-kernel packet queue. The user-space daemon then uses the `libipq` library to read the metadata of the queued packets from `ip_queue` and re-inject the packet back to the network stack after the specified amount of delay. In order to inject the delay, the delay daemon uses a timer-wheel implementation [5] where it maintains a queue of packets scheduled to be sent in order of their send times. A single timer corresponding to the next packet to be sent is set. When this timer fires, all packets that were to be sent out at or before the current time are sent and the timer is reset for the new packet at the head of the queue. Because the standard clock tick interval for unmodified the 2.6 Linux kernel series is 1ms, that is the best resolution expected for this timer implementation. In chapter 4, we present benchmark results for the injection mechanism that quantify its accuracy under various system loads. In order to rectify effects due to any inaccuracies in the timer mechanism, the delay daemon measures the amount of delay actually injected into each packet, and reports the mean back to the central coordinator.

Because the delay injection mechanism only reads fixed size metadata from the kernel, the efficiency of the mechanism is independent of the size of the packet. In the current implementation of `ip_queue`, the metadata is 72 bytes long. Thus, the overhead introduced due to data copying from kernel to userspace is expected to be small.

Delay Daemon on Planetlab machines

The implementation of the delay daemon for Planetlab nodes is illustrated in figure 3.3. Since `ip_queue` is not available on planetlab nodes, we utilize the `tap` network device. `tap` simulates an Ethernet device and it operates with Layer 2 packets such as Ethernet frames. On PlanetLab nodes, VNET² emulates a single TAP interface `tap0`. Each slice may access its own packets through `tap0` by reading from and writing to the `tap0` interface via IP or packet sockets. We configure applications send and receive packet via the

²VNET is a PlanetLab module that provides virtualized network access on PlanetLab nodes

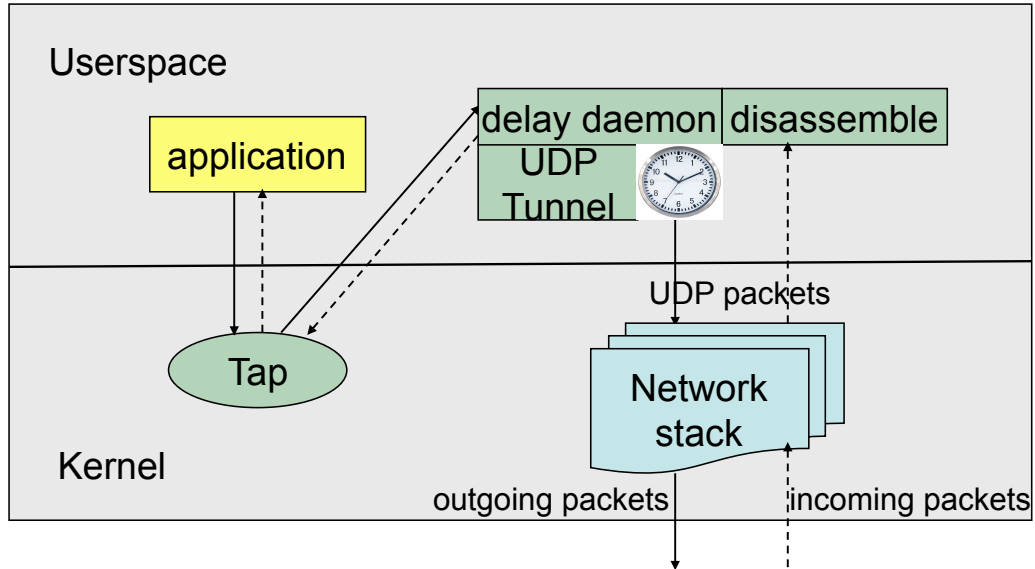


Figure 3.3: Implementation of delay daemon on Planetlab

`tap0` device. The delay daemon listen on `tap0` device and read from it the applications' packets(including the Layer 2 header). Then the daemon removes the packets' Layer 2 header, modifies the Layer 3 header, delay them using the technique described above and send the entire Layer 3 packets as payload via a UDP tunnel to the packet's destination. On receiving a packet via the UDP tunnel, the daemon disassembles the packet, remove the UDP header, modify the Layer 3 header and write the payload to the `tap0` device. The user application, which is configured to listen on the `tap0` device, then receive the packets.

Our second implementation on PlanetLab requires applications to listen on `tap` device and the delay daemon to run at every node in the system. Also, the delay daemon need to read the entire packet payload from the `tap` device to the user space, which introduces larger copy overhead than the first implementation. However, our second implementation is only specific for nodes on PlanetLab that do not have `ip_queue` module support. On regular Linux machines, the first implementation should be widely feasible.

3.1.3 The Central Coordinator

Finally, the central coordinator controls the monitoring system, and periodically executes the measurement algorithm to construct and update the link gradient graph for the target application. To instantiate the coordinator for a distributed application, the user provides a list of nodes (host, process names) that are part of the application, and a target metric. Currently, the framework supports response time as a target metric, and has a module that can parse Apache web server logs to extract response time data for a specified URL. Using this module, any application that has a web-based interface can be supported, and the link gradient graph for any individual transaction (URL) can be constructed. The link gradient graphs are constructed using only existing application workload response time series' extracted from the web logs. No additional workload is required for measurement purposes and thus, the technique introduces minimal interference to a running system.

The coordinator periodically queries the sniffer daemons on each of the target hosts and aggregates the link information they provide to create a communication graph of the target application. The communication graph is a directed graph where nodes represent vertices and communication links as reported by the sniffer daemon are the edges. To avoid insignificant communication edges, the coordinator allows the user to set a threshold for a number of messages that must be sent before a communication connection is added to this graph. Since the communicator graph reports all the links between nodes that form a part of the target application, it may also include communications that are not part of the target itself or that belong to more than one application. For instance, consider a 3-tier web service that allows multiple transactions (e.g., browse, buy, sell etc.). Since each transaction uses the same set of components, any communication links initiated as part of one of these transactions would appear in the communication graphs for all of them.

Finally, the coordinator also instructs the delay daemon to start or stop delay injection as determined by the link gradient graph construction algorithm that is described next.

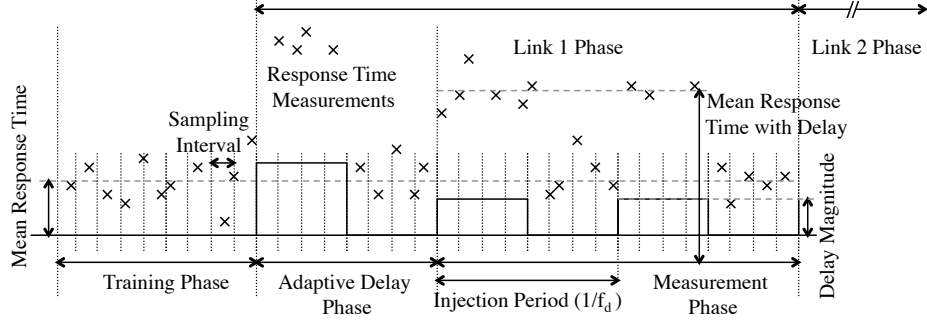


Figure 3.4: Timeline for the measurement process

3.2 Measurement Algorithm

The process of constructing the link gradient graph for an application consists of two phases: a) the training phase, and b) a set of measurement phases, one for each communication link. The training phase is conducted only once for the entire link gradient graph creation process. In this phase, the coordinator constructs the communication graph, and passively collects the system response times for the transaction in question. It uses the response time series in order to determine the parameters for the delay square wave that is to be injected into the communication links. The measurement phase is conducted once for every communication link in the communication graph, and is the active phase where a delay is introduced into the target link and the system response is measured. Next, we describe the two phases in detail. Figure 3.4 presents a timeline of the entire process.

3.2.1 Training Phase

The training phase is used to estimate parameters such as the magnitude of the delay injected into the system and the frequency at which it is injected for the active measurement phases. To ensure the validity of using the parameters estimated in this phase during the later phases, we assume that the request rate for each transaction and the characteristics of any noise in the response time series remain stable between the training phase and the

measurement phase for a link. These two assumptions are reasonable since the time needed to extract the link gradient for a communication link is a few minutes, and we do not expect the measurement to be done during periods of rapidly changing traffic or flash crowds. Although it is not required, we assume that these assumptions hold for the entire duration of the link gradient graph construction. This allows us to conduct only a single training phase, and use its results for all the measurement phases. The training phase comprises the following steps.

1. Since the applications own workload is used to perform the link gradient measurement, the measurement framework has no control over when response time measurements (data points) are made. In order to obtain the periodic time series required for the Fourier Transform, the framework creates bins of length equal to a *sampling interval* as shown in figure 3.4, and uses the mean of all the data points within a bin as a single sample point in the time series. To choose the length of the sampling interval, the coordinator records the mean \bar{m}_t and standard deviation $\sigma(m)_t$ of the time interval between k consecutive requests. The sampling interval is then chosen as $\Delta T_s = \bar{m}_t + 3\sigma(m)_t$ to ensure that at-least k points are averaged in each sampling interval with a high probability (0.999 if the time between k requests has a normal distribution). The parameter k that governs the number of data points per bin is provided as an input by the user. Once the bin size is computed, framework can construct a response time series from the web server log files.
2. In order to choose an appropriate frequency and magnitude for delay injection, it is important to examine the characteristics of the frequency domain representation of the system's response time series without perturbation. To do so, the coordinator divides the response time series into M (we use $M = 9$) different chunks of N sampling points. The parameter N is a user specified parameter that indicates the number of sampling points used in each link gradient computation. In our implementation,

the total number of sampling points per measurement phase N is restricted to be a power of 2 so that a fast radix 2 FFT algorithm can be used. The Fourier Transform for each of the M chunks is then computed, and these transforms are then averaged to obtain the mean $\overline{\text{FFT}}^0(f)$ and standard deviation $\sigma(\text{FFT}^0)(f)$ (both complex numbers) of the Fourier Transform at each frequency $f = \frac{k}{N \cdot \Delta T_s}$. These can be used to select both an appropriate delay magnitude and frequency.

