Service Management Using Application Performance Prediction

Jim Martin, Shobhana Kirtane
Department of Computer Science, Clemson University
jim.martin, skirtan@cs.clemson.edu

Abstract

Application performance prediction is an integral component of the control loops that form the backbone of an autonomic system. As an example, a self-managing system must be able to predict if a dynamic resource request will meet service level objectives. In this paper we propose and validate a service management architecture for a web services environment. The architecture allows a client application to interact with a service manager, request a performance prediction for a web service, execute the web service, and have the client return the actual performance to the service manager. The service prediction takes into account both network delays as well as server delays. Both analytic and time-series based models are employed. We report on our progress to date in the development and usage of such models in a WebSphere-based service management application.

1. Introduction

Corporate IT professionals struggle with complex and evolving information systems on a daily basis. Autonomic computing addresses the complexity of today's IT environment and is positioned to meet future on demand environments by providing self-managing systems. Application performance prediction is an integral component of the monitor, analyze, plan and execute control loops that form the backbone of an autonomic system. As an example, a self managing system must be able to predict if a resource request can be fulfilled while still meeting service level objectives. As Figure 1 suggests, application performance prediction, performance monitoring and resource management are fundamental to service management.

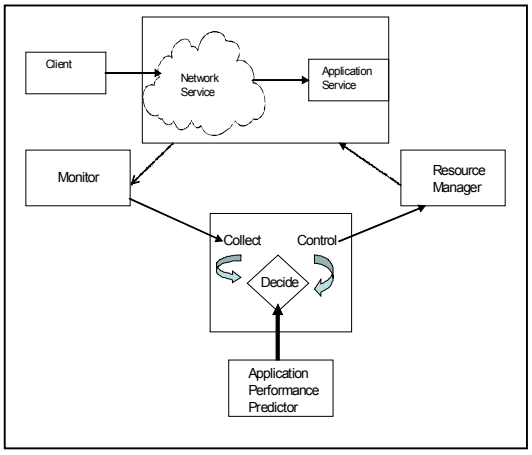

Figure 1. Autonomic control loop

The current Web Services framework consisting of the Web Services Description Language (WSDL), the Universal Discovery, Description and Integration (UDDI) service registry and the Simple Object Access Protocol (SOAP) supports e-business on the Internet [1,2,3,4,5,6]. Autonomic computing is synergistic with Web Services in that both facilitate the deployment of systems that require minimal human intervention. Methods and frameworks for SLA management in a Web Services environment have been proposed [7,8,9]. However much of this prior work has focused on creating language constructs and systems that automate the definition, negotiation and enforcement of SLAs. There has not been a significant effort to couple Web Services management with performance prediction.

In this paper we propose and validate a service management architecture for a Web Services environment that is based on application prediction. A client application interacts with the service manager, requests a performance prediction for a web service, executes the web service, and then informs the service manager of achieved performance. We focus on transactions and we assume that the service level

construct presented to a client is a simple response time making the SLA language trivial. The service prediction takes into account both network delays and server delays. Both analytic and time-series based models are employed.

We further show that the proposed service management architecture facilitates the deployment of autonomic capabilities. For example, assume that an Autonomic Element (which consists of an autonomic manager and managed elements [10,11]) is responsible for the adequate provisioning of a system to meet service level objectives. The service manager can accept a client request, and based on client performance requirements and predicted results, either accept (possibly with system reconfiguration), renegotiate or decline the request. Given a system that provides resource provisioning controls, our architecture can be readily adapted for this environment. We use a more practical application of our service management architecture. We develop an Autonomic Element whose function is to detect malicious Denial-of-Service attacks. Once detected, the Autonomic Element can take appropriate action including adjusting firewall rules, rate-limiting certain incoming or outgoing traffic, switching server IP addresses or simply generating an alert.

The remainder of the paper is organized as follows. In section 2 we present the service management system architecture. In section 3 we discuss the analytic and time-series models used for application performance prediction. In section 4 we present the results of our analysis of the system. In the last section we summarize conclusions and future work.

2. System Architecture

Figure 2 illustrates the system architecture. Client applications request a service from a Web Services provider. The client asks the service manager for the predicted performance. The service manager interacts with the system and, based on current conditions, computes and returns a performance prediction to the client. The client executes the service and optionally sends the achieved performance results back to the service manager.

