

AutoPerf: An Automated Load Generator and Performance Measurement Tool for Multi-tier Software Systems

M. Tech. Project Dissertation

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Shrirang Shirodkar

Roll No: 05305007

under the guidance of

Prof. Varsha Apte
Computer Science and Engineering
IIT Bombay



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgments

I would like to sincerely thank my guide, **Prof. Varsha Apte**, for her guidance, motivation, as well as the time she spent with me out of her busy schedule discussing the technical matter. I would also like to thank **Vipul Mathur** and **Kautilya Jain** for their invaluable inputs to my work.

Shrirang Shirodkar
CSE, IIT Bombay

Abstract

E-business activities and Web services are essentially real time processes in which performance problems have a high cost. Therefore, capacity planning methodologies should be able to anticipate performance problems for electronic business environments. Measurement-based capacity analysis can be done by generating synthetic load on the system using load generators, profiling the server and taking measurements at the client in a synchronized manner and using the observations for capacity planning purposes. Our tool, *AutoPerf*, can perform capacity analysis of systems in a test environment, when given only the workload description, deployment configuration and user behavior characteristics as input, thus minimizing human intervention in the process of capacity analysis.

Another approach of capacity analysis involves use of performance models. An essential prerequisite for performance modeling is resource usage information about the system. Several modeling tools exist that accept the resource requirements per transaction, the message flows, the workload description and the deployment configuration as inputs and solve an underlying queuing model to provide performance measures. This project automates the process of resource consumption discovery, thus making the process much more smoother, error-free and also facilitating repeatability. The resource consumption details obtained by *AutoPerf* can be fed to a modeling tool to perform capacity planning. The focus of this thesis is on performing measurement-based capacity analysis using minimal inputs.

Contents

1	Introduction	1
1.1	Capacity Analysis	2
1.2	What is a performance measurement tool?	4
1.3	Characteristics of an ideal tool for performance measurement	4
1.3.1	Existing tools for performance measurements	7
1.3.2	Need for an automated capacity analysis and profiling tool	8
1.4	Contributions of this thesis	10
2	AutoPerf: An Automated Load Generator and Performance Measurement Tool for Multi-tier Software Systems	13
2.1	Input Description	13
2.2	Output Description	13
2.2.1	Profiling mode, at a single load level	14
2.2.2	Profiling mode, at multiple load levels	14
2.2.3	Capacity Analysis mode	15
2.3	Architecture of AutoPerf	15
2.3.1	Controller	15
2.3.2	Load generator	16
2.3.3	Profiler	16
2.4	Features of AutoPerf	17
2.4.1	Automation of execution count determination	20
2.4.2	Automated capacity analysis	22
2.4.3	Probabilistic Session Generation	24
2.4.4	Warm-up	27
2.4.5	Dynamic URL generation	28
2.5	Resource requirements of AutoPerf	28
3	Experimental Results	30
3.1	Profiling mode, at a single load level	30
3.2	Profiling mode, at multiple load levels	32
3.3	Capacity analysis mode	32
4	Validation of AutoPerf	38
4.1	Validation using Mercury Load Runner	38
4.2	Self-Validation	38
5	Future Work	41
	Appendices	42

A	AutoPerf: Sample Input File	43
B	AutoPerf: Usage	47
B.1	Profiling mode and at a single load level	47
B.2	Profiling mode and at multiple load levels	47
B.3	Capacity analysis mode	47

List of Figures

1.1	A typical web-based distributed system.	2
1.2	Correlating results of a load generator and profilers manually	5
1.3	Sample workload description	6
1.4	Sample deployment configuration	7
1.5	Snapshot of Load Runner's input[4]	9
1.6	Snapshot of Load Runner's output[4]	9
1.7	Login transaction in <i>SquirrelMail</i>	12
1.8	A very small and static response from Apache web server	12
2.1	Client-side and server-side measurements performed by AutoPerf in profiling mode for multiple transactions	15
2.2	AutoPerf Architecture	16
2.3	Graph of cumulative service demand of a resource as would be obtained by the master (load generator and controller) from the slave (profiler) against execution count i.e. number of times a transaction is repeated, by simulating a transaction with service demand 300 milli seconds.	21
2.4	Graph of true service demand and the observed service demand of a resource as observed by the master (load generator and controller) against execution count i.e. number of times a transaction is repeated, by simulating a transaction with service demand for the resource 300 milli seconds.	22
2.5	A client-server system.	23
2.6	A client-server system with multiple server nodes.	25
2.7	Customer behavior model graph	25
3.1	Deployment configuration of the webmail system.	31
3.2	Throughput and Response time curves obtained using AutoPerf for the <i>WriteEntry</i> transaction in <i>WebCalendar</i>	33
3.3	WebCalendar: Throughput curve obtained by AutoPerf during capacity analysis	36
3.4	WebCalendar: CPU service demand at web server and database server obtained by AutoPerf during capacity analysis	37
3.5	WebCalendar: CPU utilization at web server and database server obtained by AutoPerf during capacity analysis	37
4.1	Throughput curves obtained using AutoPerf and Mercury Load Runner	39
4.2	Response time curves obtained using AutoPerf and Mercury Load Runner	39

List of Tables

2.1	Matrix representation of a customer behavior model graph	24
3.1	Resource usage at Apache web server (<i>httpd</i>) for <i>Login</i> transaction in SquirrelMail	31
3.2	Message sequence chart : <i>Atm_Page</i> transaction in “ <i>Duke’s Bank</i> ”[6]	33
3.3	Simple task elapsed times : <i>Atm_Page</i> transaction in “ <i>Duke’s Bank</i> ”[6]	34
3.4	Used Heap and Non Heap Memory at various load levels : <i>Atm_Page</i> transaction in “ <i>Duke’s Bank</i> ”[6]	35
3.5	Garbage collection statistics at different load levels : <i>Atm_Page</i> transaction in “ <i>Duke’s Bank</i> ”[6]	35
3.6	Customer Behavior Model Graph used during capacity analysis of <i>WebCalendar</i>	36
4.1	AutoPerf results while profiling a <i>Login</i> transaction in SquirrelMail.	38

Chapter 1

Introduction

Over the last few years, the World Wide Web has transformed from being a medium only for distribution of static content to a more interactive one in which responses from a web server are often generated dynamically. Also, it has witnessed a marked increase in the users of web services. Thus, servers today have to handle requests from a large number of users and also which are much more computationally intensive than they used to be earlier.

E-business activities and Web services are essentially real time processes in which performance problems have a high cost. A recent research recommends that retailers should attempt to keep page rendering to no longer than four seconds to avoid site abandonment by customers. Thirty-three percent of dissatisfied online shoppers cite the performance of the web site as the primary reason for their dissatisfaction. The significance of the impact of low user-perceived response time is evident from the fact that forty-two percent of online shoppers with two or more years of online tenure consider quick page loading most important for their loyalty to a site [7]. Frustrated users can translate into lost customers and lost revenue.

As a result, performance monitoring and tuning has become one of the major management activities for application servers. The efforts invested in it is clear from a study which shows that performance monitoring and tuning takes up fifty-five percent of the total system management cost. [14]

Therefore, capacity planning methodologies should be able to anticipate performance problems for electronic business environments. *Service level agreements* which are specified in terms of upper bound on response times, lower bound on throughput and so on, aim at providing performance guarantee. To achieve this, often, applications are hosted on *data centers*. A data center is useful to host critical applications and to lease storage and computational facilities to such applications. Such data centers guarantee availability and performance guarantees to their clients. However, from the data centers' point of view, along with meeting the service level agreements, it is essential that their hardware resources are squeezed to yield maximum value before investing in new hardware. The process of sizing the infrastructure so that service performance requirements are met, is called *capacity planning*[11].

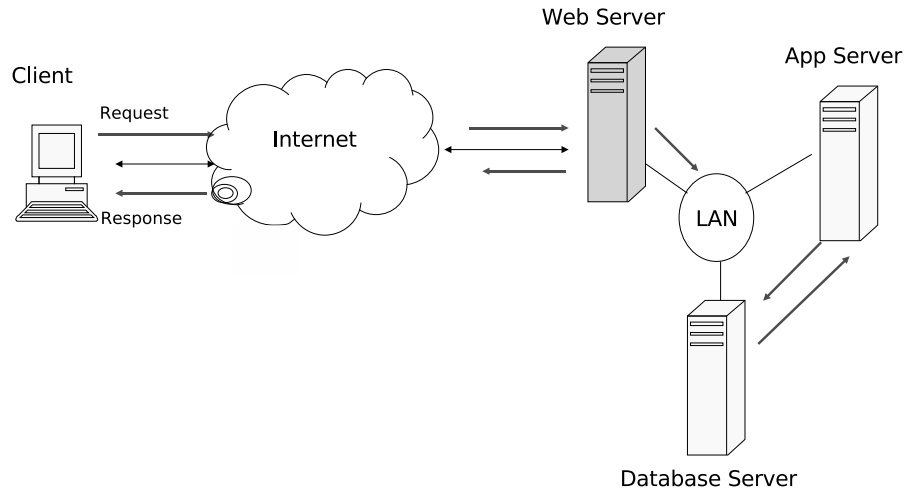


Figure 1.1: A typical web-based distributed system.

1.1 Capacity Analysis

A typical web-based distributed system consists of multiple types of servers. Various servers like web server, application server, database server, authentication server, etc. constitute such systems, an example of which is shown in figure 1.1. Capacity analysis can be used to assess various workload and user behavior scenarios and server configurations and answer what-if questions like - “Should we mirror the web server to balance the load?”, “Should we replicate the database and distribute it across more than one database server?”. Specifically, it is useful to help answer the following questions, given the system deployment information, workload description and the users’ behavioral characteristics.

- How many users can the system support?
- What is the maximum possible throughput of the system?
- What is the average resource utilization for a given load level?
- What is the average response time of a transaction for a given load level?