3. To choose a delay magnitude, imagine that a delay frequency $f_d = \frac{k_d}{N \cdot \Delta T_s}$, $k_d \in \{0, \dots, N - 1\}$ has been chosen, a delay of magnitude A_d has been injected. Then, according to Equation 2.4, the Fourier Transform of the resulting response time series is equal to $\text{FFT}^d(f_k) = \frac{k_d A_d}{\sin(\frac{\pi}{2n})} e^{j(\frac{\pi}{2n} - \delta)} + \text{FFT}^0(f_k)$. However, because they are measured at different times, the value $\text{FFT}^0(f_k)$ that appears in this equation can be different than the value $\overline{\text{FFT}}^0(f)$ computed in step 2. To reduce the relative error due to this mismatch, we make it proportional to the standard deviation $A_d = d * \sigma(\text{FFT}^0)(f)$ computed in the previous step (which is a measure of the natural variability of the Fourier Transform components). The constant of proportionality d , is called the *delay scale factor*, and is set to 30 in all our experiments. We examine the sensitivity of the measurement technique to the delay scale factor in chapter 4.
4. Given the method to compute the delay magnitude for a particular frequency, the coordinator simply computes the delay magnitude for all frequencies whose periods are factors of the measurement phase length (i.e., $f_d = \frac{k_d}{N \cdot \Delta T_s}$, $k_d \in \{0, \dots, N - 1\}$), and picks the frequency that would result in the smallest delay magnitude so as to minimize the perturbation to the system.

3.2.2 Measurement Phase

Once the coordinator has finished computing the bin size ΔT_s , delay magnitude A_d , and injection frequency f_d in the training phase, the measurement phases are relatively straight forward. The measurement phase for each link consists of two sub-phases as shown in figure 3.4: the adaptive delay phase, and the active measurement phase. In the adaptive delay phase, the coordinator injects a single cycle delay with a small magnitude A_d^0 and frequency f_d into the target communication link and measures the corresponding response time series from the web server logs. It also obtains the mean value of the delay that was actually injected, and adjusts the value A_d^0 to reflect it. However, if the link gradient $\frac{\delta \bar{r}^t}{\delta l}$ is not equal to 1, the actual delay that would manifest in the response time as a result of the injection would be $\frac{\delta \bar{r}^t}{\delta l} A_d$, and can become very high if a lot of messages are being exchanged on the link. Therefore, in the adaptive delay phase, the coordinator computes a rough estimate $\frac{\delta \bar{r}^t}{\delta l}$ of the link gradient using Equation 2.6, and uses it to scale the magnitude of the injection delay $A'_d = A_d / \frac{\delta \bar{r}^t}{\delta l}$ so that the overall magnitude of the delay in the response time is approximately equal to A_d . In this manner, the adaptive delay sub-phase ensures that intrusiveness is kept low irrespective of the value of the link gradient.

In the active measurement sub-phase, the coordinator injects a delay of magnitude A'_d with frequency f_d into the system, collects enough response time measurements to compute a time series of N bins, and computes the final value of link gradient using the measured response times, the mean value of the actual delay injected as reported by the delay daemon, and Equation 2.6. In this manner, when provided with a list of the application nodes, the location of the web server log, and the target URLs, the coordinator can automatically generate the entire link gradient graph for all the target transactions in all the applications.

CHAPTER 4

Micro Benchmarks

As described in chapter 3, the algorithm for computing the link gradient for an application requires a few parameters as inputs. These include the total number of sampling bins N to use for each link gradient measurement phase, the delay scale factor d , and the minimum number of data points per bin k . In this chapter, we present micro-benchmarks using a controlled application setup to examine the sensitivity of the accuracy of the measurement technique to these and other factors. The goal of these experiments is to provide a better understanding of the strengths and limitations of the technique, and to identify ranges of parameter values that can be used during runtime measurement.

A simple setup consisting of a single front-end server communicating with a single back-end server running on the same testbed as described in chapter 2 is used in all the experiments in this chapter. When the frontend server receives a client’s request, it simply makes a call to the back-end server. The backend server processes the request for a normally distributed random amount of time with a mean of 100msec and a standard deviation of 30msec and returns back to the frontend server, which then generates a reply back to the client. The client node in the experimental setup generates requests one at a time and waits to generate the next request until it receives a response from the first. The response time is recorded by the frontend server in a “web-like” log file.

The metric used for measuring accuracy in this setup is the link gradient of the forward link between the frontend and backend server, which is expected to be 1. Each experiment is replicated 10 times to compute the mean and standard deviation of the link gradient.

However, we first examine the accuracy of our netfilter based user-mode delay injection mechanism.

4.1 Accuracy of Delay Injection

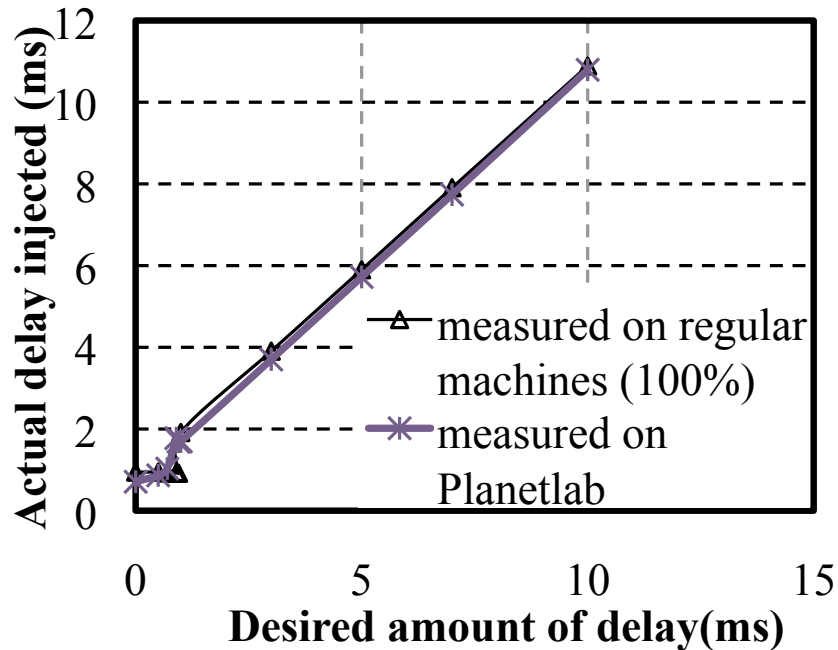


Figure 4.1: Accuracy of delay injection

In the first set of experiments, we measure the actual amount of delay injected into the packets of the target link as a function of the desired delay injection. To evaluate whether the accuracy of the user-mode mechanism depends on whether the system is heavily loaded or not, the experiments are conducted under both 0% CPU load and 100% CPU load conditions. We do not consider I/O load and memory usage in the benchmark, since our injection technique only utilizes the CPU and memory resources and the memory usage is kept as low as 5MB. We measured both the delay injected as measured by the delay daemon (using the `gettimeofday` system call) as well as the end-to-end delay introduced (as measured by an ICMP ping command between the two servers). The resulting mean injected delay is

shown in figure 4.1. As seen in the figure, for desired delays up to 1 msec, both the injected delay and end-to-end delay are accurate except for a consistent shift of 1msec. Moreover, the accuracy is practically unaffected by the CPU load. However, when the desired amount of delay is smaller than 1 millisecond, the actual injected delay is on the average 1 millisecond. This is due to the fact that in Linux kernel we used for our experiments, the scheduling granularity is 1 millisecond (1 jiffy). When the requested amount of delay is greater than 1 millisecond, our injection library injects roughly a delay of 1 jiffy plus the requested amount of delay. The extra 1 jiffy accounts for one scheduling overhead. When the desired delay is less the scheduling overhead, the amount of delay injected is roughly equal to the scheduling overhead.

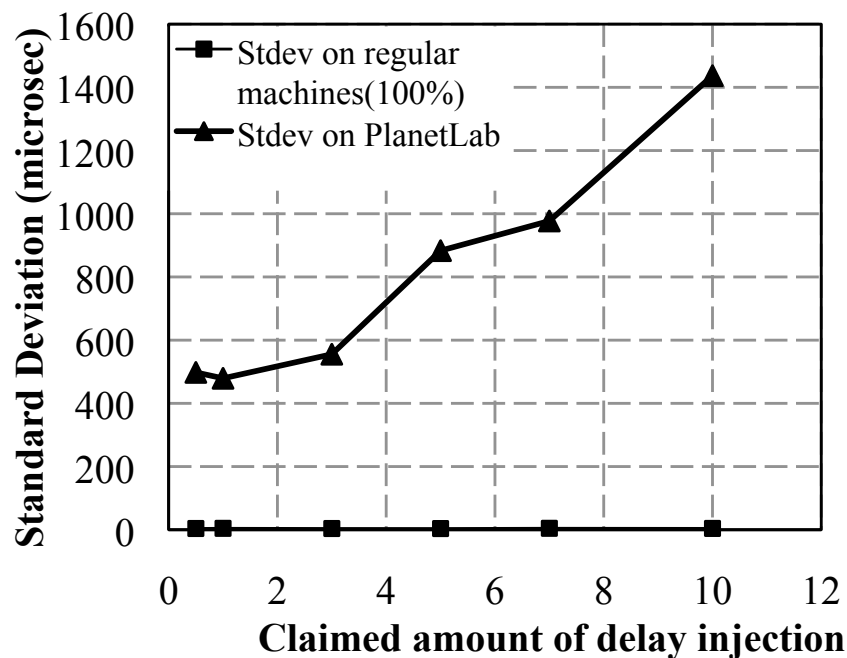


Figure 4.2: Reliability of the injection mechanism

Figure 4.2 shows the plot of standard deviation of the overhead introduced by our injection mechanism. From the figure, we can see that the standard deviation of overhead is less than 5 microsecond even under 100% CPU workload. Therefore, this shows that the injection library is able to inject delays as small as 1 msec reliably even under heavy

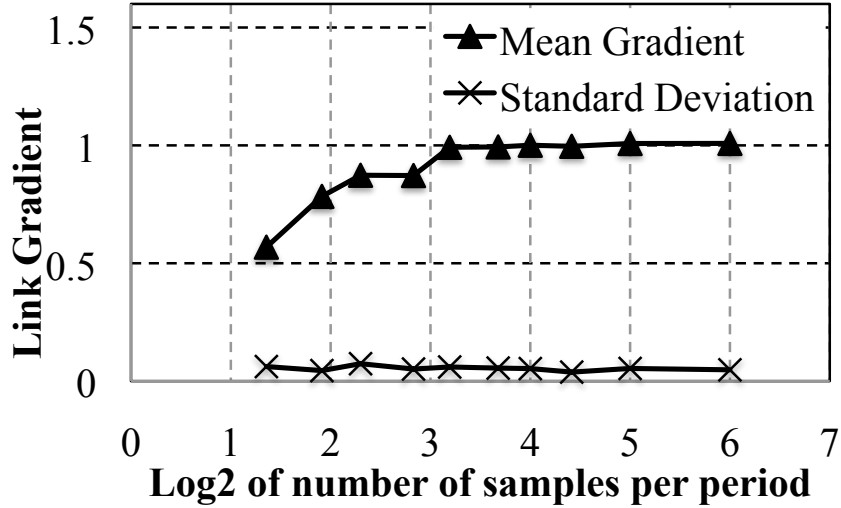


Figure 4.3: Sampling Points per Period

workload. While using the injection library to measure link gradients, we always ensure that the injection delays are greater than 1ms and correct for the consistent 1ms difference between requested and actual delay injected by measuring the actual delay injected during runtime, and reporting it back to the central coordinator.