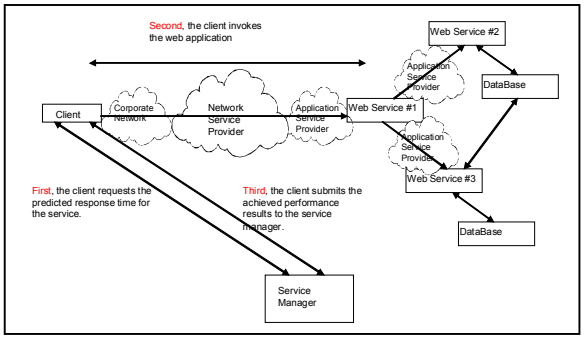

Figure 2. System architecture

To simplify the language used to describe service performance objectives, we classify services by server type. Web servers, FTP servers, database servers, e-commerce servers, audio-video servers and compute servers are best assessed using response times. FTP servers are assessed with throughput, audio-video servers are assessed by packet delay and loss and compute servers are assessed using an instructions per second metric. In this paper we focus on transaction oriented services where, regardless of the complexity of the transaction, the performance construct at the client reduces to a single response time measure. The simplest service is an HTTP GET operation from a Web server. We refer to this as a single level transaction. A more complex application is a Web Service that involves multiple servers (as illustrated by Figure 2). We will refer to the target service as the target Web Service in this paper even though the service might be an HTTP transaction. In a complex Web Service, we refer to all additional services that are invoked in response to the top level target Web Service as embedded services. We also use the term transaction interchangeably with service.

Figure 3 illustrates a model of the transaction illustrated in Figure 2. A single request to the target server $S_{1,1}$ from the client involves two levels of additional transactions. We model the total delay by decomposing the Web Service into individual transactions each having a network and server delay component. The service can be modeled as a form of a graph[1]. The root represents the client and the first descendent of the root represents the target Web Service. All other vertices in the graph represent further transactions that are invoked to fulfill the original client request. The tree edges represent the services at the end of a chain of transactions. A cost

[1] The transaction model is not a true tree since there can be cycles in the graph.

(i.e., in terms of predicted delay) is computed for each edge. By finding the breadth-first search tree, the total delay will be the cost of the path with the maximum cost [12]. The graph tree model assumes that a Web Service calls all embedded services concurrently. Since the services compute their delay (at the request of the service manager), it is trivial for a service to take into account synchronization of embedded services. In this case however, the total delay will not be the cost of the maximum delay path.

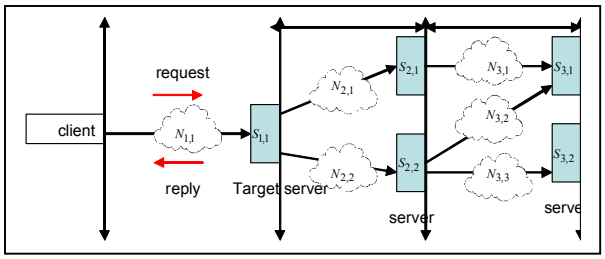

Figure 3. Transaction model

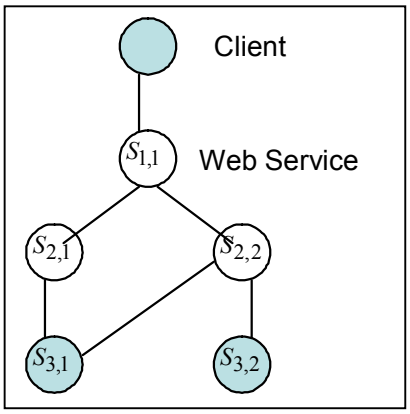

Figure 4. Graph model of a Web Service

In our system, the total transaction delay is found by having each individual service in the tree determine the component of delay incurred by local delays and all embedded services invoked. Figure 5 illustrates the components of the service manager. We describe each component separately and then highlight its operation in response to a client request.

- Service Provider: The application service provider that houses the web services used by clients. Each Web Service interacts with the service manager by registering itself.
- Service: A Web Service that implements a client-server transaction. The service must respond to service manager requests for a service delay prediction. To compute its predicted delay, the service will interact with the local server prediction component as well as (recursively) with the service manager to obtain the predicted delays of embedded transactions.
- Client: Prior to executing a particular Web Service, the client requests a performance prediction from the service manager.
- Server Prediction: All servers in the system execute a monitor program that predicts the *serverDelay* for a particular service.
- Network Monitors: Each network has a monitor that obtains performance measures that are required by the Network Prediction component.
- Service Manager: The autonomic manager of the system. The service manager accepts client requests, coordinates system components to fulfill the request and processes the final results from clients based on some overall management function.
- System Control: The interface by which the service manager can adapt the system.
- Service Configuration: When a service starts, it registers with the configuration service to establish itself with the service manager.
- Network Prediction: This component interfaces with network service monitors deployed at network ingress and egress locations to obtain network performance statistics that are needed to predict the impact of the network on the application.