Answers to these questions help the site owners to determine the hardware utilization of their system. As a result, more hardware can be employed where needed and on the contrary excess resource usage can be avoided. Thus capacity analysis can be employed to reduce the operating costs of the system. Ensuring that the average response time of the transactions is within the tolerable limits of the users, helps achieve user satisfaction. Increased user satisfaction implies greater user loyalty to the site which in turn would aid in earning higher revenue for the site. Thus capacity analysis is a must for large sites and data centers.

Capacity Analysis can be performed by any one of the following two ways or by the combination of both.

Measurement based capacity analysis:

This approach involves generating synthetic load on the system using load generators and obtaining client-side metrics such as response time, throughput, etc. Measurement based capacity analysis involves subjecting the system to load and then obtaining various user-perceived measurements at the client-side. The system can be subjected to a range of *load levels* (i.e. the number of concurrent users accessing the system) and to different workloads (mix of transactions being requested) and the behavior of the system can be observed to study the effect of increasing load on the system. The client-side measurements obtained can help in determining whether or not the service level agreements are met.

Measurement based capacity analysis is useful to characterize the behavior of a server by treating it as a black box. On the contrary, it can also be used to profile the server to obtain various server-side measurements (egs. service demand) which can be used to help solve a performance model of the system.

To perform measurements, it is necessary that the system exists. The system then needs to be altered physically to observe the effects of the new configurations on it. If this approach of physically altering the system and then letting it operate under the new conditions to study the new configuration is cost-effective, then measurements is a good way of performing capacity analysis.

Performance model based capacity analysis:

Performance model, as described in [8], is a representation of a system in terms of logical and quantitative relationships, which can be altered to study the corresponding changes in the behavior of the model, and hence that of the system under study. Once such a model is built, it can then be examined to see how it can be used to answer the questions of interest about the system (like those discussed at the beginning of this section).

Performance models are useful when the actual system does not exist yet, but yet we wish to study it in various proposed alternative configurations, to decide how it should be built in the first place. When using a model, there is always a question of validity of the model, i.e. whether it accurately reflects the system for the purposes of the decision to be made [8].

Also, sometimes performance models may become very complex. Complex protocols, scheduling policies, etc. involved in the system may make modeling a poor choice for capacity analysis. In such cases, measurement based capacity analysis is often a better option.

Solving a performance model of a system often requires information like resource consumption by each of the server-components per request. Obtaining this information requires measurements to be performed on the system under study. Hence, measurement based capacity analysis is extremely useful if it also gathers resource consumption profiles (resource utilization, service demands, etc.) at the servers along with the client-side measures. This aids in solving performance models of the system as well.

Measurements, thus, are necessary irrespective of whether we perform measurement based capacity analysis or performance model based capacity analysis.

Section 1.2 describes a basic performance measurement tool. Section 1.3 explains the desired characteristics of an ideal performance measurement tool. Section 1.3.1 discusses some of the existing performance testing tools and section 1.3.2 elaborates on the need of an automated capacity analysis and profiling tool.

1.2 What is a performance measurement tool?

A performance testing tool is a tool used to accurately test the end-to-end performance of a system before it is actually deployed. The testing process involves generating synthetic load on the system under test by emulating users of that system. For the measurements to be effective, the synthetic load generated must resemble the realistic workload, i.e. that expected to be generated by the real users when the system is in live environment. A performance measurement tool measures various user-perceived measurements such as end-user response times, throughput, number of errors, etc. An effective performance testing tool, should determine maximum throughput achievable, number of users at which it is achieved and so on. Support for generating URLs to be requested dynamically, session-management, etc makes a performance measurement tool very effective. To perform capacity analysis, the tools need to be provided with information about the system itself, the users of the system (think time), workload description, number of virtual users to be emulated, duration for which load is to be generated, etc. A performance testing tool that obtains server side resource consumption statistics is very useful for performance modeling. Such a tool automates what a tester would have otherwise had to do manually with the aid of load generators and profilers separately as shown in figure 1.2.

1.3 Characteristics of an ideal tool for performance measurement

An ideal tool for performance measurement (capacity analysis and profiling) should require as input only the *workload description* of the system under test, the *user behavior description* and the *server deployment configuration*.

The *workload description* of a system can be described in terms of the definitions of the transactions (URLs, etc.) supported by the system, their proportions in the workload at steady state and the load level (number of concurrent users of the system). Figure 1.3 shows a sample workload description. This sample workload description describes a workload consisting of transactions Login_Transaction, Search_Query and Purchase_Item with the corresponding URLs as shown and running in the proportion 6:3:1 in steady state. Such a description should be enough to describe the workload. However, a more effective description of the mix of transactions would be to summarise it in the form of a customer behavior model graph as explained in section 2.4.3.

The system's *user behavior description* would be required by the tool as a summary of the characteristics of the users' system usage pattern. A simple summarisation of the users' behavior would capture only the think time, which is the average time the users of the system remain idle before requesting the next URL. The time within which after submitting a request to the server, if the complete response is not received by the user, the user aborts the site is called

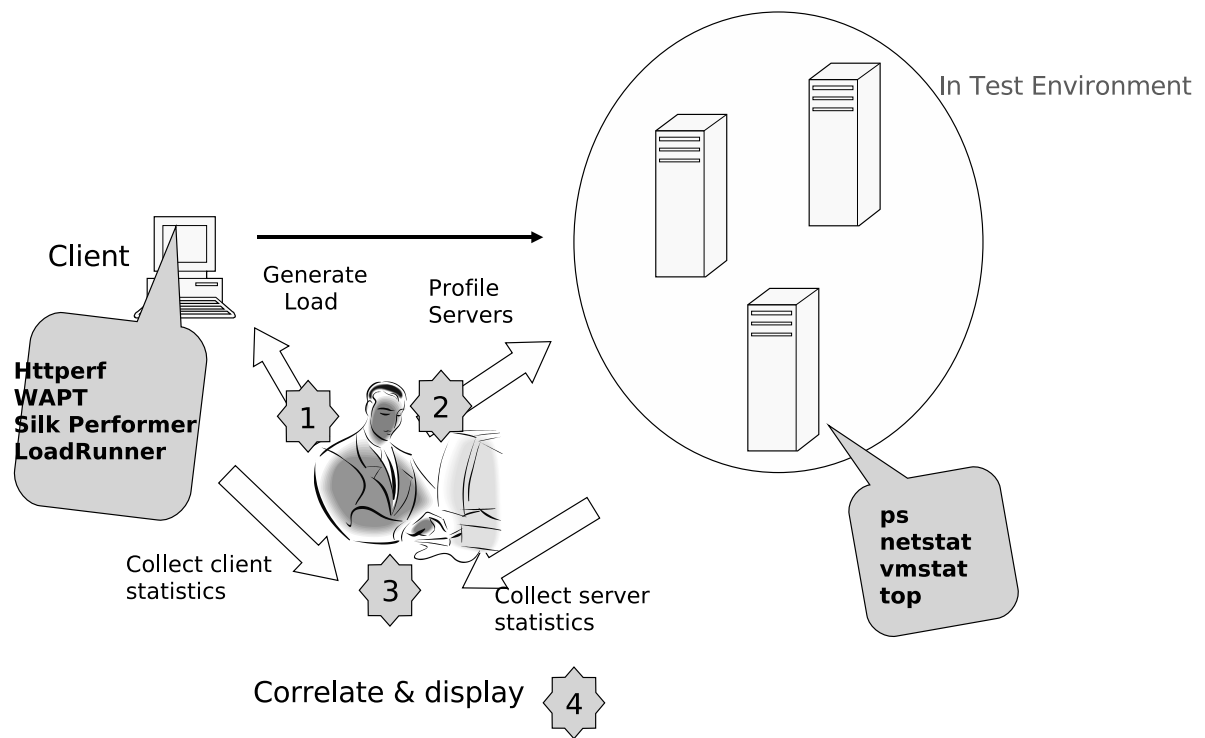


Figure 1.2: Correlating results of a load generator and profilers manually

the timeout interval. An effective user behavior description would take into consideration the user-timeouts.

The *server deployment configuration* describes the server sites, i.e. the machines on which the server is deployed and which processes running on those machines form the server. It typically consists of the server-machine names (or IPs) and the server-process names. Figure 1.4 shows a deployment configuration of a hypothetical server. It tells us that the Authentication Server is made up of the ‘authdaemon’ process deployed on the machine with IP 10.12.41.1, that the database server is made up of the process ‘mysqld’ running on the machine aloha.cse.iitb.ac.in and that the machine with IP 10.12.41.3 hosts the application server made up of the process ‘aspnet_wp’. A fine-grained performance measurement tool can also take as input, the names of the *methods* that are to be profiled in the server process.

In return, the tool should be able to determine the maximum throughput that can be achieved and the load level at which it is achieved. It should be able to profile the server to obtain measures like CPU service demand, CPU utilization, memory consumption, etc. which aid in solving performance models. It should be able to determine how many machines out of the ones available for load generation are sufficient to generate the required synthetic load, and should employ additional machines to generate load only when required. Care should be taken during load generation to ensure that the load generator itself is not a bottleneck (Simulation of too many users on the same machine can cause the load generating machine itself to become a bottleneck). The tool should be capable of determining server *warm-up* and should initiate measurements only after that. It should not be needed to specify to the tool, the duration of load generation, the range of load levels within which load has to be generated and the steps

```

<Transaction>
  <Transaction_Name>
    Login_Transaction
  </Transaction_Name>
  <Transaction_url>
    http://www.eshopping.com?username=user&password=pass
  </Transaction_url>
</Transaction>

```

```

<Transaction>
  <Transaction_Name>
    Search_Query
  </Transaction_Name>
  <Transaction_url>
    http://www.eshopping.com?searchitem=telephone
  </Transaction_url>
</Transaction>

```

```

<Transaction>
  <Transaction_Name>
    Purchase_Item
  </Transaction_Name>
  <Transaction_url>
    http://www.eshopping.com?purchaseId=20021
  </Transaction_url>
</Transaction>

```

User behavior description:

In steady state, 60% of the transactions executed are *Login_Transaction*, 30% are *Search_Query* and the rest are *Purchase_Item* transactions.