4.2 Sampling Points per Period

In the first set of experiments measuring the accuracy of the results, we vary the number of sampling points per cycle of the square wave (i.e., the injection period) to see how this parameter affects the accuracy of the link gradient computation. We carried out experiments with a fixed minimum number of data points per bin set (k) set to 8, a fixed amount of delay of 30 msec (roughly corresponding to a delay scale factor of about 30), and a fixed N set to 128. The number of sampling points per injection period were varied between slightly more than 2 (the Nyquist rate), up to 64 (corresponding to only 2 cycles during the entire measurement phase). From the results of the shown in figure 4.3, the accuracy is poor at an injection frequency close to the Nyquist frequency, but improves rapidly as the number

of data points per injection period increase (and the number of cycles decrease) and approaches 1 when there are more than 8 data points per period (corresponding to frequencies between 0 and 1/16 the Nyquist frequency in this case). Moreover, further experimentation showed that any increase in the length of measurement phase (and the number of injection periods) does not help compensate the error introduced by decreasing number of sampling points per period.

This result is counter-intuitive to the perception we had that accuracy of the method would improve as the number of complete injection cycles increased (due to an increase in the “regularity” of the perturbation). To understand the reason for this result, recall from section 3.2 that the central coordinator bins response time measurements and averages all the data points in a bin to produce a uniformly spaced time series. As a result, bins that fall on the edges of the square wave injection have values that may lie between 0 and 1, causing a smoothing of the square wave injection pattern. That causes the link gradient to be inaccurate since the derivation of equation 2.6 requires a perfect square wave. However, as the number of injection cycles decrease due to an increase number of sample points per period, the smoothed sample points (which only occur at cycle edges) decrease in comparison to the total number of points, thus leading to improved accuracy.

From these results, it is clear that if the number of sample points in an experiment is fixed (i.e., fixed experiment length), the injected frequency must be low enough that noise does not cause inaccuracy in the results, but high enough that the system’s normal response does not have a significant component at that frequency. In our implementation, we achieve this result by only considering relative frequencies between 2 and 16 (corresponding to between 2 and 16 injection cycles per measurement phase respectively) in the delay minimization in step 4 of the training phase of the measurement algorithm of section 3.2. We have discovered that this restriction does not significantly reduce the number of “good” frequencies available for delay injection, and in practice provides very good results.

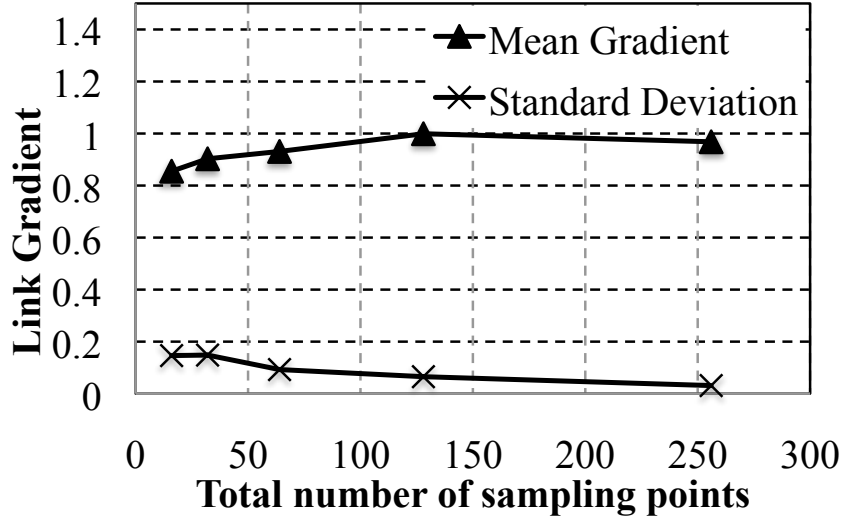


Figure 4.4: Link Gradients v.s. Total Number of Sampling Points

4.3 Total Number of Sampling Points

Next, we investigate how the total number of sampling points affects the accuracy, stability, and intrusiveness of the link gradient measurements. In these experiments, the amount of delay injected is fixed to 30 msec, the bin size (k) is set to 8, and the number of sampling points per experiment is varied from 16 to 256. Figure 4.4 shows both the resulting mean link gradient, and the corresponding standard deviation (which provides a metric of stability of the link gradient measurement) across 10 experiments. As expected, we observe that as the number of sampling points increases, the accuracy increases until it reaches and remains close to the correct value (1.0). Moreover, the standard deviation of the result decreases monotonically as we increase the number of sampling points indicating improving stability (each individual link gradient measurement is closer to its true value).

In addition, Figure 4.5 shows how the standard deviation of the response time series without any delay injection $\sigma(\text{FFT}^0)(f_d)$ changes as the number of sampling points in the measurement interval are changed. The figure shows that the mean standard deviation across all the frequencies in the FFT, the smallest standard deviation amongst all the fre-

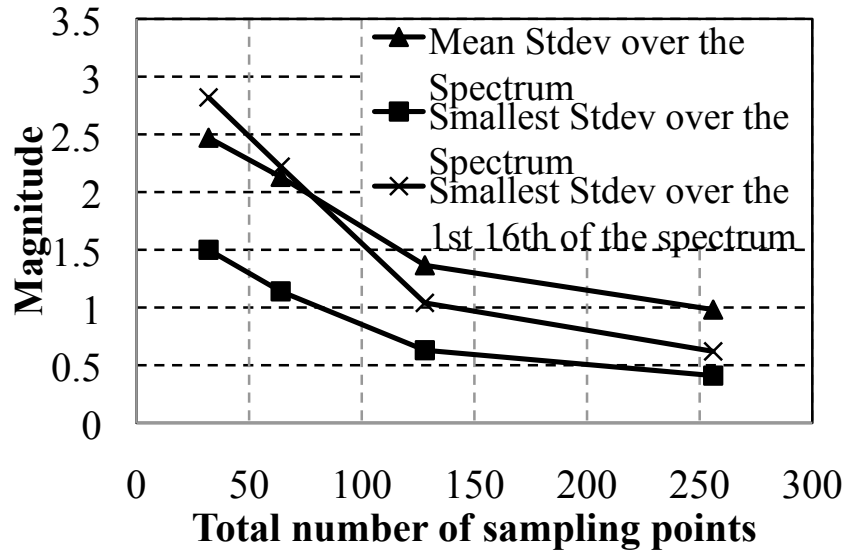


Figure 4.5: Amount of Delay v.s. Total Number of Sampling Points

quencies, and the smallest standard deviation amongst the first 1/16th part of the spectrum (the set of frequencies used to choose the injection frequency) all decrease monotonically as the number of sample points decrease. This is due to the fact that, as the number of sample points increases, the corresponding increase in the number of injection cycles causes a reduction in any periodic noise components. Moreover, as the granularity in the frequency domain increases (due to a larger number of sample points), smaller amounts of noise energy is aggregated into each available frequency in the spectrum, this leading to a smaller standard deviation in those components. Now recall from step 3 of section 3.2 that the amplitude of the injected delay is set to be a multiple of the frequency with the smallest standard deviation ($A_d = d * \sigma(\text{FFT}^0)(f)$). Therefore, the amount of delay injected into the link and its associated perturbation decrease as well.

From these results, it is clear that increasing the total number of sample points (and thus the number of data points) is always a good thing if everything else remains the same. However, such an increase leads to an increase in the length of the measurement phase, and may be dictated by external factors (e.g., how long the user can wait before obtaining results).

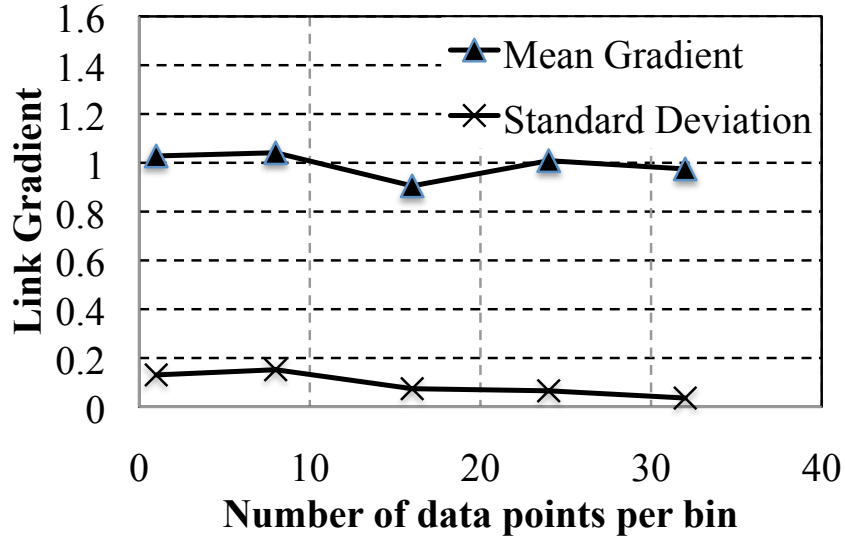


Figure 4.6: Sampling Bin Size

4.4 Data Points per Bin

In the next set of experiments we evaluate the sensitivity of the link gradient measurement to the minimum number of data points per bin (k) used to compute the bin size. Recall from step 1 that the response time series is constructed by averaging all the response time data points in each bin. Therefore, the binning also acts an averaging filter, and performs noise filtering functions without making any assumptions about the stochastic properties of the noise. Figure 4.6 shows how the accuracy of the link gradient computation changes as the k parameter is changed from 1 to 32 with the total number of sample points fixed at 64, the number of sampling points per injection period is fixed at 32, and the injection delay is set to 30 msec.