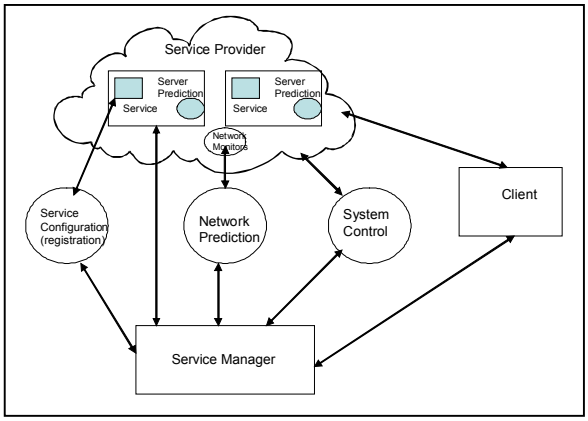

Figure 5. Service manager design

As an example, the following steps occur if the client were to invoke the service $S_{1,1}$ illustrated in Figures 2 and 3.

1. The client contacts the service manager requesting the predicted response time for service $S_{1,1}$.

2. The service manager assigns a unique ID to the client request and contacts the target server, $S_{1,1}$, and requests an estimate of the server delay.
3. The $S_{1,1}$ server is responsible for calculating the delay for all other services it will use. It contacts the service manager (twice) requesting the predicted response time for services $S_{2,1}$ and $S_{2,2}$. The service manager will ask $S_{2,1}$ and $S_{2,2}$ for their respected delay prediction which in turn leads to the system manager asking $S_{3,1}$ and $S_{3,2}$ for their prediction. The number of recursive calls to the service manager depends on the depth of the transaction tree.
4. Eventually the lowest level services (i.e., the edge nodes $S_{3,1}$ and $S_{3,2}$) are able to immediately reply to the service manager's request allowing the chain of prediction requests to the service manager to complete.
5. When the service manager gets a predicted delay back from a server it invokes the network prediction component to predict the impact of the network on the transaction. The network prediction code interacts with the appropriate network service monitors to obtain the necessary data to compute a prediction. The service manager simply combines the network delay with the server delay to generate the transaction delay

3. Prediction Models

The response time prediction for a single level transaction involves two models: a model of how the network impacts the transaction and a model of how server delays impact the transaction. This approach assumes that both components of delay are independent of each other. We rely on the method of *critical path analysis* to justify this assumption. Critical path analysis is a method to identify an application's performance [13]. The method makes use of the observation that only some component activities in a distributed system impact overall response time. In [14], the authors apply the method to a Web server under various levels of load. The results suggest that high server delays can contribute up to 80% of the overall file transfer time for small files. For larger files, the impact of the network tends to dominate regardless of server load. However, as illustrated in Figure 6, for HTTP transactions, the majority of server delay occurs between the arrival of the last packet containing the HTTP GET and the first data packet sent in response by the server. This delay does not impact TCP's RTT estimate nor will it cause packet loss. Based on this result (which we confirmed in our measurements) we model a transaction assuming that the server delay can be added independently to the network delay.

3.1. Network Delay Model

There have been a significant number of analytic models of TCP developed over the past 10 years. One of the earliest efforts modeled TCP's saw-tooth sending rate behavior as a function of *1/sqrt(p)* where *p* is the loss rate [15]. More sophisticated models use different packet loss models (from highly bursty to uncorrelated) and add TCP timeout, connection startup and teardown effects [16,17,18,19]. Based on our earlier work, we observed that each model is best suited to specific environments[20]. The prediction error of each model increases in environments that violate assumptions that have been made. The models are most sensitive to assumptions that were made about packet loss. Although each model represents a significant contribution, none accurately models the wide range of loss dynamics that can exist in a TCP/IP network.

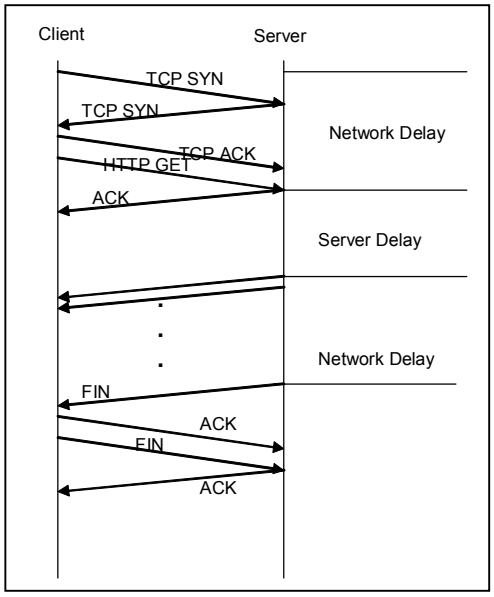

Figure 6. Critical path analysis of a TCP Cx

Building on the work of others (notably Guerin's correlated loss ideas [19]), we developed and validated a response time model of TCP transactions[20]. The model is a stochastic model of TCP that includes the startup time, the impact of slow start, the contribution from steady-state data transfer and the connection teardown. The parameters to the model are shown in equation 1.