Figure 1.3: Sample workload description

```

<server_component>
  <name>Authentication Server</name>
  <node>10.12.41.1</node>
  <process>authdaemon</process>
</server_component>

<server_component>
  <name>Database Server</name>
  <node>aloha.cse.iitb.ac.in</node>
  <process>mysqld</process>
</server_component>

<server_component>
  <name>Application Server</name>
  <node>10.12.41.3</node>
  <process>aspnet_wp</process>
</server_component>

```

Figure 1.4: Sample deployment configuration

For a server consisting of an authentication server, database server and an application server deployed on nodes 10.12.41.1, aloha.cse.iitb.ac.in and 10.12.41.3 respectively, the only deployment configuration required by a capacity analysis and profiling tool should be as shown in figure. The server-machine name/IP and the server-process name on that machine must be sufficient for the tool to profile that server-component.

with which load levels have to be increased. The capacity analysis and profiling tool should be intelligent enough to determine these, with only the inputs discussed in this section.

1.3.1 Existing tools for performance measurements

Httpperf, Flood, Load Runner, Silk Performer, JMeter, Torture, etc. are some examples of performance measurement tools.

- **Httpperf:** Httpperf[2] is a load generator which obtains client-side measurements. It does not obtain any server-side measures useful for performance modeling of the system under test.
- **Flood:** Flood[1] is a load generator which obtains client-side measurements. It does not profile the server for measures useful for performance modeling of the system.
- **JMeter:** Apache JMeter[3] is a pure Java desktop application designed to load test functional behavior and measure performance. It however cannot obtain any server-side measures.
- **Torture:** Torture is a Perlscript used to stress test an application and to measure performance. It can obtain client-side measurements but can't obtain any server-side measures like most other load generators.

- **Load Runner:** Load Runner[4] is a leading load generator which can perform client-side measurements. Also, it obtains aggregate server-side measures but not fine-grained server-side measures on a per-process per-transaction basis, egs. service demand of a transaction at the database server. It has a rich GUI and supports a large suite of protocols.

As seen above, while some are only load generators, others try to collect some resource statistics at the servers too. Httpperf, Flood, Torture, etc. are examples of load generators. Load generators in addition to their use for checking the robustness of the system by subjecting it to varying load conditions are useful to obtain various client side metrics like response time, throughput, error information, etc. However they don't provide any insight into what is happening at the servers.

Tools like Load Runner and Silk Performer, are examples of leading proprietary performance testing tools which integrate a load generator and profilers. However, the profiler here is only of the form of a statistics-collector at the server-machines. They report aggregate resource usage statistics on the server node. They do not report resource usage per transaction per server-component (i.e. at a finer level), which is a crucial input for performance models. The tools require inputs like duration of load generation, range of load levels within which to generate load and steps of load levels within which to increase load, which are part of neither the workload description nor the user behavior characteristics nor the deployment configuration - an example of which is shown in figure 1.5 - and which should ideally be determined by the performance testing tools themselves. This is further elaborated in the next section.

1.3.2 Need for an automated capacity analysis and profiling tool

As discussed in section 1.3.1, some of the tools that do performance testing are only load generators, while some provide server side resource statistics, though at a very coarse level. Yet, most performace testing tools of the latter type have one or more of the following deficiencies:

- Tools require lot of inputs to be specified which assist them in carrying out the process of capacity analysis process. Ideally, the tool should carry out the capacity analysis process in a completely automated manner with only the inputs specified in section 1.3. Figure 1.5 shows some of the inputs needed to be specified to Load Runner (in one of it's various modes of operation). As can be seen, Load Runner requires the minimum and maximum number of virtual users (i.e. the range in which capacity analysis is to be performed) to be specified by the tool user. It also needs as input the duration to run the load generation for.
- Tools are available to measure response times per transaction, but not for measuring service demands per transaction. Service demand per transaction per server-component is an important input to performance models which the existing tools do not determine. Figure 1.6 shows how Load Runner reports various measures only on a per server-machine basis and not on a per server-component basis. As can be seen, Load Runner does not report measures on the server side (egs. service demand) on a per transaction per server-component basis. Instead, it reports only aggregate measures obtained on the server-machine as a whole. Thus, the results of load testing are not useful further for performance modeling of the system.
- To obtain service demand per transaction per server-component using existing tools, the co-ordination between the load generator and profilers need to be done manually. The

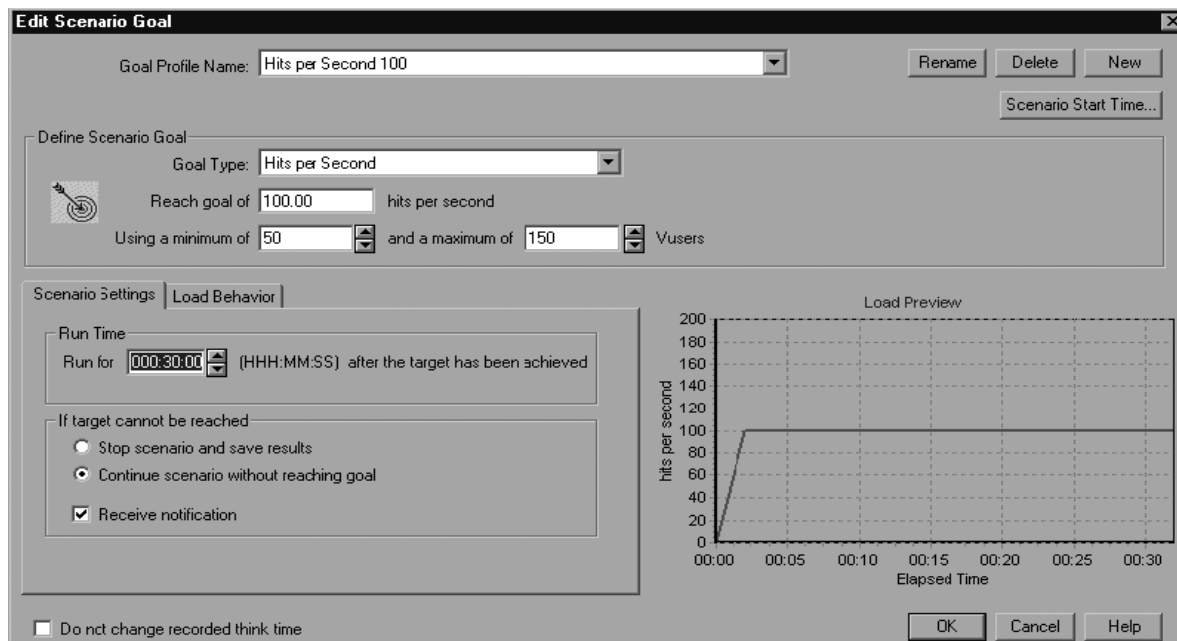


Figure 1.5: Snapshot of Load Runner's input[4]

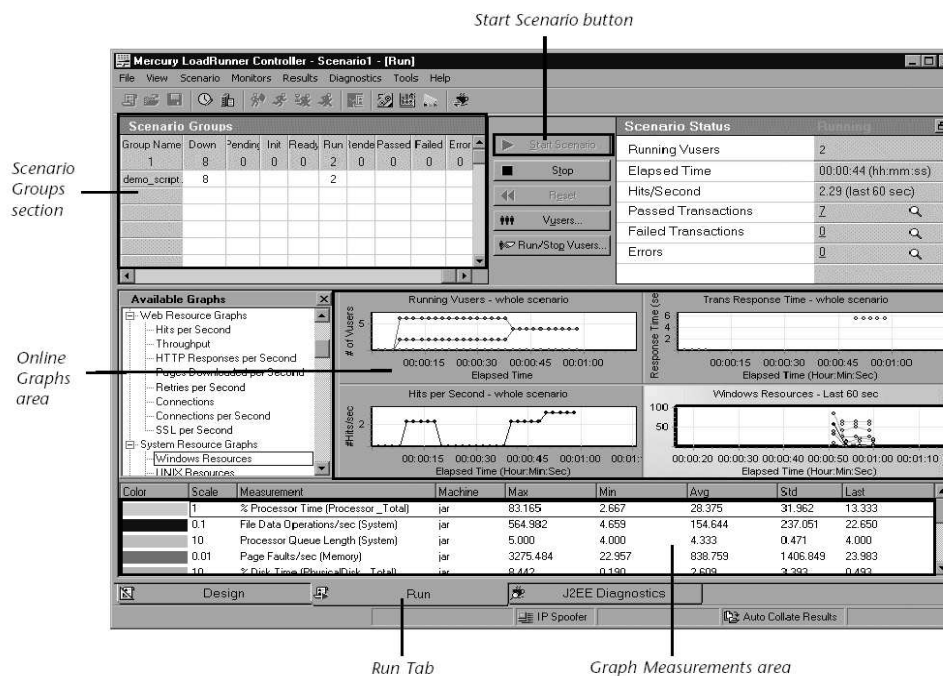


Figure 1.6: Snapshot of Load Runner's output[4]

obtained results would then be needed to be processed further to obtain per transaction per server-component service demand. This manual process is error-prone and tedious.