As seen in the graph, that although the number of data points per bin does not affect the mean link gradient value much, the standard deviation does slowly reduce, indicating, as expected, that increasing the number of data points per bin increases the stability of the result. However, if additional data points are indeed available, then comparison with previous results indicates that it is much better to increase the total number of sampling

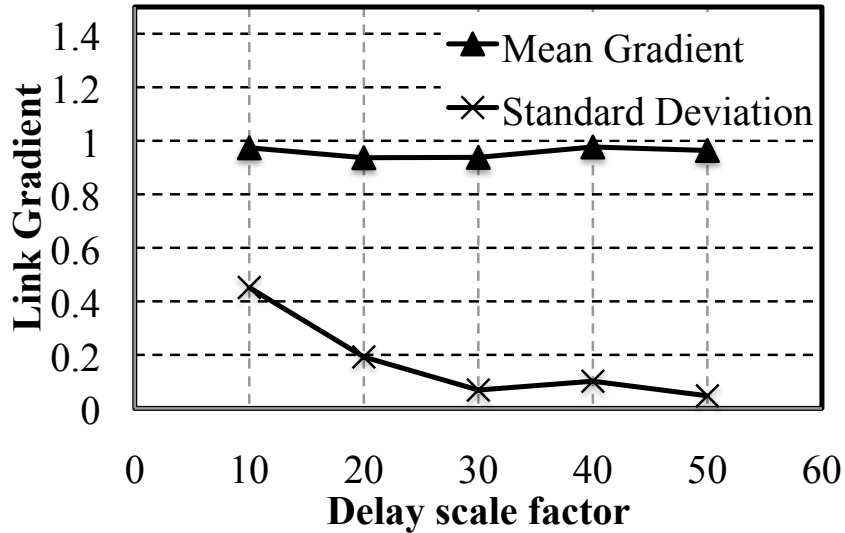


Figure 4.7: Delay Scale Factor

points rather than increasing the number of data points per bin, since the former not only leads to an increase in the result stability, but also requires a smaller delay to be injected in the process. Therefore, in our implementation, we set the default value of k to our lowest tested value of 8 until the user indicates otherwise.

4.5 Delay Scale Factor

In figure 4.7, we look at how the delay scale factor d affects the accuracy and stability of the result. In practice, one might expect that as long as the injected delay remains below some application specific threshold such that the application or TCP behavior is not significantly affected, the more we perturb the system, the more accurate the result. However, large perturbation is only possible in a prototype system before deployment. In a running system, the perturbation should be as low as possible while still allowing accurate computation of the link gradients. Fortunately, the results of changing the delay scale factor from 10 through 50 while keeping the total number of sample points N as 64, and number of data points per bin $k = 12$ show that low delay scale factors do not affect the accuracy

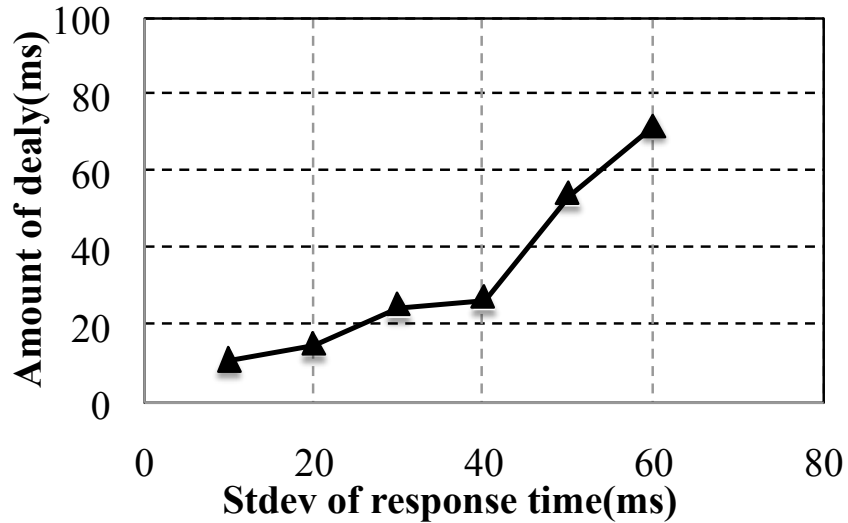


Figure 4.8: Amount of Delay V.S. Standard Deviation of Response Time

of the link gradient, just its variance. Moreover, the variance decreases rapidly at first, and much more slowly after the scale factor increases beyond 30. The result is expected since a lower scale factor implies that the injected delay is smaller in comparison with the natural variance of response time spectrum. Based on these results, our implementation uses a default delay scale factor of 30 as a good trade-off between intrusiveness and stability unless the user indicates otherwise. Since the delay magnitude can be increased without increasing the length of the experiment, the results suggest that when quick results are required, increasing the delay is a good way to ensure stability (with the warning that too much delay injection might trigger recovery mechanisms in the application, and might be counter-productive).

4.6 Standard Deviation of Response Time

Finally, to conclude the micro-benchmark results, we examine how the amount of delay required for a fixed delay scale factor (set to 30) varies as a function of the variance of the system's response time. Figure 4.8 shows a plot of the amount of delay $A_d = d \cdot$

$\sigma(\text{FFT}^0)(f_d)$ computed by training phase of the coordinator as the standard deviation of the system's response time is changed from 10 to 60 (by changing the variance of the normally distributed response time of the back-end server). As expected, the amount of delay required also increases almost linearly. However, it never exceeds standard deviation of the system response time. We believe that this result indicates that our technique is able to achieve its objective of low intrusiveness (comparable with the normal variance of the system itself), while still achieving high levels of accuracy in the micro-benchmarks.

In this section, we have examined the accuracy of the link gradient measurement as a function of several parameters of the measurement framework and the target system. These experiments provide a general guideline that can help users understand choose how to set up the parameters to the framework. In general, we recommend that the best way to improve the accuracy and stability of the results while minimizing intrusiveness is to increase the number of sampling points per measurement phase. However, recognizing that increasing the measurement duration may not be possible, the delay scale factor can be increased as long as it does not trigger recovery mechanisms within the application.

4.7 Simple Application Scenarios

Our tested application scenarios, shown in figure 4.9, represent various communication patterns between server tiers in a two-tier service. Specifically, the scenarios from 1 to 7 in the figure include repeated invocation, sequential invocations to multiple second tier servers, load balancing, caching, two different types of parallel invocation, and asynchronous communication. The experiments for each scenario follow the methodology outlined in the previous sections. In particular, for each link, the transaction response times are first collected during the observation stage to calculate the amount and frequency of the delay to be injected, and then the perturbed response times are collected in the injection stage to

calculate the link gradient. The front end server *Server1* is responsible for collecting the response times for each transaction.

Transactions are generated at the client according to a Poisson arrival process with arrival rates for each scenario as shown in table 4.1. The service times for *Server2* and *Server3* in all scenarios are normally distributed. For *Server2*, the mean is 100 ms and standard deviation 30 ms in all cases, while for *Server3*, these parameters depend on the scenario. In particular, the values for scenario 2 are 100 ms and 30 ms, for scenario 3 are 150 ms and 45 ms, and for scenarios 5 and 6 are 200 ms and 60 ms. The cache miss rate in scenario 4 is 0.3 from *Server1* to *Server2*. Inputs to our framework are the same across all scenarios—the number of sampling points per experiment is set to 128, delay scale factor to 30 and data points per sampling bin to 12. The specific parameters calculated by our framework during the first stage and used during the second are shown in table 4.1.

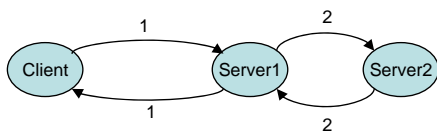
Case	arrival rate	SP freq.	freq.	delay	length	\bar{rt} w/o delay	\bar{rt} w/ delay
1	$4s^{-1}$	0.33	5.16e-3Hz	36ms	384s	202.03	217.90
2	$4s^{-1}$	0.33	2.58e-3Hz	46ms	384s	200.16	222.34
3	$6s^{-1}$	0.50	1.95e-2Hz	42ms	256s	124.77	135.4
4	$6s^{-1}$	0.50	2.73e-2Hz	82ms	256s	58.33	77.76
5	$4s^{-1}$	0.33	5.16e-3Hz	61ms	384s	201.2	228.91
6	$6s^{-1}$	0.33	2.58e-2Hz	30ms	256s	98.63	112.91
7	$6s^{-1}$	0.50	2.73e-2Hz	27ms	256s	100.20	101.14

Table 4.1: Parameters and mean Response time. Notes: SP: sampling

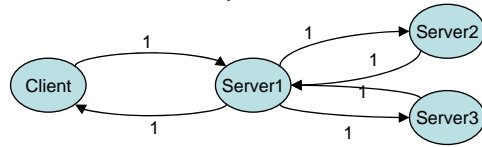
The results of the link gradient calculation for each scenario are presented in table 4.2. C, S1, S2, and S3 are used to denote the Client, Server1, Server2, and Server3, respectively. E is the expected or ideal gradient value calculated as described in chapter 2, while MF and MR are the measured link gradients in the forward and reverse direction, respectively. The mean across $M = 9$ experiments is given for each gradient along with the standard deviation in parentheses.