(1) response time = f(first packet loss rate, correlated loss rate, RTT, amount of data transferred)

Two packet loss metrics are required so that the level of correlated packet loss can be assessed. The first packet loss rate is the frequency that blocks of loss occur. The correlated loss rate is the probability of consecutive losses following a first loss event. The other parameters are the path RTT and the amount of data to be transferred from the server to the client. If the loss rate and RTT parameters accurately reflect stationary network conditions between the client and server, we found that the predicted response time will be within 15% of the actual response time over a range of network conditions.

3.2 Server Delay Model

As mentioned, the impact of the server delay on TCP transfers can be modeled as an independent delay that can be added to the network delay. In response to client requests, the service manager will request a service delay prediction from the target Web Service and subsequently from all embedded services. A server specific delay prediction component runs on each Web Service machine. The server prediction code includes a model for each type of service (i.e., for each server type such as Web server, Web Services server, database server). We have developed and implemented a delay model that is appropriate for a Web server. We have also applied the model to a single transaction Web Service that is designed to emulate a simple HTTP Get service. We discuss this later in the paper.

We estimate Web server delay, T_{server}, using the following model:

$$T_{server} = W(T_{QueueDelay} + T_{CPUDelay} + (1-H)T_{Disk})$$

$T_{QueueDelay}$: The waiting time from when the request reaches the web server until when the first packet of data is sent. This will depend on current server load but is independent of the size of data associated with the request. We estimate this delay by running an application monitor on the server machine. For a Web server (or a simple Web Service) we run a Unix sockets program that periodically issues an HTTP GET for a small object from the Web server on the localhost. The prediction code takes the difference of the minimum response time from the average of the response time over the last 5 minutes. The majority of the difference reflects the waiting time experienced by client requests.

$T_{CPUDelay}$: The amount of CPU overhead associated with a request. For an HTTP request, the processing time is estimated to be very small (microseconds).

H: The cache hit rate. The web object might be in the drive's cache, the file system's cache or the web server's cache. We assume a 20% aggregate hit rate.

T_{Disk}: The time it takes to transfer the data from disk into memory. Since the performance of modern drives are very comparable, we use the specifications for a commonly used disk drive[2]. We assume an average positioning delay of 13.1 milliseconds and a transfer delay based on a transfer rate of 150MBytes/second.

W: A weighting factor that compensates for inaccuracies in the model caused by system specific behaviors or configurations. For example, through experimentation, we found that a *weight* of 3 was necessary to model the Apache Web server.

4. Network Setup

To validate our system we conducted two sets of experiments. Both sets utilized the network testbed illustrated in Figure 7. The network testbed consists of 2 private networks connected through a router. The machine labeled Dummynet Router is a Dell Optiplex GX260 with 512Mbytes of RAM and is equipped with a single Pentium processor running at 2.4 GHz. It runs FreeBSD 4.8 and uses *dummynet*[21] to control path latency and bandwidth. For these studies, *dummynet* was configured as two unidirectional pipes with a bandwidth of 10Mbps and a latency of 30ms. The capacity of the *dummynet* queues was set to 40 packets each. Both private networks operate at 1Gbps.

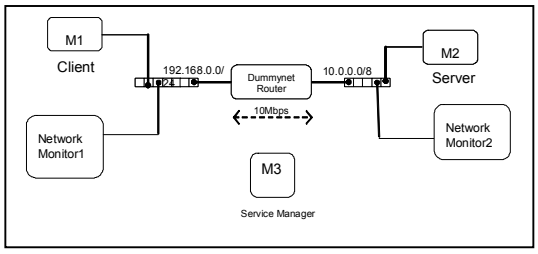

Figure 7. Network testbed

[2] The 80 GB Western Digital Caviar SE 7200 RPM, 8 MB Cache, Serial ATA Hard Drive has an average access time (including seek, settle and latency random delays) of 13.1milliseconds. http://store.westerndigital.com/.

The M1 and M2 machines are Dell Precision 450 workstations equipped with dual 2.4GHz Xeon processors and 1GBytes and 2GBytes of RAM respectively. The M3 machine is a Dell Optiplex GX260 with 800MBytes of RAM. The machines labeled Network Monitor1 and Network Monitor2 are Dell Optiplex GX1p with 128MBytes RAM. All machines run Redhat 8.0 with kernel version 2.4.18-14 compiled for symmetric multiprocessing support. The client is located at M1 and the server is located at M2. IBM WebSphere Studio Application Developer Version 5.1.2 for Linux and IBM WebSphere Application Server Version 5.1 are installed on M1, M2, and M3.

We used the *Surge* (Scalable URL Reference Generator) [22] to generate background traffic. The location of the *Surge* clients depends on the experiment but in all experiments the *Surge* clients interact with the Apache Web Server (version 2.0.4) operating on machine M2. The TCP socket buffer sizes were set to 64Kbytes on all machines. The client is a Unix sockets program that periodically requests a particular Web object from the Web server on machine M2. The standard Ping program is run between the Network Monitor1 and Network Monitor2 machines. The Ping results are used only for the visualization of WAN performance presented in this paper. The network performance data required by the network prediction component is collected by the network monitor programs which also run on these machines. This program is a Unix sockets program that collects the RTT and loss data as required by the prediction server, periodically submitting the results to the service manager.