- The rate at which load is to be generated on the servers and its duration need to be specified as input to the existing tools as shown in figure 1.5. An intelligent tool should determine these inputs by itself (because these are inputs which describe the load generation process and are not descriptions of the system).
- Some tools allow the performance analyst to define virtual users and their actions (use cases). Also, the mix of such virtual users to generate can be specified in the form of percentage of numbers of virtual users of each type. However, a customer behavior model graph (CBMG) [9] is a more accurate representation of the realistic workload.
- Tools give the performance analyst an insight into the utilization of various resources like disk, CPU, memory, etc. They also tell which (software) resources are bottlenecks, but by using monitors available for specific backend systems. Such monitors are available only for widely used backend systems. Thus the tools cannot report bottlenecks and resource consumption details when the server employs new backend systems or when monitors are not available for the employed backend systems.
- A good tool for performance measurements should automatically detect the initial fluctuations in measurements, also called as the warm-up phase. Measurements taken during this phase can be erroneous. Hence, to avoid this, the profiling process and measurements at the client-side need to be started only after sufficient warm-up of the server. This makes the measurements more accurate. Section 2.4.4 describes the techniques used by AutoPerf to determine server warm-up.
- It should be able to detect the capacity of the system. Capacity can be defined as the load level (or the number of concurrent users being served) at which the server throughput is maximum.
- A good performance measurement tool should provide mechanisms to avoid caching at the servers, without which if load is generated and measurements performed can lead to measurements quite different from those under realistic load conditions. For example, if repetitions of the same transaction during load generation is causing the same database query to be executed, then the result of the query can be cached at the database server, and the database server would then remain unutilized. In such a case the measurement results will not be accurate. This can be avoided by generating different URLs in each web-request.
- Profiling certain transactions might require the user to first log into the system. As a result, user sessions need to be maintained by performance measurement tools. Some of the existing load generators do not support session management.

1.4 Contributions of this thesis

The work done as part of this thesis carries forward the tool named *AutoProf* earlier, developed at IIT Bombay as is described in [12]. Following is a brief overview of the contributions of this thesis.

- Our main goal behind developing the tool AutoPerf is to demonstrate a new paradigm for performance testing and profiling. We wish to present a prototype of the tool which demonstrates this shift in paradigm. For the purpose of pace of development and to make it easy to handle the increase in complexity of load generator after the incorporation of the proposed improvements, it was decided to migrate the master part of AutoPerf - Load Generator, Controller and Master part of the Profiler - to Java. This involved re-implementing the Master part of AutoProf in Java, which was earlier fully implemented in C++.

The decision to migrate the Master part of AutoPerf from C++ to Java was taken after conducting various experiments on different servers and confirming that the Java implementation does not affect the crucial metrics that we measure on the Master side of AutoPerf - response time, throughput, etc. This involved conducting experiments consisting of load generation at different load levels on different servers using the C++ as well as Java implementation of AutoPerf and performing measurements at the master for various client-side measures. The response times observed using both the implementations were compared and the difference in the observed values was found to be negligible. Further, this negligible difference could be attributed either to changes in some uncontrolled factors in the runtime environment i.e. measurement errors along with the differences possibly introduced due to the implementation in Java. Graphs 1.7 and 1.8 show the response times observed under varying load conditions for different servers. As throughput can also be given as,

$$Throughput = \frac{Load\ Level}{Average\ Response\ Time + Think\ Time}$$

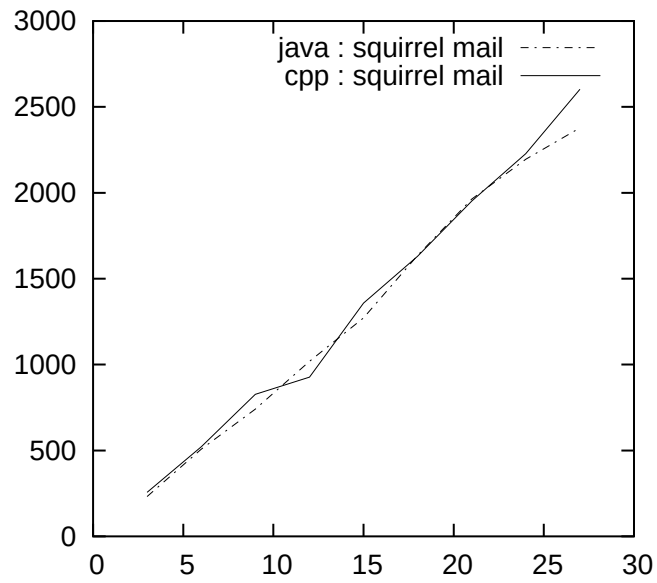
and think time is constant for a given experiment, throughput measurement errors can only consist of response time measurement errors. Hence, graphs of comparison of throughputs observed using the C++ and Java implementations of AutoPerf are not presented separately.

Figure 1.7 shows the response times observed for the 'Login' transaction in squirrel mail using the C++ implementation of AutoProf and the same using the Java implementation.

Figure 1.8 shows the response times observed to open a static web page on apache server using the existing C++ implementation of AutoProf and the Java implementation. The response in this case consisted of a very small string.

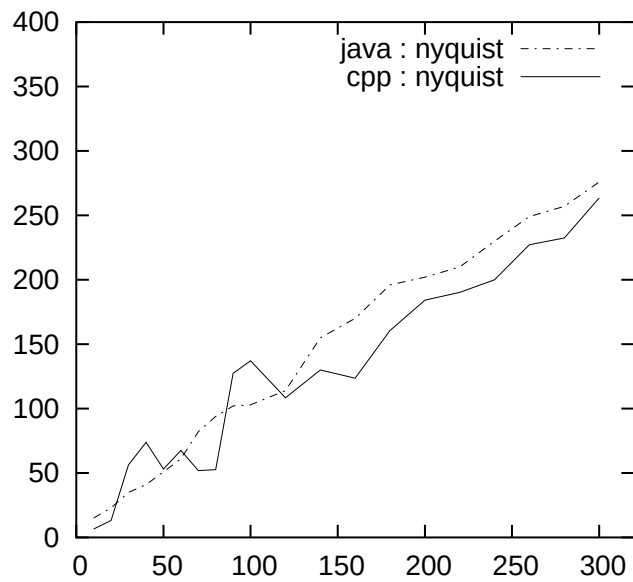
The results in graphs 1.7 and 1.8 confirm that implementation in Java does not affect the measurements being made.

- The focus of this thesis was on minimising inputs required by the tool, so as to facilitate user convenience. This means that the tool should have enough intelligence built into it, so as to determine the values of these eliminated inputs by itself. The improvements made to AutoPerf in order to achieve this goal are elaborated in section 2.4.



X axis: Load Level (Number of Users)
Y axis: Response time in milliseconds

Figure 1.7: Login transaction in *SquirrelMail*



X axis: Load Level (Number of Users)
Y axis: Response time in milliseconds

Figure 1.8: A very small and static response from Apache web server

Chapter 2

AutoPerf: An Automated Load Generator and Performance Measurement Tool for Multi-tier Software Systems

Section 1.3.2 throws light on the limitations of existing performance testing tools and the need of an automated tool for capacity analysis and profiling. *AutoPerf*, is an intelligent performance testing tool which overcomes the limitations mentioned in section 1.3.2. AutoPerf demonstrates a new paradigm for performance testing, wherein the tool is intelligent enough to perform capacity analysis of the system under test and to profile the resource consumption of the server when given as input only the server deployment configuration, workload description and user behavior description.

2.1 Input Description

To perform capacity analysis and fine-grained server profiling of the system under test, AutoPerf requires as input the following:

- (i.) Server deployment configuration
- (ii.) Workload description
- (iii.) User behavior description.

As described in section 1.3, the *deployment configuration* consists of the IP addresses or hostnames of the server-machines and the names of the server-components, i.e. the server processes running on them. It also reads as input a summarisation of the mix of transactions in the form of a CBMG (discussed in detail in section 2.4.3) and the definitions (URLs,etc) of each transaction which form the *workload description*. AutoPerf requires the *user behavior description* in the form of think time i.e. average time the users of the system remain idle before requesting the next URL and user-timeout as described in section 1.3.

2.2 Output Description

AutoPerf can operate in 3 modes:

- (i.) Profiling mode and at a single load level

- (ii.) Profiling mode and at multiple load levels (till system saturation)
- (iii.) Capacity analysis mode in which a mix of transactions are generated probabilistically, thereby emulating realistic scenario.

2.2.1 Profiling mode, at a single load level

This is the simplest of all the modes of operation. In this mode, load is generated at a specified load level. Servers are profiled and client-side measurements are performed. In this mode, AutoPerf reports various client-side measures like the throughput, the average response time of a transaction and server profiles like the CPU service demand of each of the server-processes per transaction, CPU utilization of each of the server-processes, memory usage of each of the server-processes, disk bytes read by each of the server-machines, disk bytes written by each of the server-machines, network bytes read by each of the server-machines and network bytes written by each of the server-machines, all at the specified load level. Java profilers (*JavaProf* [6]) are deployed at the servers. Java components of the server are profiled at method level and average method level elapsed times at different physical as well as virtual resources are reported. Additionally, heap and non heap memory usage of the application being profiled and the message sequence chart and garbage collection information is reported by JavaProf.

2.2.2 Profiling mode, at multiple load levels

In this mode, the server is profiled when load is generated for a single transaction at various load levels. In this mode, AutoPerf reports various client-side measures like throughput, the average response time of a server transaction, etc. and server-side resource profiles like the CPU service demand of each of the server-process per transaction, CPU utilization of each of the server-processes, memory usage of each of the server-processes, disk bytes read by each of the server-machines, disk bytes written by each of the server-machines, network bytes read by each of the server-machines and network bytes written by each of the server-machines, all at each load level separately. Java profilers (*JavaProf* [6]) are deployed at the servers. Java components of the server are profiled at method level and average method level elapsed times at different physical as well as virtual resources are reported. Additionally, heap and non heap memory usage of the application being profiled and the message sequence chart and garbage collection information is reported by JavaProf.

Figure 2.1 shows some of the measures that are reported by AutoPerf for a hypothetical system which consists of a web server, authentication server and an IMAP server. It obtains client-side measures like response time, throughput etc. at the load generator all at different load levels and at the same time obtains resource profiles like CPU service demand per transaction, CPU utilization at the server, memory usage, etc (not all are shown in figure 2.1) all at each load level and at each of the various server components, i.e. the web server, the authentication server and the IMAP server.

The range of load levels within which load generation and profiling needs to be profiled is determined by AutoPerf itself. It automatically determined the load levels at which load generation and profiling needs to be performed. The tool is itself capable determining the duration of load level at each load level.