These results demonstrate that our technique can extract the link gradients quite accurately in most cases, with the computed value being quite close to the expected value. Note

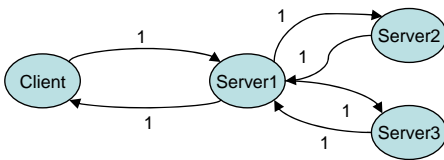
1. Basic Multi-tier



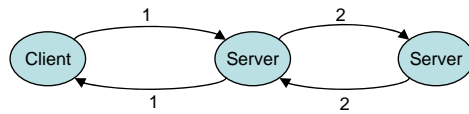
2. Two-Tier Sequential



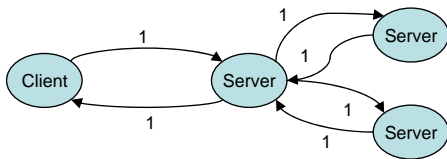
3. Two-Tier Load Balanced (OR)



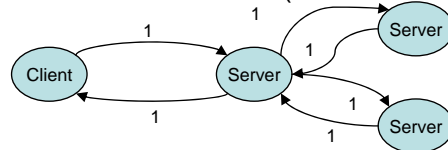
4. Two-tier with caching (0.3 rate)



5. Two-Tier Parallel (AND)



6. Two-Tier Parallel (return after 1st reply)



7. Asynchronous invocation

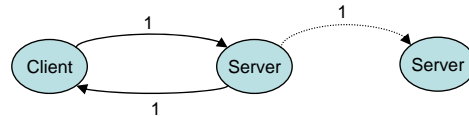


Figure 4.9: Example Scenarios

Case	S1 – S2			S1 – S3		
	E	MF	MR	E	MF	MR
1	2	1.92(0.09)	1.96(0.05)	N/A	N/A	N/A
2	1	1.00(0.06)	1.01(0.06)	1	0.99(0.07)	1.00(0.05)
3	0.5	0.51(0.06)	0.47(0.06)	0.5	0.53(0.07)	0.49(0.06)
4	0.6	0.69(0.11)	0.71(0.10)	N/A	N/A	N/A
5	0	0.14(0.05)	0.16(0.02)	1	0.95(0.06)	0.96(0.06)
6	1	0.92(0.05)	0.93(0.05)	0	0.08(0.05)	0.10(0.04)
7	0	0.07(0.03)	0.07(0.04)	N/A	N/A	N/A

Table 4.2: Results

especially that the values in scenario 3 capture the load balancing being performed across the two second-tier servers, and that the values in scenario 4 capture the caching behavior at $S1$.

Scenarios 5 and 6 also demonstrate the ability of the approach to render more complex situations accurately. These scenarios depict the situation where $S1$ communicates with $S2$ and $S3$ in parallel, and then either waits for all replies or the first reply, respectively. In scenario 5, because $S1$ waits for both $S2$ and $S3$, and the mean service time of $S3$ is greater than $S2$, the main impact on response time is $S3$ rather than the latency on the link from $S1$ to $S2$. These results demonstrate that our technique is able to capture this bottleneck at the link between $S1$ and $S3$. Also, the non-zero link gradient between $S1$ and $S2$ indicates that this link might become a performance bottleneck when the service time of $S2$ is greater than the service time of $S3$. In contrast, in scenario 6, the fast link between $S1$ and $S2$ has more impact on response time since the front end server only needs the first reply to continue.

Finally, to quantify the overhead for our technique, we compare the mean response times with and without delay injection for all scenarios. These values are shown in table 4.1. The results illustrate that our technique is able to extract link gradient information with relatively low perturbation in most cases. Note that the increase in response time

is naturally correlated with the amount of delay injected for each scenario as shown in table 4.1.

CHAPTER 5

Evaluation: RUBiS on PlanetLab

In this chapter, we evaluate the approach and its predictive power in a realistic setting. Predictive power is measured by measuring the link gradient’s ability to predict the response time of each application transaction in a configuration *different* from the one that was used to compute the link gradient. We show that although the relationship between latency and end-to-end response time changes across varying workloads, configurations, communication models, and load-balancing policies, the link gradient is accurately able to capture those effects. Finally, we show how the link gradient can be used to optimize single application configurations including on a transaction-by-transaction basis and multiple applications.

5.1 Experimental Setup

We used a deployment of RUBiS on PlanetLab. RUBiS [3] is a well-known eBay-like auction application that has been extensively used as a benchmark in the literature. Although small, it is representative of many multi-transaction, multitier web applications and can be configured using many settings (e.g., load balancing, connection pooling and replication) that make it hard to predict the effects of changing network latency. Deployment on PlanetLab nodes distributed across the United States ensured that the measurements were made in a realistic, heavily shared, high-variance, challenging wide-area environment.

We used the 3-tier Java-servlet-based version of RUBiS with a front-end Apache server (WS), middle-tier Tomcat application servers (AS), and a back-end MySQL database server

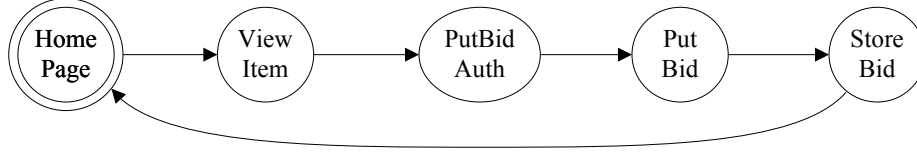


Figure 5.1: Client Transition Diagram

Users/ λ	Link	View Item	Store Bid	PutBid Auth	Put Bid	λ_{lg}
20/ 2.23	AS-WS	1.92	1.05	1.78	2.04	2.21
	DB-AS	15.14	18.76	0.63	18.03	2.20
30/ 3.40	AS-WS	1.47	0.68	1.46	1.75	3.31
	DB-AS	16.99	18.90	0.51	19.13	3.32
40/ 4.41	AS-WS	1.86	1.02	1.78	1.89	4.41
	DB-AS	16.28	18.19	0.83	18.77	4.39
50/ 5.56	AS-WS	1.70	0.82	2.08	1.58	5.54
	DB-AS	18.24	20.37	0.46	20.40	5.48
60/ 6.75	AS-WS	1.48	0.87	2.05	1.97	6.62
	DB-AS	21.74	22.80	0.87	23.15	6.53

Table 5.1: Link Gradients

(DB). We used the standard RUBiS workload generator with randomly generated TPC-W client think times [6], but for space reasons, restricted each client to only 5 out of 26 possible transactions, chosen because of their potential to stress different parts of the system. Each client uses the bidding oriented transition diagram shown in figure 5.1 - *ViewItem* (VI) returns information about an item, *PutBidAuth* (PBA) returns a user authentication page, *PutBid* (PB) performs authentication and returns detailed bidding information, and *StoreBid* (SB) stores a bid in the database. Of these, *PutBidAuth* is web and application server centric, while the other transactions are application and database server oriented.

5.2 Link Gradient Computation

We used a setup of RUBiS identical to the one shown in figure 1.1 of chapter 1 with a single server of each type and the default configuration settings. The workload generator and web server were located in San Diego (UCSD), while the app and db servers were

located in Pittsburgh (CMU). In all experiments, the link gradient algorithm was set to use 1 data point per bin, 3456 sampling points per experiment, and a delay scale factor of 30. Table 5.1 shows the measured link gradients for all the transactions as the workload is varied from 20 to 60 concurrent clients. It also shows the normal throughput of the system for each workload (λ) and the modified throughput during link gradient measurement (λ_{lg}) for each link. Although we measured the link gradients for both directions of each link (e.g., WS-AS and AS-WS), the table shows only a single direction because the results were very similar. However, as we show later, that may not necessarily be the case, and the ability of our approach to measure link gradient in both directions independently can be useful in asymmetric network setups such as ADSL lines and satellite links.

Comparing the link gradients for the application server-database link, one can clearly see the difference between the small gradient for the web server oriented *PutBidAuth* transaction and the large gradients for the others (database-oriented). The magnitude of the link gradients provides guidance for targeted application optimization, e.g., moving of components with high link gradients closer together or increasing of cache sizes across such links. The table also shows that the link gradient is not a static metric and increases with workload for all links, possibly due to queuing effects. This observation strengthens our claim that an unintrusive runtime technique is desirable for link gradient measurement. Finally, the throughput measurements show that the system throughput changes by less than 5% in all cases, thus demonstrating the low intrusiveness of the technique.

Link	Delay (msec)	View Item	Store Bid	PutBid Auth	Put Bid
Std. Dev.		210.78	160.5	82.46	151.07
AS-WS	24	28.10	28.01	53.82	39.25
DB-AS	6	65.95	94.32	18.9	86.62

Table 5.2: Response Time Perturbation

Another measure of intrusiveness is the increase in response time due to the delays injected during measurement. Those results are shown for a workload of 30 clients in Ta-

ble 5.2. The first row indicates the standard deviation of each transaction’s response time during normal system operation, while the other rows show both the injected per-message delay and the change in response time during the measurement process. All numbers are in milliseconds. Although the noisy PlanetLab environment requires much higher delays for some links than an exclusive environment would, the impact is still within the system’s normal behavior. In all cases, one can see that the additional delay is (sometimes significantly) less than the system’s normal standard deviation. One reason for the larger delays is the high variance Tun/Tap injection mechanism on PlanetLab—based on our experience in local experiments, we believe that using the `ip_queue` injector leads to significantly better results.

5.3 Predictive Power

Next, we use Table 5.1 to evaluate the link gradient’s predictive power. We use the link gradient equation $rt^2 = rt^1 + \sum_{i \in \text{Links}} (l_i^2 - l_i^1) \cdot \vec{\nabla} \bar{rt}(l_i)$ to predict a transaction’s response time rt^2 in a new configuration based on its current response time rt^1 and link latencies l_i^1 , the link gradient, and the latencies l_i^2 in the new configuration. The prediction is compared against a measured response time obtained by actually deployment. To estimate one-way link latency, we compute the mean of 1000 TCP RTTs, and divide it by two.