We conducted two experiments which we refer to as Set1 and Set2. The objective of the experiments was to validate the network and server delay models. To achieve this goal we used the simplest type of server, a Web server. The experiments consist of a client requesting a web object of varying size from the server. Prior to issuing the HTTP Get, the client requests a transaction prediction from the service manager. In the Set1 experiment, in addition to varying the web object size we also vary the congestion level over the emulated WAN link. The objective of the Set1 experiment is to validate the prediction models when the majority of the delay is caused by the network. The WAN utilization is controlled by varying the number of *Surge* clients that run on machine M1. The Set2 experiment is similar except that there is no competing traffic over the WAN. Instead, a varying number of *Surge* clients run on Network Monitor2 resulting in a range of server utilization levels.

The experimental procedure for both Set1 and Set2 is as follows. For a particular run, two parameters are set. The number of *Surge* clients and the size of the Web object that is being transferred from the server to the client. The number of *Surge* clients differs for Set1 and Set2. The size of the Web objects varies from 10Kbytes to 80Kbytes in 10Kbyte increments. A run begins by starting and running the *Surge* clients for roughly 5 minutes. Then the Ping and network monitors are started. After 5 more minutes, the client is started and repeatedly pulls the configured Web object. After another 15 minutes , we stop the test. The mean of the client response time samples represents the actual response time. During the run, network statistics are sent to the service manager which computes the predicted response time.

5. Analysis and Result

In this section we present empirical evidence that demonstrates and validates our models using HTTP transactions. We conclude this section by summarizing the results when we repeat the Set1 and Set2 experiments using a single transaction Web Service.

5.1 Set1 Analysis

Figures 8 and 9 illustrate the WAN performance for the Set1 experiment. A Ping loss monitor ran between the Network Monitor1 and Network Monitor2 machines. The loss rate varied from 0 to 6%. The utilization of the WAN varied from 0 to 100%.

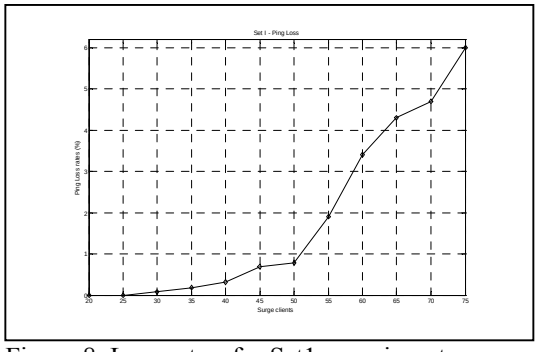

Figure 8. Loss rates for Set1 experiment

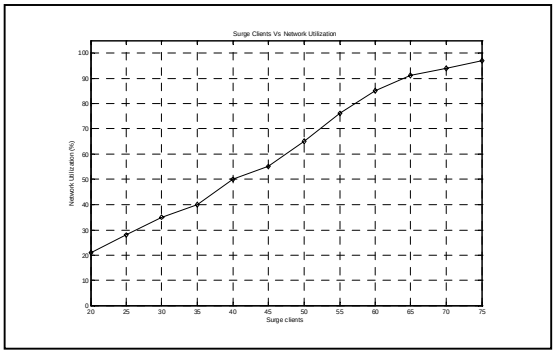

Figure 9. Bottleneck link utilization for Set1

Figure 10 compares the results of the predicted response time to the actual response time for the runs configured to transfer 10Kbytes, 30Kbytes, 60Kbytes and 80Kbytes of data. Excluding the run corresponding to a network load of 70 Web users and a data size of 10KBytes, the error associated with the prediction ranged from 0 to 35%. This range is slightly larger than what we found in our previous results, however the experiments done in this study include a broader range of network conditions. We observe that the prediction error is lowest for the runs where the loss rate was in the range of .5% - 2%. The component of the predicted network delay that models steady state TCP behavior assumes there is loss. The model's 'sweet spot' (i.e., range where it is most accurate) is when the loss rate is in the .5% to 2% range. Outside of this range, TCP implementation differences and configuration settings become significant. We see that the prediction error increases as the loss rate approaches 0. Further, this effect is magnified as the size of the Web object increases which is reasonable since the contribution of the steady state component of the network delay increases with larger Web object sizes.

The run with a prediction error of 69% (the highest error) was configured with a load of 70 Web users and a data size of 10Kbytes. This result is seen for the same load with other data sizes. The contribution of server delay was negligible for all runs (including the 70 Web users runs) as the bottleneck was the WAN. Therefore, the majority of the 69% prediction error for the run at the 70 Web user load level is in the network delay model. We are currently analyzing the model to explain this result.