This mode of operation is useful to see the change in service demand of the transaction, CPU utilization, etc with changes in load level. This can help point out some unanticipated behavior

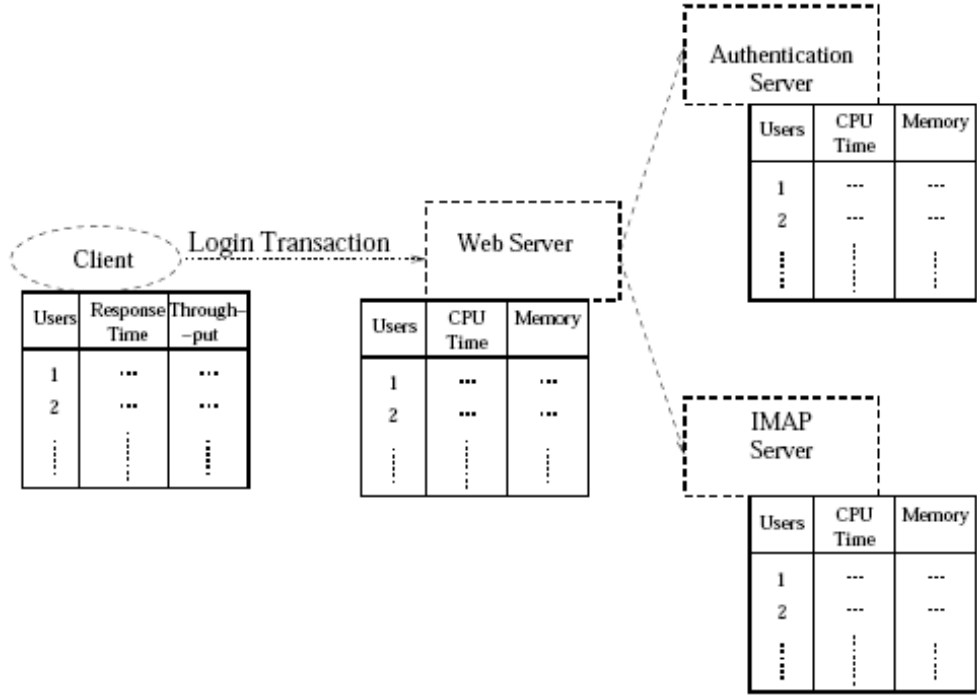


Figure 2.1: Client-side and server-side measurements performed by AutoPerf in profiling mode for multiple transactions
(Not all measurements performed are shown)

of the server, egs. an exponential increase in service demand of the transaction at the database server, the cause of which could be a poorly written database query, etc.

2.2.3 Capacity Analysis mode

This mode of operation is extremely useful to generate load which resembles the realistic workload by generating a mix of transactions according to a customer behavior model graph as described in section 2.4.3. In this mode, in addition to the output described in section 2.2.2, AutoPerf reports maximum achievable throughput and saturation load level (i.e. load level at which maximum throughput is achieved). However, the measurements reported here are those resulting due to a mix of transactions, egs. service demand per transaction is the ratio of cumulative service demand as a result of the mix of transactions executed to the total number of transactions executed.

2.3 Architecture of AutoPerf

AutoPerf mainly consists of three modules:

2.3.1 Controller

The controller co-ordinates the load generator and the profilers. The main job of the controller is to select the load level, begin load generation at that level and collect measurements from load generator and profilers when load generation is over. Using the obtained measurements,

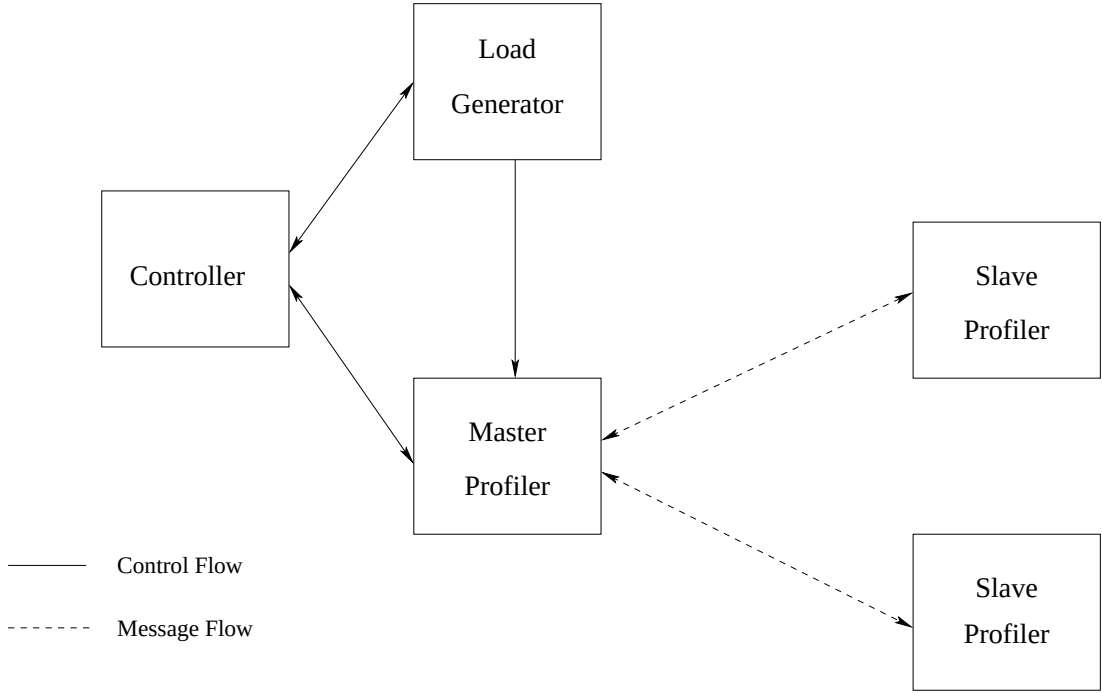


Figure 2.2: AutoPerf Architecture

the controller determines the next load level. The load levels are determined in such a way that the trend of the performance measures can be obtained over a range of load-levels withing which throughput saturates, by generating load at as few load levels as possible.

2.3.2 Load generator

It generates requests for the server at the load-level specified by the controller. It is responsible for detecting warmup of the server and then starting the profiler.

The primary responsibility of the load generator is to generated load (web-requests) on the web-server at the load level (number of concurrent users using the system) specified to it by the controller. The load generator automatically determines server warm-up and when warm-up is detected, it directs the profiler to start profiling.

2.3.3 Profiler

The profiler has a master slave architecture. The master profiler executes in the same process as the load generator and the controller, while the slave profilers execute as a daemon process at each of the software servers of the web-based system under test. When the load generator starts the profiler, the master part of the profiler initializes and starts all the slave profilers. A slave profiler is deployed at each of the servers. The slave profiler is responsible for collecting resource usage information at the server and returning it to the master profiler when queried for, which in turn forwards the server-side resource profiles to the Controller.

Master Profiler:

The main job of the master is to interact with the slaves at the server nodes. Master gives commands to the slave to start capturing the resource usages, stop profiling and sending the

profiled data to it. When the load generator directs the master to start profiling, the master sends the initialisation commands to each of the slaves to start capturing the resource usage. When the load generator finishes, it directs the master profiler to stop profiling, the master then sends a command to the slave profilers indicating them to stop profiling and to send the resource profiles to the master. The data collected by the master is then used by the controller to determine the highest load-level up to which load needs to be generated.

Slave Profiler:

The slave profiler could either be a profiler built for linux systems [12], or it could be a profiler for windows platforms[6] . The profilers are operating system specific and are deployed at the server sites accordingly. JavaProf[6], a Java profiler compatible with AutoPerf for profiling methods in Java applications has been developed [6]. Java profilers (*JavaProf* [6]) are deployed at the server. Java components of the server are profiled at method level and average method level elapsed times at different physical as well as virtual resources are reported. Also, heap and non heap memory usage of the application being profiled and the message sequence chart and garbage collection information is reported[6].

Figure 2.2 explains the architecture of AutoPerf diagrammatically.

2.4 Features of AutoPerf

Following is a brief summary of AutoPerf's features:

- Also other tools require the maximum load level (number of concurrent users) i.e. upper bound on saturation load level and steps with which to increase the load level on the server, to be specified by the user. Ideally, the user should be relieved of this, by incorporating enough intelligence in the tool, so that the tool is capable of determining itself the maximum load level that the system under test is able to sustain. Section 2.4.2 describes the technique used by AutoPerf for estimation of saturation load level.
- AutoPerf automatically determines the right value of execution count, i.e. number of times to repeat each transaction. Section 2.4.1 describes this process in detail.
- During capacity analysis, once the range of load levels withing which the saturation load level lies is known, AutoPerf determines the saturation load level and the maximum throughput of the system that is achievable by generating load at as few load levels as possible using the algorithm described in [12]. Algorithm 1 summarizes the algorithm. Current tools increase the load sequentially, in steps and within range of load levels as specified by the user, to determine the maximum throughput achievable.
- To truly perform capacity analysis, it is necessary to emulate realistic load scenarios. Realistic load conditions can be emulated by AutoPerf in the capacity analysis mode of operation in which it reads as input a Customer Behavior Model Graph (CBMG)[9] (discussed in section 2.4.3) and various transactions can be generated in proportion as summarized by the CBMG. Most other performance testing tools just take as input the proportion (percentages) of each type of user, where each user type is summarised by a scenario i.e. a sequence of transactions. A CBMG is a far better approximation of user behavior. Section 2.4.3 describes the capacity analysis mode of operation, also called as probabilistic session generation.
- AutoPerf detects server warm up and starts measurements only after that so that the measurements are representative of the actual values of response time, throughput, etc.

When load generation just begins this may not be true. For example, initial number of server threads may not be sufficient to handle the current load level, which might result in spawning of additional threads. To take care of such transient scenarios, server warmup detection is necessary.