In the first set of experiments, we kept the workload constant at 30 clients and the locations of the web and database server fixed at UCSD and CMU respectively, but moved the application server to 6 different sites across the US. Figure 5.2 shows the predicted and measured response times for each transaction in each different configuration along with 95% confidence intervals. The confidence interval for the predicted response time rt^2 takes into account the errors introduced due to both the original response time rt^1 and the latency measurements, and is calculated using the equation $\rho(rt^2) = \rho(rt^1) + \sum_{i \in \text{Links}} (\rho(l_i^2) -$

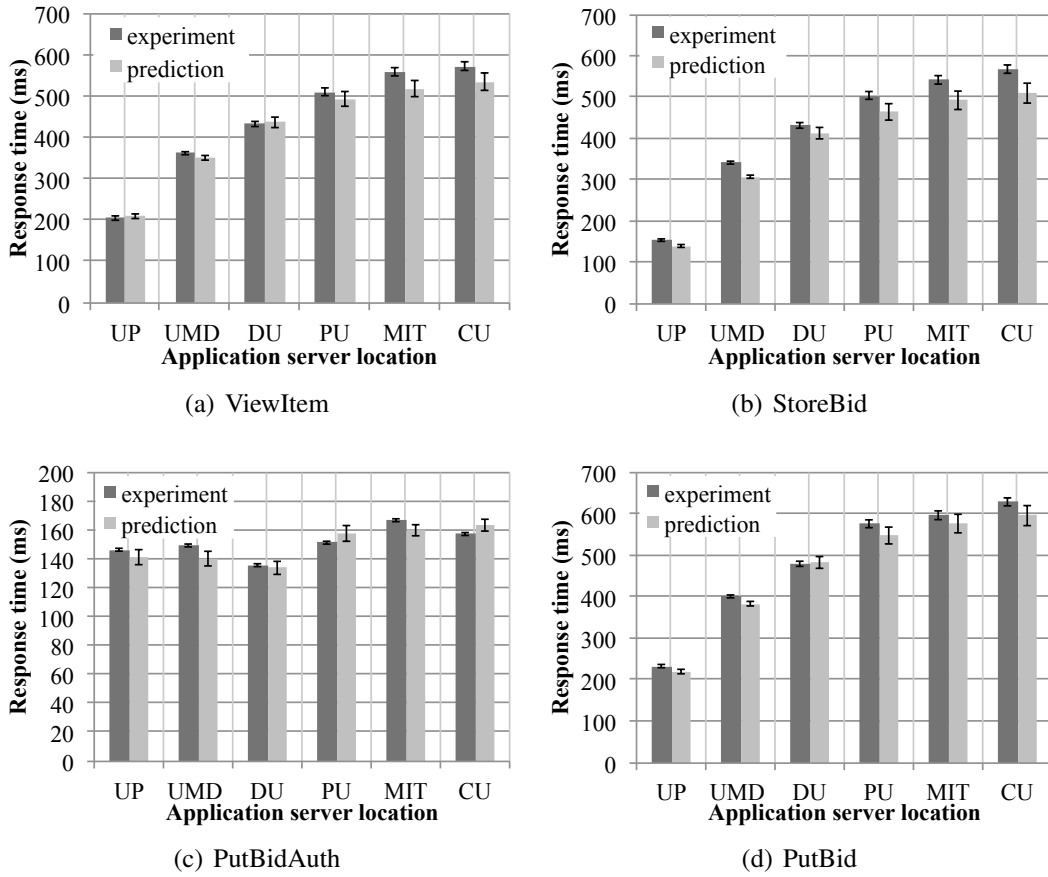


Figure 5.2: Predicting the Effects of Component Placement. Server location key: UP:U.Pitt, DU:Duke, PU:Princeton, CU:Cornell

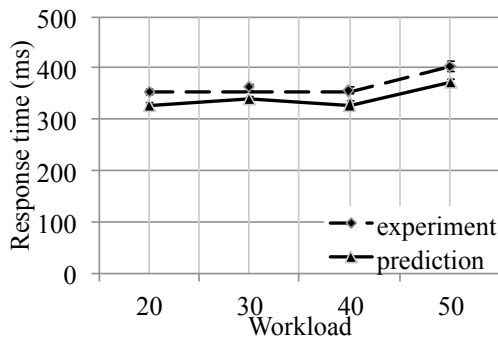
$\rho(l_i^1) \cdot \vec{\nabla} \bar{rt}(l_i)^2$, where $\rho(rt^2)$, $\rho(rt^1)$ and $\rho(l_i)$ are the variances in the original response time and latency measurements, respectively.

The results show good agreement between the predicted and measured response times across all transactions, which suggests that the link gradient is able to accurately capture the effect of link latency changes on application response time. Although most results are within the margin of error, there is a small but systematic increase in error as the difference between old and new link latencies increases¹. The reason is that increasing latency may cause an increase in non-linear queuing effects, especially if a system is heavily loaded, thus making the link gradient, which is a linear metric, increasingly optimistic. We discuss the practical implications in detail in chapter 6, but note that up to the nationwide scales we tested, the metric shows excellent accuracy.

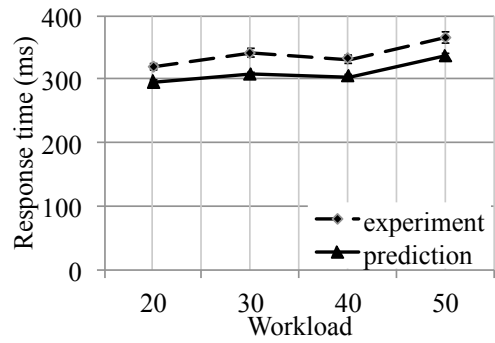
In the next set of experiments, we kept the web and database server unchanged at UCSD and CMU respectively, changed the location of the application server to UMD (University of Maryland), and measured the predictive ability of the link gradient as the workload was changed from 20 to 50 concurrent clients. The results presented in figure 5.3 show that as the workload increases, the link gradient is able to track the changes in response time across all the transactions to within the limits of experimental error. Note that because the link gradients are computed independently for the various workloads, the predictions do not suffer from any systemic errors due to higher-order effects as a result of increasing workloads.

The results show that the link gradient is a useful predictive tool that can be used by system administrators to evaluate alternative system component placements under varying workload conditions.

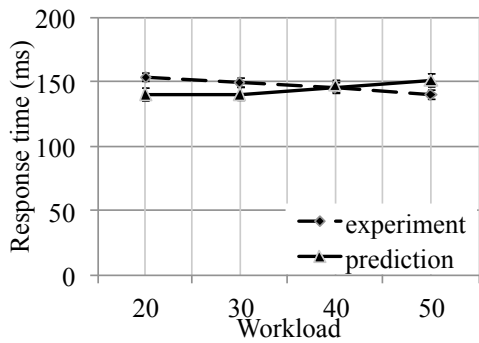
¹The configurations are arranged in increasing order of latency between the application server and database sites.



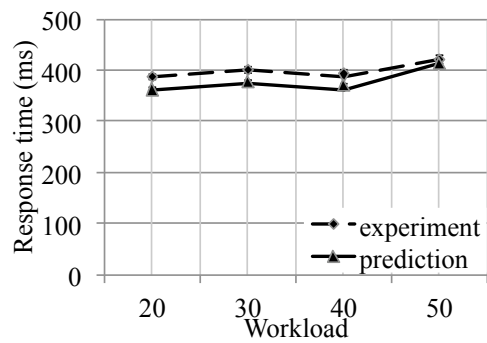
(a) ViewItem



(b) StoreBid



(c) PutBidAuth



(d) PutBid

Figure 5.3: Prediction results under different workload

Link	ViewItem	StoreBid	PutBidAuth	PutBid
AS-WS	2.20	0.98	1.69	1.93
DB-AS	2.87	5.08	0.66	5.87

Table 5.3: Link Gradients with Connection Pooling

5.4 Communication Pattern Variations

So far we have shown that the link gradient works for a conventional request response type application. However, real enterprise systems typically have many different types of communication patterns such as synchronous and asynchronous calls, load balancing, and connection pooling. Next, we examine if the link gradients accurately capture the effect of link latency on response time across such variations.

Connection Pooling: *Connection pooling* is a technique used to optimize networked applications by recycling connections shared among different requests, thus altering the application’s communication patterns. With connection pooling enabled, requests can use existing connections between the AS and DB rather than initialize a new connection every time. When we computed the link gradients for a workload of 60 clients after enabling connection pooling, we obtained the link gradients shown in Table 5.3. Comparing the gradients with those for the default setup in Table 5.1, we see as predicted that while the gradient on the AS-DB link for the web and application server centric *PutBidAuth* transaction remains relatively unchanged, the link gradients for the other transactions are substantially reduced. Figure 5.4(a) shows the predicted vs. measured response time results in a new configuration (with the app server moved from CMU to UMD) both with and without connection pooling. As can be seen from the figure, the predictions match the measured response times to within error tolerances.

State Replication: Next, we constructed a scenario with two application servers configured to perform passive session-state replication for fault tolerance purposes. The server AS1 is designated as the primary and is forwarded all the requests by the web server. The

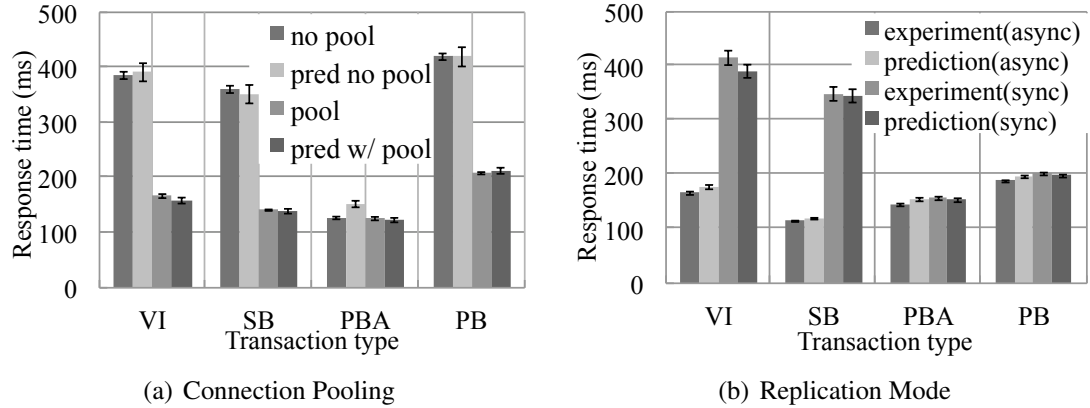


Figure 5.4: Communication Pattern Effects

Sync	ViewItem	StoreBid	PutBidAuth	PutBid
AS1-AS2	4.18	4.03	0.15	0.11
AS2-AS1	4.68	5.24	0.12	0.35
Async	ViewItem	StoreBid	PutBidAuth	PutBid
AS1-AS2	0.14	0.05	0.10	0.11
AS2-AS1	0.03	0.03	0.02	0.04

Table 5.4: Link Gradients for Replication Modes

application server AS2 is designated as the backup and only receives state updates from AS1. Tomcat allows replication to proceed either synchronously, such that requests do not return to the caller until state transfer to the backup is complete, or asynchronously, such that requests can return before state transfer is complete. Since RUBiS does not use session state, we modified the *ViewItem* and *PutBid* transactions to store dummy session state, but left the other transactions untouched. Then, we computed link gradients using both synchronous and asynchronous replication modes with the web server in UCSD and the database in CMU as before, but with both application servers placed at the same site (Cornell) because of the replication modes' multicast requirements.