5.2 Set2 Analysis

In the Set2 experiment, we vary the number of *Surge* clients running at Network Monitor2 to control the load at the server (machine M2). Figure 11 plots the actual and predicted response times for 4 different Web object sizes as the load on the server changes. The X-axis represents the server load and is the average of the first load average number available in */proc/loadavg* sampled by a monitor program that executes during a run. The load average number gives the number of jobs in the run queue (state R) or waiting for disk I/O (state D) averaged over 1 minute.

Server Load	Server Delay Contribution
20	5.1%
30	6.1%
40	20%
50	35%
60	62.5%
70	76%
80	83.5%
90	86%

Table 1: Server delay contribution for 10KByte Web object runs

Figure 11 shows that the prediction error is less than 25% except for the highest server load level. Table 1 shows the contribution of the server delay for each predicted response time in the Set2 10KByte graph. At the highest load level, 86% of the response time is due to the server delays. However at this level of server load, the model underestimates the actual load by a significant amount (50%).

5.3 Denial-of-Service Detection

To demonstrate a practical use for our service management architecture, we show how it could be used to detect a Denial-of-Service attack. We set the Surge client load at machine M1 to 40 (which leads to a loss rate of about .5% over the WAN). We launched a TCP-SYN attack on M2 from Network Machine2. The dynamics created by the attack cause a significant increase in the prediction error. Figure 12 shows the predicted and actual response times during the attack. We also plot the original data from the Set1 experiment which represents the same experiment but without a DoS. The figure indicates that the prediction error increases from 20% to over 150% during an attack.

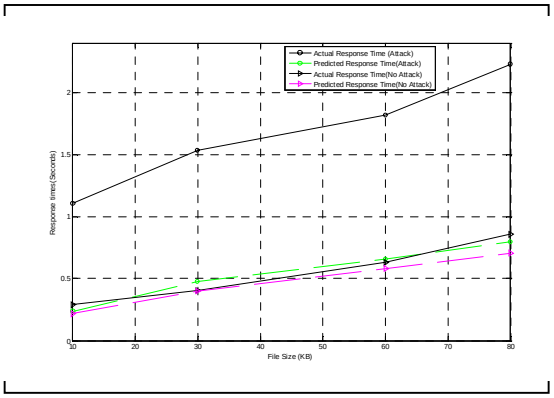

Figure 12. Prediction error during a DoS attack

5.4 Web Service Transactions

We duplicated the Set1 and Set2 experiments using a single transaction Web Service. Once the client code begins executing, it sends a request to the Web Service to transfer a particular data file. When the request is received by the Web Service, it transfers the file to the client. The client requests a prediction for the file transfer from the service manager. The Web Service therefore is identical to the previous Web transaction. The actual and the predicted results were very similar as compared to the Web server results. However, a *weight* value of 2 for the server delay model led to a better fit. In this experiment the Web server was loaded, not the WebSphere application server. We conjecture that if we were to load the server with WebSphere requests, we would see a significant difference in server delay characteristics between Apache and WebSphere.

6. Conclusions and Future Work

In this paper we presented a service management architecture that utilizes application prediction. The system supports client prediction requests of transaction oriented Web Services. The approach models a Web Service as a type of tree. The total service response time is the additive cost of all edges of the path that forms the highest weight breadth-first search tree. The prediction of a transaction assumes that even though server delays are likely coupled to network delays, they can be calculated independently and then combined to obtain the transaction response time.

We prototyped an implementation of the proposed service management architecture. The service manager supports two types of services: HTTP Web object requests and Web Service transactions using the Apache and WebSphere servers respectively. Our results showed that, with some exceptions, the service manager's prediction is within 25% of the actual response time. We also have shown that our service management architecture has potential as a method for detecting and responding to Denial-of-Service attacks.

In the future, we will address the issues identified in this paper. For example, both the network delay model and the server delay model exhibit inexplicably high prediction error levels at the extreme boundaries of our experiments. Also the server delay models need to be more adaptive to better absorb application specific delay characteristics. Finally, we plan on extending the Web Service transaction model to support more synchronization of embedded service requests and computation delays.

10. References

[1] SOAP Version 1.2 Part 1: Messaging Framework, W3C Recommendation, M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. Nielsen, 24 June 2003 (See http://www.w3.org/TR/2003/REC-soap12-part1-20030624/).

[2]SOAP Version 1.2 Part 2: Adjuncts, W3C Recommendation, M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. Nielsen, 24 June 2003 (See http://www.w3.org/TR/2003/REC-soap12-part2-20030624/).

[3]Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Working Draft, R. Chinnici, M. Gudgin, J-J. Moreau, J. Schlimmer, S. Weerawarana, 10 November 2003 (See http://www.w3.org/TR/2003/WD-wsdl20-20031110/).