Algorithm 1 Algorithm to perform capacity analysis within range of load levels $[a, b]$

Generate load at level a and measure the throughput T_a

Generate load at level b and measure the throughput T_b

CapacityAnalysis ($a, b, T_a, T_b, permissible_error$)

function *CapacityAnalysis*(a, b, T_a, T_b, e)

begin

if ($a \geq b$)

return

$c \leftarrow \frac{a+b}{2}$

 Generate load at level c and measure the throughput T_c

 Calculate $T'_c \leftarrow \frac{T_a+T_b}{2}$

if $\frac{|T'_c-T_c|}{T_c} \geq e$

then

CapacityAnalysis(a, c, T_a, T_c, e)

CapacityAnalysis(c, b, T_c, T_b, e)

end if

end function

In order to detect server-warmup, in profiling mode we maintain a moving average window of transaction response times to detect the steady state of the server, as described in [10]. AutoPerf uses a window of size w to record the w most recent transaction response times. Every time a new transaction occurs the response time value observed is added to the window, and the window is shifted. If the difference in the average of the values in the window before and after shifting is within a certain threshold limit, we say that the server has warmed-up and we begin measurements then onwards. Too small a value of w will result in too much fluctuations between successive averages of values in the window and too large a value of w , would result in no fluctuations at all. This method of detecting warm-up is based on Welch's method for discarding the initial transient and detecting the steady state [8].

- AutoPerf provides resource consumption details on a per transaction per server-component basis. Also for Java applications, it reports the message sequence chart, heap and non-heap memory usage, garbage collection information and average method level elapsed times at different physical as well as virtual resources.

AutoPerf initialises the profilers deployed at the various server components after warm-up of the system under test and then begins load generation. After the process of load generation, the duration of which is determined by the tool itself, the profilers are stopped and resource profiles are collected from them. The service demand of a transaction can be determined as the ratio of the cumulative service demand reported by the profiler to the number of transactions executed by the load generator. Service demand per transaction determined in this manner serves as a crucial input to various performance models.

AutoPerf provides resource consumption details on a per transaction per server-component

basis. Using current tools, obtaining this would require manually co-ordinating the entire process of load generation, initialising the profilers, stopping them and collecting resource profiles from them, and then drawing inferences to obtain the per transaction service demand at each of the server components.

- AutoPerf has a support for generating URLs dynamically.

In the load generation process, if the same URL is requested repeatedly by the load generator, the response of the request may be cached at the server. As a result successive requests would take up less resources of the CPU than it would have if it were requested for the first time. As an example consider the request `http://www.cse.iitb.ac.in?rollnumber=05305007` which shows the details of the student with roll number 05305007. Suppose that this request when sent to the web server, causes a call to the database server, where the details of the requested student are queried for in a database and the results returned to the web server. Now, if the same URL is requested repeatedly, caching can take place at multiple places. The application at the web server could be maintaining its own cache. Also, caching could take place even at the database server. As a result, the observed service demand in such a scenario would not reflect the true service demand of the transaction. In order to avoid this, multiple URLs should be requested, and which can be requested with the help of variables which can assume a range or a list of values as shown below:

```
<sequence name="rollno">05305007,05305402,05000001</sequence>
<url>http://www.cse.iitb.ac.in/student_details?roll=\${rollno}\</url>
```

In the above example the sequence variable is *rollno* and it can take values 05305007, 05305402 or 05000001 randomly. As a result successive URLs would be different, thus reducing the effects of caching.

- It has a provision to specify average think time which is representative of the average time the users of the system remain idle before requesting the next URL. This is necessary so as to emulate load with close resemblance to actual user behavior.
- AutoPerf supports user-timeouts. The time after which an user aborts the site can be specified to AutoPerf as part of the user behavior description.
- Provides support for cookie management, HTTP redirection and various HTTP methods like POST and GET.

AutoPerf supports the HTTP protocol, which includes features like URL redirection and various request methods such as GET and POST. Not all requested pages are available at the request location. Often the server redirects a URL to some other page on the same web server or to some different server. The status code of HTTP response indicates whether the requested page needs to be fetched from some other location and the actual location of the page is specified in the response packet. AutoPerf automatically checks if the response needs to be fetched from some redirected location, and if so fetches the response accordingly.

When a user enters information in a form in an HTML page and submits it, the information can be sent from the browser to the server in 2 ways - either by appending it to the URL, which happens in case of a GET request, or it can be sent within the body of the HTTP request, which happens in case of a POST method. AutoPerf supports both kinds of forms in HTML.

- AutoPerf defines a generic interface between the Master and Slave Profiler, which is independent of the platform on which the server is deployed, so that profiling of resources on servers irrespective of the platform on which they are deployed is possible. AutoPerf currently has profilers developed for Linux and Windows platforms and also a method level Java profiler (*JavaProf* [6]) has been developed.

The following sections elaborate some of the techniques used by AutoPerf and the user behavior description supported by it.

2.4.1 Automation of execution count determination

AutoPerf consists of 2 main components:

- Master i.e. load generator and controller
- Slave i.e. profiler

The load generator is independent of the platform on which the server-components are deployed. However, the profilers are platform specific. AutoPerf has profilers for Windows and Linux platforms. The Linux profiler provides cumulative service demand (i.e. sum of service demands of all the transactions executed initialising the profilers) information with a granularity (least count) of 1 second (i.e. in steps of 1 second). So an actual increase in cumulative service demand by a value which is less than 1 second may not necessarily be reflected in that reported by the profiler. Hence, to bring the resulting imprecision in the cumulative service demand and hence in the service demand of the transactions ($service\ demand = \frac{cumulative\ service\ demand}{no.\ of\ transactions\ served}$) within acceptable limits, it is necessary to repeat the transaction being profiled for n_1 number of times, one after the other, where n_1 is sufficiently large. AutoPerf automatically determines this value of n_1 .

Also, even though the measurements are initiated after sufficient server warm-up, the measured throughput is more accurate if the process of load generation is continued for a longer duration. The transaction being profiled should be repeated for say n_2 iterations, such that n_2 is sufficiently large, to result in an accurate measurement of throughput.

Thus, the repetition count n , which we also refer to as *execution count*, should be such that $n \geq n_1$ and $n \geq n_2$. Larger values of n would mean lower inaccuracy in the reported service demand and throughput. AutoPerf itself determines n and stops load generation after that and performs measurements thereafter.

Figure 2.3 shows a graph of cumulative service demand (of a resource and for a particular transaction) as would be obtained by the master (load generator and controller) from the slave (profiler) against execution count, when slave is queried for resource usage information after every execution of the transaction being profiled. The values shown in the figure are obtained by simulating the master and the slave components of AutoPerf and a transaction with service demand 300 milli seconds. As can be seen, cumulative service demand values increase in steps of 1 second.

Figure 2.4 shows a graph of true value of service demand and that which would be observed for a transaction against execution count by simulating the master, slave and a transaction with service demand of 300 milli seconds. It can be easily seen from figure 2.4 that when execution count is low, the observed service demand oscillates heavily after every transaction execution,

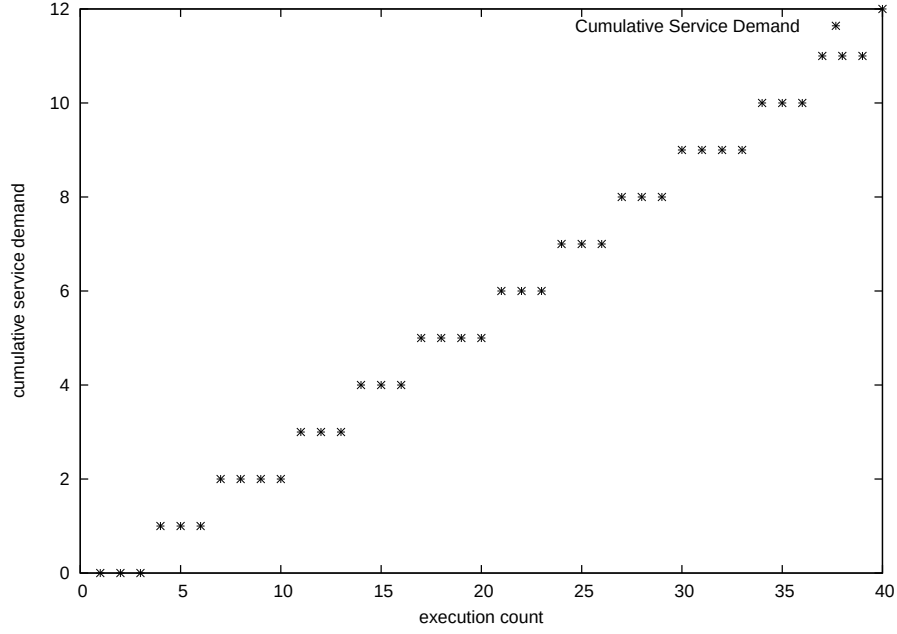


Figure 2.3: Graph of cumulative service demand of a resource as would be obtained by the master (load generator and controller) from the slave (profiler) against execution count i.e. number of times a transaction is repeated, by simulating a transaction with service demand 300 milli seconds.

but as execution count increases, the oscillations dampen, and approach the true service demand.

Thus, it is necessary that execution count is sufficiently large, so that the observed value of service demand is reasonably close to the true value. AutoPerf determines n_1 (n_1 is defined in the preceding paragraphs) as the execution count beyond which the amplitude of oscillations in the service demand observed is within permissible limits. AutoPerf maintains a moving window of most recent w observed service demands. It obtains service demands from the profilers after every few requests of the transaction being profiled as is described further. After every query to the profiler for the resource usage information, the window is updated (shifted) to now remove the oldest value from it and insert the most recent value of service demand into it, and the new average of the values in the window is calculated. This process of generating further requests continues, till a point in time when the deviation of each of the w values in the window from the average of all those w values is within the permissible limits, thereby automating the process of determination of the value of execution count, with respect to the measurement of service demand. The execution count at this time is considered to be a fairly correct value of n_1 .

To ensure that the measured throughput is also accurate, AutoPerf maintains a window of most recently measured throughput values. It measures throughput after every few requests of the transaction being profiled as is described further. The window for throughput value is maintained similar to that of service demand values and n_2 is calculated accordingly. The execution count n is determined as $\max(n_1, n_2)$.