Table 5.4 shows the computed link gradients for the link between the primary and backup Tomcat servers under both synchrony settings and in both directions. As shown, the link gradient can clearly distinguish between the two replication modes for the modified transactions. Furthermore, as expected, the synchronous mode has much larger link

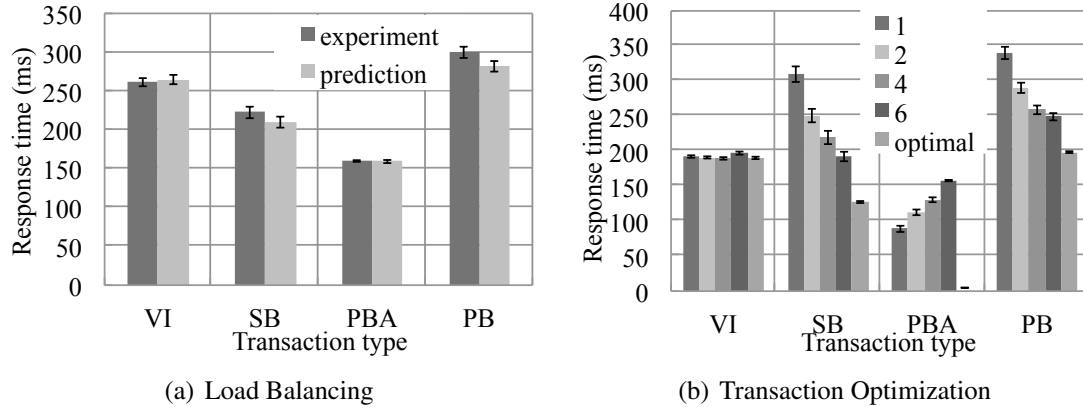
gradients than the asynchronous mode. However, the asymmetry between the forward and reverse link gradients on the primary-backup link in the asynchronous mode is puzzling as it is the only such asymmetry we discovered in the entire application. We speculate that the reason for the slight dependency of the response time on the AS1-AS2 link is because of interference effects, possibly due to locking, between the threads handling the request and the communications thread responsible for sending the session state to the backup replica.

Figure 5.4(b) shows the predicted vs. measured response times when the link latency between the two application servers was increased by 20 ms (we could not move the servers to a different location due to multicast requirements.) As can be seen, the predictions match the experimental results quite well for all the transactions, showing that the link gradient is able to accurately capture the effects of different communication patterns without requiring any prior information about them.

Load Balancing: The last communication pattern we consider is uniform load balancing using Apache's *mod_jk* module across two identical application servers without any state replication. We computed the link gradients for this scenario with the web server in UCSD, and the two application servers and the database located at CMU. The results are shown in Table 5.5. Comparing the results with the link gradients for the unreplicated service shown in Table 5.1, it can be seen that the link gradients are roughly halved (compared to the gradients without load balancing) for *all* the links, not just the ones between the web server and the application servers. The reason is that the link gradient measures the average effect of changes over time rather than measuring on a per-flow basis. Therefore, the reduction in transaction flows on a link due to upstream load-balancing is reflected as a reduction of the link's impact on the mean response time, thus reducing the link's gradient. That behavior is very different from probabilistic causality graphs such as those constructed in [?]. In any case, figure 5.5(b) shows that the predicted vs. measured response times show excellent

Link	ViewItem	StoreBid	PutBidAuth	PutBid
AS1-WS	1.07	0.60	1.03	0.97
AS2-WS	1.00	0.55	0.76	0.76
DB-AS1	7.17	9.31	0.09	8.58
DB-AS2	7.13	8.96	0.25	8.49

Table 5.5: Link Gradients with Load Balancing



agreement for all transactions when one of the application servers is moved from CMU to UMD.

The results validate our hypothesis that even though the link gradient is a simple metric that captures linear effects, it provides an accurate predictor of the effects of link latency on response time across various workloads, configurations, and communication patterns without requiring much a priori knowledge about the system architecture.

5.5 Per-Transaction Optimization

Link gradients can be used to optimize a system for responsiveness at transaction granularity without detailed application knowledge. Although different transaction types have different optimal locations for the AS (i.e., close to the clients vs. close to the database), the WS can route transaction types differently. Therefore, we can obtain an optimized configuration by having *both* local and remote AS copies, and using the link gradient to choose which transactions are to be routed to each server. Transactions whose WS-AS gra-

dients are higher than their AS-DB gradients are routed to the local server (the *PutBidAuth* transaction), while all other transactions are routed to the remote server.

We conducted experiments using such a setup, with the web server and one application server at UCSD, and the database and the other application server at CMU. To compare, we also conducted experiments in which all requests were load-balanced between the two servers without regard to the transaction type, but using various load-balancing ratios (the best that can be done without application-specific information). Figure 5.5(b) shows the measured response time when all the configurations were subjected to a 60 user workload. In the figure, a configuration n means that the requests were routed to the local and remote servers with a ratio of 1: n . The results show that no constant load-balancing ratio can achieve the optimal results that are achieved through the per-transaction load-balancing based on link gradients.

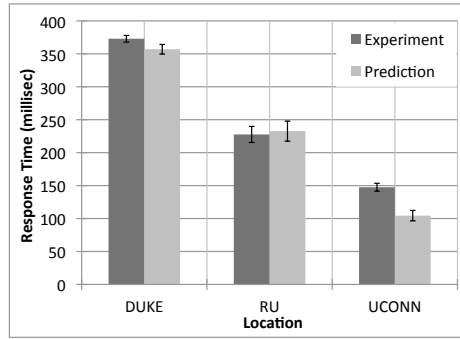
5.6 Optimization for Multiple Applications

Link gradients can also be used to optimize deployments consisting of multiple applications that share components. For example, consider an ACDN-like scenario [7] in which a company has two applications that share a common back-end database. To optimize the response time for static content, it is willing to use third-party cloud computing solutions to host the front-ends close to the clients, but for privacy and security reasons, wants to host the database in its own data centers. The decision of where to place the database can be affected by how each application uses the database (e.g., synchronous, asynchronous, stored procedure calls, caching, connection pooling) and the application workloads relative to each other. Link gradients can be used to determine at which of many possible locations the back-end should be placed so as to maximize the responsiveness across both applications.

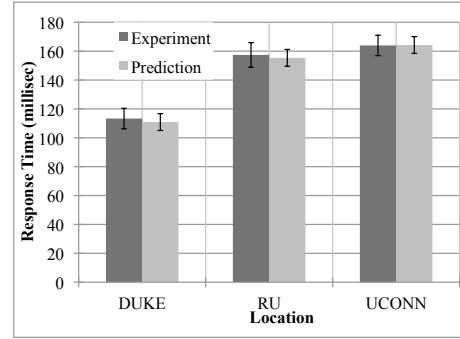
To demonstrate this scenario, we use RUBiS and the PHP version of RUBBoS [4], a bulletin board benchmark, as the two applications. We deploy the front-ends for RUBiS (WS and AS) at MIT and for RUBBoS (WS only) at USF (U. South Florida), and consider 3 different locations for the back-end database. The response time at each of these possible locations is predicted using the link gradients measured on each application running independently, with a workload of 20 clients each. The link latencies associated with the candidate configurations are measured in the field. Figure 5.5 shows the measured average response time over all transactions and the predicted response time over all transactions for each application, with the database at each of the possible locations.

The results show that the Java-servlet-based RUBiS application is much more sensitive to database server placement than the PHP based RUBBoS application (because PHP processing in the web server is a bottleneck for RUBBoS). The predicted response times, when averaged using the relative application importance as weights, can be used to make a decision on the best location. More importantly, the results also show that link gradient measurements conducted in isolation for each application can be used to predict, with good accuracy, the response times in shared scenarios. The only experimental measurement that did not fall within the confidence interval for the prediction was for RUBiS when the DB was located at U. Conn. Upon further investigation, we discovered that the reason was that the PlanetLab server used at U. Conn. was a much slower machine, with a processor speed of 2.0GHz (compared to the 3.0GHz machines used in all the other nodes).

By predicting per-transaction response times, the gradients also support reconfiguration of the system if user behavior patterns (i.e., the fraction of each transaction type) change. Although the opportunities for optimization in these applications are limited, the results pave the way for link gradients to be used for response time optimization in larger systems in conjunction with search algorithms that allow a systematic exploration of the space of a



(c) RUBiS



(d) RUBBoS

Figure 5.5: Predicting the Effects of Database Placement for RUBiS and RUBBoS. (RU: Rutgers, NJ and UConn: U. Connecticut)

large number of deployment configurations. Testing the predictions with large production systems, especially those running at AT&T, will be part of our future work.