[4]UDDI Version 3.02 API Specification. Universal Description, Discovery and Integration, uddi.org, June 2001.

[5]UDDI Version 3.02, Universal Description, Discovery and Integration, October 2004, http://uddi.org/pubs/uddi_v3.htm.

[6]A. Keller, G. Kar, H. Ludwig, A. Dan, J. Hellerstein, "Managing Dynamic Services: A Contract Based Approach to a Conceptual Architecture", Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS2002), pp 513-528, April 2002.

[7]A. Sahai, A. Durante, V. Machiraju, "Automated SLA Monitoring for Web Services", IEEE/IFIP DSOM 2002, Montreal, Canada, Oct. 2002.

[8]M. Buco, et. Al., "Managing eBusiness on Demand SLA Contracts in Business Terms Using the Cross-SLA Execution Manager SAM", The Sixth International Symposium on Autonomous Decentrazlized Systems (ISADS03), April 2003.

[9]A. Keller, H. Ludwig, “The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services”, IBM Research Report, RC22456, May 2002.

[10]J. Kephart, D. Chess, “The Vision of Autonomic Computing”, IEEE Software, pp 41-50, January 2003.

[11]A. Ganek, T. Corbi, “The Dawning of the Autonomic Computing Era”, IBM Systems Journal, Vol. 42, No. 1, pp5-18.

[12]T. Cormen, C.Leiserson, R. Rivest, “Introduction to Algorithms”, McGraw-Hill, 1989.

[13]K. Lockyer, Introduction to Critical Path Analysis, Pitman Publishing Company, New York, NY, 1964.

[14]P. Barford, M. Crovella, “Critical Path Analysis of TCP Transactions”, SIGCOMM 2000, pp 127-138, Jan 2000.

[15]M. Mathis, J. Semke and J. Mahdavi, “The Macroscopic Behavior of TCP Congestion Avoidance Algorithm”, in Computer Communications Review, vol. 27, number 3, July 1997.

[16]J. Padhye, V. Firoiu, D. Towsley and J. Kurose, “Modeling TCP throughput: A simple model and its empirical validation”, in SIGCOMM ’98, September 1998.

[17]M. Goyal, R. Guerin, and R. Rajan, “Predicting TCP throughput from non-invasive data”, 2001.

[18]M. Goyal, R. Guerin and R. Rajan, “Predicting TCP throughput from non-invasive network sampling”, in IEEE Infocom 2002.

[19]N.Cardwell, S.Savage and T.Anderson , “Modeling TCP Latency,” in Proceedings of IEEE INFOCOM, Tel Aviv, Israel, March 2000

[20] J. Martin, H. Karlapudi, “Web Application Performance Prediction”, Proceedings of the IASTED International Conference on Communication and Computer Networks, (Boston, MA, Nov, 2004), pp. 281-286.

[21]L. Rizzo, “Dummynet: a simple approach to the evaluation of network protocols,” ACM Computer Communication Review, vol. 27, no. 1 , January 1997

[22]P. Barford, M. Crovella, “Generating Representative Web Workloads for Network and Server Performance Evaluation”, In Proceedings of Performance ’98, ACM SIGMETRICS’98, pp151-160, July 1998.