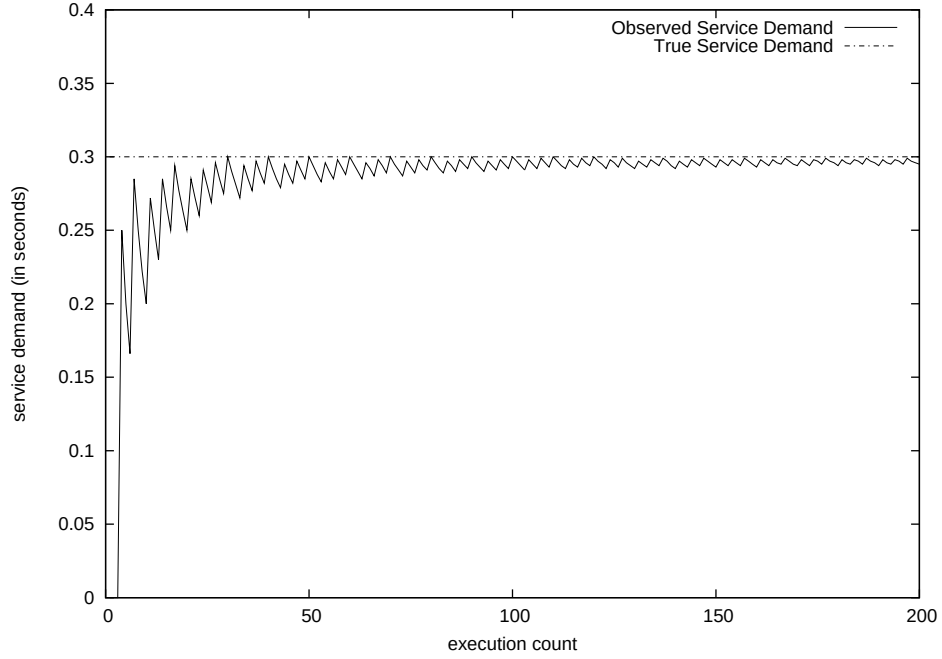


Figure 2.4: Graph of true service demand and the observed service demand of a resource as observed by the master (load generator and controller) against execution count i.e. number of times a transaction is repeated, by simulating a transaction with service demand for the resource 300 milli seconds.

Frequency of throughput and service demand measurement

To maintain the window of w most recent throughput and service demand values, we need to perform measurements periodically. The most primitive approach to do this is to perform measurements after a duration, say t , where t can be defined heuristically. However, this method of performing measurements has a pitfall. If t is defined such that sufficient number of transactions take place within the duration t as lower load levels, then as load level increases much higher number of transactions will be executed within the same duration t , thus unnecessarily causing the measurements to be performed after duration t , when it could be performed even after a much lower inter-measurement interval.

A better approach is to perform measurements after every k transactions, where k can be defined heuristically. This overcomes the drawbacks of the earlier approach. However, as load level increases, k transactions can be executed in very short periods as compared to that at lower load levels. This could cause the window of throughput values to diverge unnecessarily. However, if measurements are performed after every $k * l$ transactions, where l is the load level at which load is being generated, then the above limitations are overcome. AutoPerf employs this approach to determine inter-measurement interval.

2.4.2 Automated capacity analysis

As part of capacity analysis, AutoPerf subjects the system under test to various levels of load i.e. number of concurrent users accessing the system. Thus it is possible to determine the saturation load level i.e. load level beyond which throughput saturates or degrades. A naive approach of

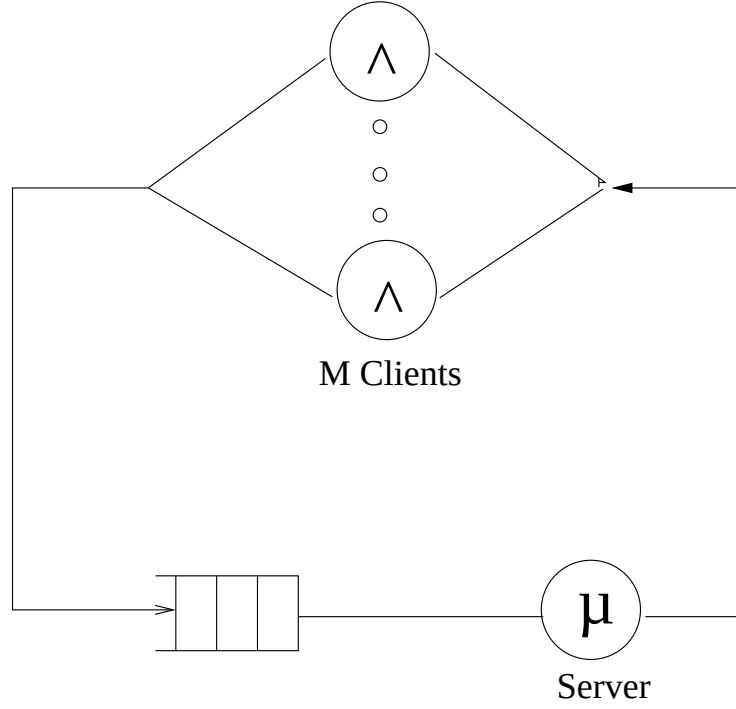


Figure 2.5: A client-server system.

capacity analysis would be to generate load at every load level, incrementally, starting from 1, till the point of saturation (maximum throughput) is reached. This approach of determining saturation load level, although simple, could result in the experiment taking too long time. In order to accelerate this process of determination of saturation load level, it is essential that load be generated at fewer load levels.[12] describes such a mechanism. This mechanism requires as input, an upper bound, M , on the saturation load level. Requiring this to be specified as input has the following disadvantages:

- The saturation load level M^* could be well beyond M ($M^* \gg M$), thus resulting in AutoPerf reporting a lower value of saturation load level.
- The saturation load level M^* could be much lower than M ($M^* \ll M$), as a result of which AutoPerf would end up, redundantly repeating the experiment (i.e. load generation) at much higher load levels.

AutoPerf earlier required M to be specified as input. The following sections describes how AutoPerf now automatically determines M .

Estimation of saturation load level

Figure 2.5 shows a client-server system with M clients, average think time $\frac{1}{\lambda}$ and average service rate of server μ . The saturation load level $M_{predicted}^*$ of such a system is given by $1 + \frac{\mu}{\lambda}$ [15]. This means, the maximum number of clients that can be supported by the server without saturation in throughput is $M_{predicted}^*$.

The observed saturation load level $M_{observed}^*$ may be lower than $M_{predicted}^*$. This could be due to variety of reasons like server's restriction on number of concurrent users being served,

	<i>Entry</i>	<i>Login</i>	<i>Search</i>	<i>Inquire</i>	<i>BookTickets</i>	<i>Exit</i>
<i>Entry</i>	1.0	0.0	0.0	0.0	0.0	0.0
<i>Login</i>	0.0	0.0	0.4	0.2	0.4	0.0
<i>Search</i>	0.0	0.1	0.0	0.6	Inquire	
<i>Inquire</i>	0.0	0.3	0.0	0.0	0.4	0.3
<i>BookTickets</i>	0.0	0.1	0.0	0.5	0.0	0.4
<i>Exit</i>	0.0	0.0	0.0	0.0	0.0	0.0

Table 2.1: Matrix representation of a customer behavior model graph

system's limit on number of threads that can be created, etc. However, $M_{predicted}^*$ solves our purpose of obtaining an upper bound on $M_{observed}^*$. The algorithm mentioned in [12] to determine saturation load level has been modified to suit our needs of automated $M_{predicted}^*$ prediction.

Automation of saturation load level determination

Assuming that service demand of a transaction remains constant irrespective of the load level, $M_{predicted}^*$ can always be given as $M_{predicted}^* = 1 + \frac{\mu}{\lambda}$. However, the service demand of a transaction can change with load level. As a result, we determine $M_{predicted}^*$ at a load level m as $M_{predicted}^*(m) = 1 + \frac{\mu(m)}{\lambda}$.

Initially, AutoPerf determines $M = M_{predicted}^*(1)$, and performs capacity analysis up to a load level of M . Again, it determines new M as $M_{predicted}^*(M)$ and performs capacity analysis together up to M . This process is continued recursively till, capacity analysis at load level $M_{predicted}^*(M)$ has already been performed at the time of calculating new M .

In practice, a server could consist of multiple nodes. Consider such a server of figure 2.6, with service rates $\mu_1, \mu_2, \dots, \mu_n$. $M_{predicted}^*(m)$ of such a system is given as

$$M_{predicted}^*(m) = 1 + \mu_{S_b}(m) * \left(\frac{1}{\lambda} + \sum_{i \neq S_b} \frac{1}{\mu_i(m)} \right)$$

where,

$S_b \leftarrow$ Bottleneck server-component

$\mu_i(m) \leftarrow$ Service rate of server-component i at load level m

Algorithm 2 is used to generate load and to determine saturation load level. In a given range, load is always generated with minimum number of load levels (using algorithm *LoadSelector* given in [12]) and in such a way that the obtained throughput curve in that interval closely resembles the true throughput curve, i.e. the one obtained when load is generated at up to levels upto saturation. α and β control the manner in which the load is generated. Load is generated using minimal number of sample points using algorithm *LoadSelector()* given in [12] up to $\alpha * M_{final}^*$ and then at every load level further up to $\beta * M_{final}^*$. Reason to generate load at every level is to observe saturation in throughput more clearly.

2.4.3 Probabilistic Session Generation

Often, customers of a site interact with the site through a series of requests, each of which is inter related. A customer visit which consists of such inter-related requests is called a *session* [9].