CHAPTER 6

Limitations

Although the link gradient metric and our measurement algorithm have several strengths, there are some limitations. The first and most obvious limitation is the need for injection of perturbations in the target system—a requirement that limits its appeal for some critical applications. However, it is this very characteristic that provides the technique with some of its strengths such as the ability to isolate effects at the individual transaction level and the ability to perform direct measurement without intermediate models. Therefore, this limitation cannot be eliminated, but could be alleviated. Although the spectral approach provides a good start, we believe there is ample scope for further improvements in minimizing the perturbations required for accurate measurements.

A more subtle limitation is related to nonlinearity. Since the link gradient only captures the *linear* relationship between response time and latency, nonlinearities caused by factors such as timeouts and queuing effects (increased link latency may increase the service time of upstream nodes) are ignored. While our results demonstrate that linearity holds and the results are accurate over a wide range of scenarios, that may not be the case in systems that are very heavily loaded. For example, figure 6.1 shows how response time varies with increasing service time (captured traffic intensity) in an $M/M/1/K$ finite-buffer queuing station with a fixed workload. Two link gradient measurements are shown, along with a representation of the perturbations they introduce in the system (the intervals represented by black bars) are shown. The link gradient measures effectively compute the tangents at each

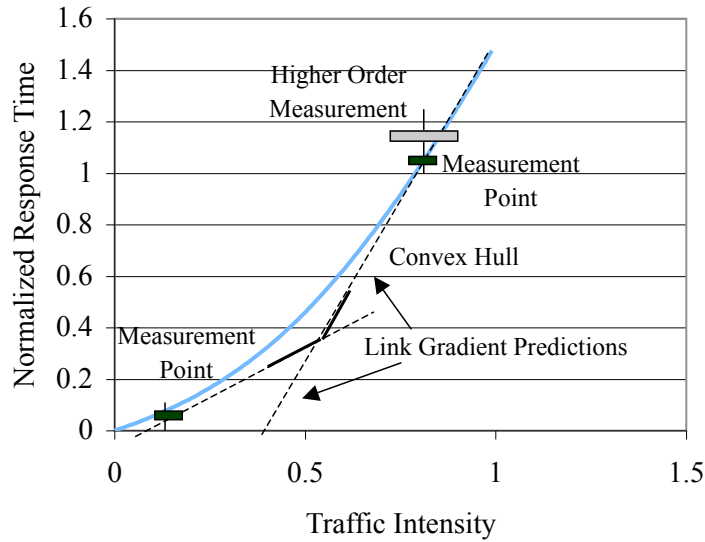


Figure 6.1: Effect of nonlinearity on Link Gradients

measurement point, and thus, over large latency changes, may fail to capture nonlinearity in the response time.

We outline two approaches for this problem. First, the link gradient definition can be expanded to include higher-order terms. During the measurement of the link gradient, perturbations with multiple magnitudes are then introduced (as shown by the gray bar in the figure), and the additional measurements are used in a regression to compute the coefficients for the higher order terms. Another solution is to recompute link gradients whenever a prediction turns out to be inaccurate, and maintain a set of link gradients whose convex (or concave in the case of gradients that decrease with increasing latency) hull as shown by the heavy lines in the figure can be used to improve predictive capability over large changes in link gradients. Based on our experience, we believe that a very small number of link gradients would be sufficient to provide acceptable accuracy over a system's entire range of operating conditions.

CHAPTER 7

Related Work

In literature, there has been work on measuring and profiling different aspects of distributed applications for debugging, optimization, modeling, and failure diagnosis. For example, [8] evaluates the impacts of communication overhead and network latency on the performance of parallel applications on tightly coupled clusters by running benchmark applications on an experimental platform where the overhead and network latency can be controlled.

Critical path analysis for parallel program execution was introduced in [9] and extended to an on-line version in [10]. The critical path is the longest path in the program activity graph (PAG) that represents program activities (computation and communication), their durations, and their precedence relationships during a single execution of the parallel program. While the critical path can be used to guide debugging and performance optimization in parallel programs, it cannot realistically be used to predict the impact of network latency change on the response time of multitier services.

Causal paths indicate how end-user requests flow through system components, and have been used to understand and analyze distributed applications' performance and to identify bottlenecks [11]. A number of techniques for determining causal paths have been proposed [11, 12, 13, 14, 15, 16, 17], each with its own advantages and disadvantages in terms of assumptions on communication patterns, accuracy, and execution cost. Although the problem would by no means be trivial, it is conceivable that placement of some restrictions on communication patterns would make it possible to use causal paths to compute the “mean number of message crossings in the path of the system response” (described in

chapter 2), and thus approximate link gradient. However, causal paths cannot capture the effects of increased link latency on other parts of the system (e.g., queuing) and thus cannot measure link gradient exactly. Of the literature on the determination of causal paths, only Magpie [13] collects enough information about resource usage along paths that detailed response time modeling might be attempted. However, the need for extensive (albeit lightweight) instrumentation precludes Magpie’s use by hosting providers, such as AT&T that often do not have the required access to the applications they host.

Signal-injection-based techniques have been used by others, mostly for determining failure dependencies. The ADD (Active Dependency Discovery) technique determines failure dependencies by active perturbation of system components and observation of their effects [18]. The ADD approach is generic and does not specify the perturbation and effect measurement methods. In [19], the ADD approach is used with fault injection as the perturbation method. The Automatic Failure-Path Interference (AFPI) technique combines pre-deployment failure injection with runtime passive monitoring [20]. While our technique could be seen as a special case of ADD, our technique is far less disruptive to the service provided and can thus be used in running production systems. Delay injection for disk and network access events is used in [15] to verify causal dependencies between such events in a component-based system. Specifically, [15] uses this technique to determine the object read and write policies in a commodity-based commercial storage cluster. However, the technique is strictly off-line and requires full control of the system workload (including message sizes, types, and frequency). Finally, while Fourier analysis has been used by others to detect periodic behavior in network routing updates [21], we are not aware of any other work in software systems research that uses the specific combination of signal injection and Fourier analysis to improve measurement accuracy.

CHAPTER 8

Conclusion and Future Work

In this paper, we introduced a new metric, the *link gradient*, that can be used to approximate how the response time of a system changes with link latencies. We proposed a novel technique to compute the link gradients by using square wave signal injection and Fast Fourier Transform with low perturbation. We implemented an automatic framework for computing the link gradients of a running multitier enterprise application. We demonstrated the efficiency and accuracy of our technique by using micro-benchmarks as well as distributed deployments of RUBiS on PlanetLab. We showed that link gradients can be used to predict service response times for new configurations.

Our future work will consist of evaluating alternative techniques for injecting signals in distributed systems as well as identifying other new system metrics.

REFERENCES

- [1] J. N. Albert M. Lai, “On the performance of wide-area thin-client computing,” *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 2, pp. 175–209, May 2006.
- [2] G. Chafle, S. Chandra, N. Karnik, V. Mann, and M. Nanda, “Improving performance of composite web services over wide area networks,” in *Proc. 2007 IEEE Congress on Services (SERVICES 2007)*, July 2007, pp. 292–299.
- [3] E. Cecchet, J. Marguerite, and W. Zwaenepoel, “Performance and scalability of EJB applications,” in *Proc. OOPSLA’02*, 2002, pp. 246–261.
- [4] ObjectWeb Consortium, “RUBBoS: Bulletin board benchmark,” <http://jmob-objectweb.org/rubbos.html>, Feb 2005.
- [5] G. Varghese and A. Lauck, “Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility,” *IEEE/ACM Trans. on Networking*, vol. 5, no. 6, pp. 824–834, 1997.
- [6] Transaction Processing Performance Council, “TPC benchmark W (web commerce) specification, v.1.7,” www.tpc.org/tpcw/spec/tpcw_17.pdf, Oct 2001.
- [7] M. Rabinovich, Z. Xiao, and A. Aggarwal, “Computing on the edge: A platform for replicating internet applications,” in *Proc. 8th Int. Workshop on Web Content Caching and Distribution*, Sept 2003.
- [8] R. Martin, A. Vahdat, D. Culler, and T. Anderson, “Effects of communication latency, overhead, and bandwidth in a cluster architecture,” in *Proc. ISCA’97*, 1997, pp. 85–97.
- [9] C.-Q. Yang and B. Miller, “Critical path analysis for the execution of parallel and distributed programs,” in *Proc. 8th Int. Conf. on Distributed Computing Systems*, 1988, pp. 366–373.
- [10] J. Hollingsworth, “Critical path profiling of message passing and shared-memory programs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 1029–1040, 1998.
- [11] B. Miller, “DPM: A measurement system for distributed programs,” *IEEE Trans. on Computers*, vol. 37, no. 2, pp. 243–248, 1988.

- [12] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. DSN'02*, 2002, pp. 595–604.
- [13] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in *Proc. OSDI'04*, Dec 2004, pp. 259–272.
- [14] M. Aguilera, J. Mogul, J. Weiner, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. SOSP*, Oct 2003.
- [15] H. Gunawi, N. Agrawal, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and J. Schindler, "Deconstructing commodity storage clusters," in *Proc. ISCA'05*, June 2005, pp. 60–71.
- [16] P. Reynolds, J. Wiener, J. Mogul, M. Aguilera, and A. Vahdat, "WAP5: black-box performance debugging for wide-area systems," in *Proc. Int. WWW Conf.*, May 2006.
- [17] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, "E2EProf: Automated end-to-end performance management for enterprise systems," in *Proc. DSN'07*, June 2007.
- [18] A. Brown, G. Kar, and A. Keller, "An active approach to characterizing dynamic dependencies for problem determination in a distributed environment." in *Proc. 7th IFIP/IEEE Int. Symp. on Integrated Network Management (IM 2001)*, May 2001, pp. 377–390.
- [19] S. Bagchi, G. Kar, and J. Hellerstein, "Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment," in *Proc. 12th Int. Workshop on Distributed Systems: Operations & Management*, Oct 2001.
- [20] G. Candea, M. Delgado, M. Chen, and A. Fox, "Automatic failure-path inference: A generic introspection technique for internet applications," in *Proc. 3rd IEEE Workshop on Internet Applications (WIAPP)*, Jun 2003.
- [21] C. Labovitz, R. Malan, and F. Jahanian, "Internet routing instability," in *Proc. SIGCOMM '97*, 1997, pp. 115–126.