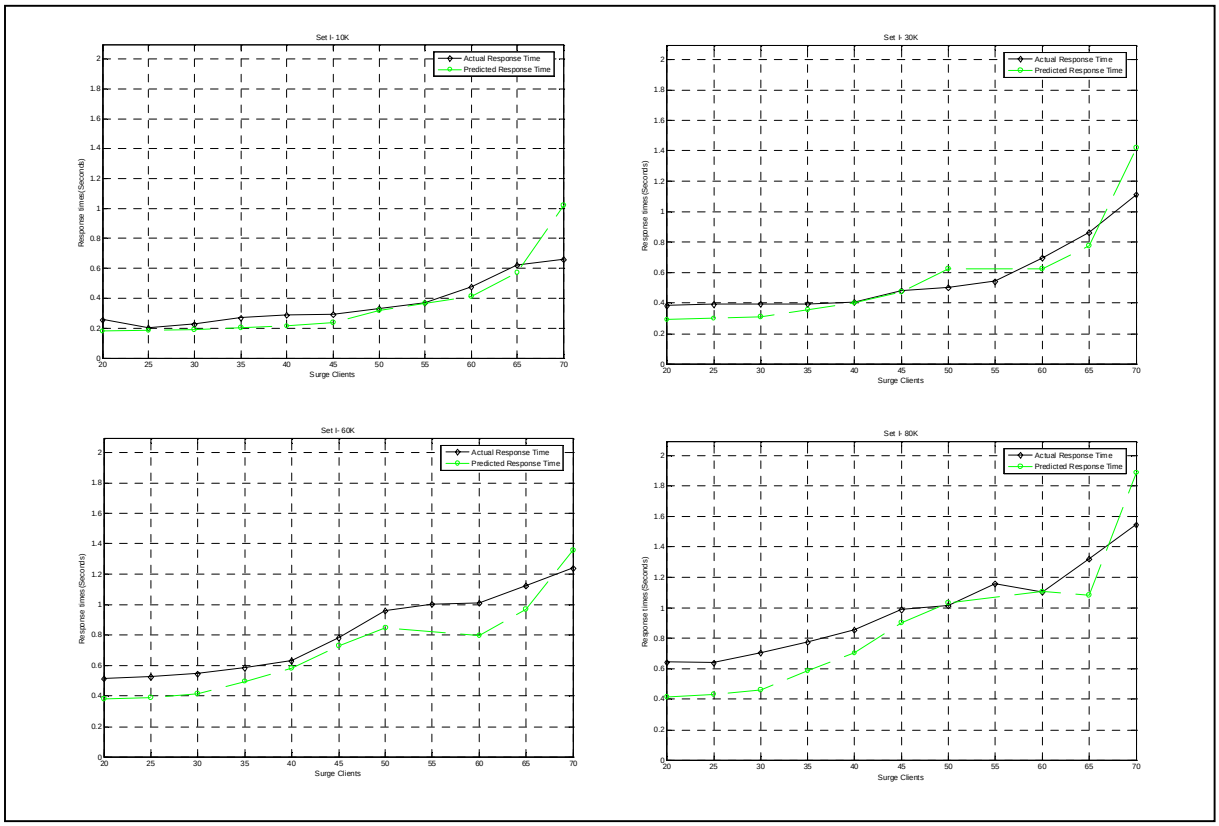

Figure 10. Set1 experiment results

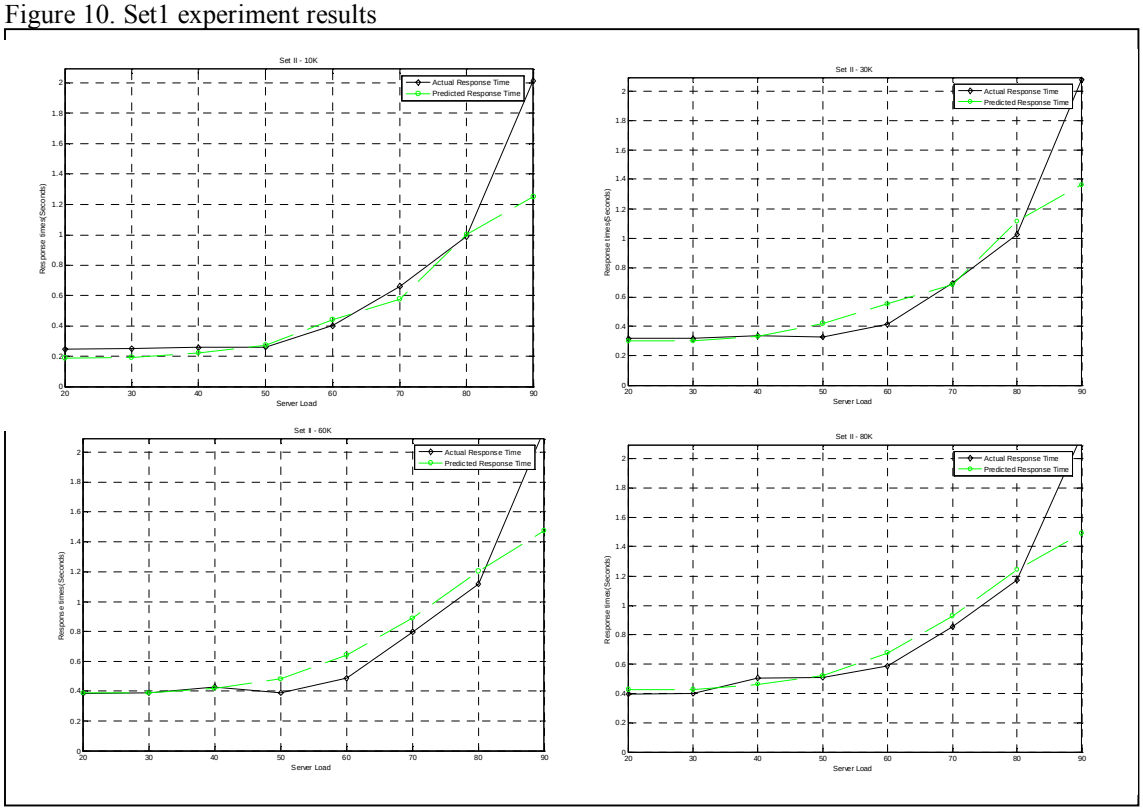

Figure 11. Set2 experiment results