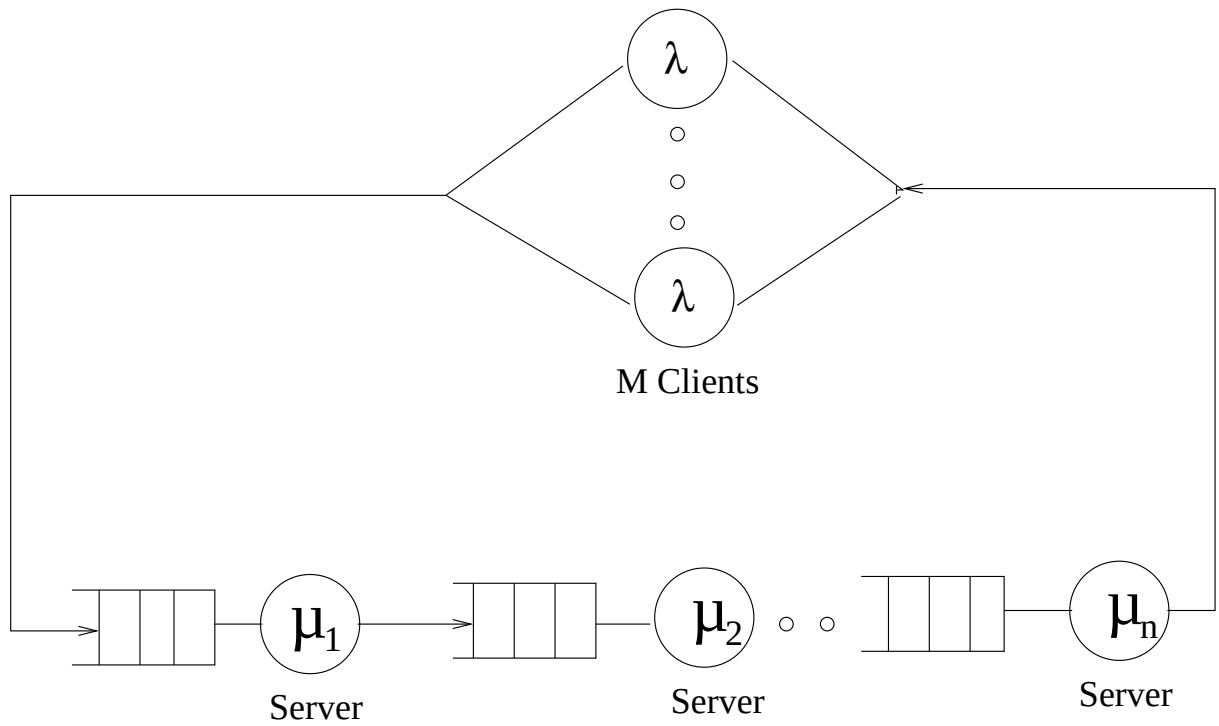


Figure 2.6: A client-server system with multiple server nodes.

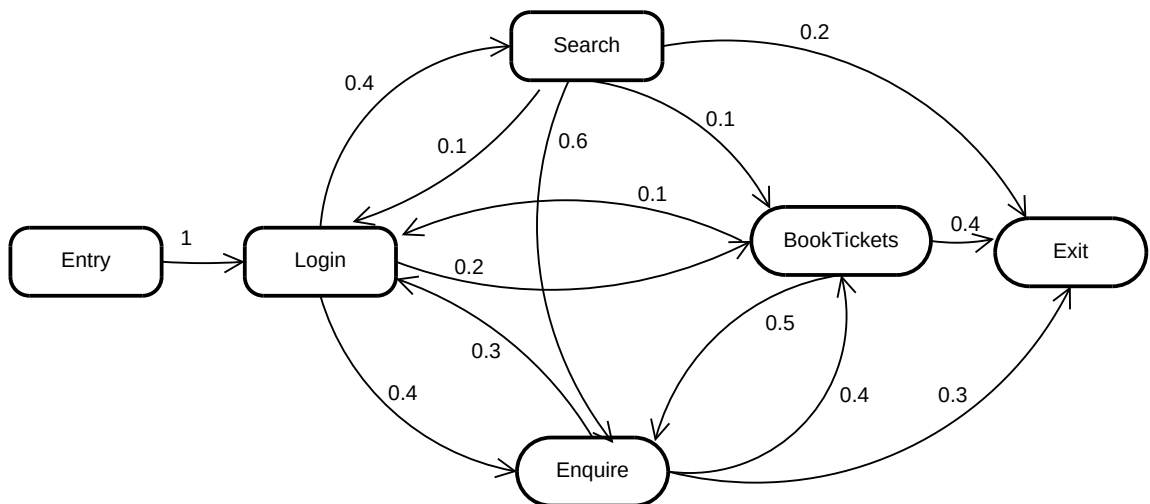


Figure 2.7: Customer behavior model graph

Algorithm 2 Algorithm for determining saturation load level and generating load.

```

procedure DetermineSaturationLoadLevel ()
begin
     $DetMaxUsers(1, 1)$ 
end procedure

```

```

procedure DetMaxUsers ( $RangeStart, RangeEnd$ )
begin
    Generate Load in the range [ $RangeStart, RangeEnd$ ]
    using  $LoadSelector(RangeStart, RangeEnd)$ 
    Calculate  $M_{predicted}^*(RangeEnd)$ 
     $NextRangeStart \leftarrow RangeEnd + 1$ 
     $NextRangeEnd \leftarrow \alpha * M_{predicted}^*(RangeEnd)$ 
    if (  $NextRangeStart \leq NextRangeEnd$  )
    then
         $DetMaxUsers(NextRangeStart, NextRangeEnd)$ 
    else
         $NextRangeEnd \leftarrow \beta * M_{predicted}^*(RangeEnd)$ 
        Generate Load at every level in the range [ $NextRangeStart, NextRangeEnd$ ]
    end if
end procedure

```

```

 $\alpha \leftarrow$  Chosen heuristically in the range  $(0, 1]$ 
 $\beta \leftarrow$  Chosen heuristically in the range  $[1, \infty)$ 
 $LoadSelector() \leftarrow$  Algorithm to determine load levels at which load is to be generated, ([12]).

```

Within a session, customers can issue requests of different types, such as login, search for a train between two stations, inquire availability of tickets, book tickets, etc. Navigational patterns of different users may be different. A *Customer Behavior Model Graph (CBMG)* is an effective means to model customer behavior.

Customer Behavior Model Graph (CBMG)

A customer behavior model graph is a directed graph of n states, where state 1 is always the *Entry* state and state n is always the *Exit* state. All the other states correspond to a transaction in the system. In the example discussed above, there will be a state for each of the transactions - *Login*, *Search*, *Inquire* and *BookTickets*. Figure 2.7 shows the corresponding CBMG.

A CBMG summarizes the navigational pattern of a group of customers, as viewed from the server side. A CBMG description formally consists of a set of states, a set of transitions between states, and an $n * n$ matrix, $P = [p_{i,j}]$, of transition probabilities between the n states. Table 2.1 shows the matrix representation of CBMG in figure 2.7.

A CBMG consists of 2 parts - *static* and *dynamic*. The static part of a CBMG summarizes the set of functions provided to the customer and the transitions between those states (functions). [9] describes a method to obtain dynamic part (transition probabilities) of CBMGs by processing the HTTP (server) logs.

AutoPerf and CBMGs

Workload description can be described to AutoPerf summarized in the form of a CBMG. AutoPerf then generates load on the server in such a way that the mix of transactions generated resembles the workload specified using the CBMG. AutoPerf itself maintains user sessions. The controller determines the next state (transaction) using the CBMG after every request is executed i.e. at the time of transition from any state. This method of generating load probabilistically and obtaining various metrics is a lot easier as compared to solving the CBMG and then modeling the system analytically.

2.4.4 Warm-up

AutoPerf employs similar but 2 different techniques to determine server warm-up, for it's different modes of operation.

Warm-up detection in profiling mode of operation

We use the technique described in [10] to detect server warm-up. AutoPerf maintains a moving window of w most-recent transaction response time values observed. Every time a new transaction is executed, it's response time is added to the window and the oldest value from the window is removed. We thus say that the window is shifted. When the difference of 2 successive windows of size w becomes less than a particular threshold, the warm-up is assumed to be detected.

Warm-up detection in capacity analysis mode of operation

In this mode of operation, as successive transactions might not be the same (they are chosen probabilistically according to the CBMG) maintaining a window of transaction response times will not work. Instead, we can maintain a window of throughput values observed at different

times during the experiment. The interval between successive measurement of throughput values is determined as described in section 2.4.1.

2.4.5 Dynamic URL generation

AutoPerf supports use of variables, so that URLs can be generated dynamically using variables. For example,

```
<transaction>
  <name>DeleteMessage</name>
  <url method="POST">
    http://www.cse.iitb.ac.in/shrirang/file$i$j$k.html
  </url>
  <sequencelist>
    <sequence name="i">2,3</sequence>
    <sequence name="j">0,1,2,3,4,5,6,7,8,9</sequence>
    <sequence name="k">0,1,2,3,4,5,6,7,8,9</sequence>
  </sequencelist>
</transaction>
```

This would generate requests for files at `http://www.cse.iitb.ac.in/shrirang/` having names from among *file200.html*, *file201.html*, ... , *file399.html* uniform randomly.

2.5 Resource requirements of AutoPerf

For experimental purposes, an Intel(R) Xeon(TM) dual CPU (each hyper-threaded) 2.80GHz machine was being used as load generator. However, it was observed that the values of CPU utilization on this machine reported by system utilities like *top*, *sar* were not consistent with their previously reported values under similar conditions. Hence, the CPU usage of the Master part of AutoPerf was measured on an Intel(R) Pentium(R) IV CPU 2.40GHz machine.

The CPU utilization of the Master part of AutoPerf at a given load level depends on the service demand of the server, think time of the users being emulated, network delay, etc. Too high a think time would result in a lower CPU utilization to be reported as compared to that at lower think time values. These two extremes would be observed when think time is infinity and when think time is such that the saturation load level of the server is the current load level itself, i.e. when, $Think_Time = Service_Time * (Current_Load_Level - 1)$. Also, CPU utilization would be high when load is being generated on a server with very low service demand. Here, resource requirements of AutoPerf were measured when load was being generated on a server with a very small and static response. The service demand observed for the profiled transaction was 1.15 milli seconds.

The profiler requires 3.94 KB of memory while the load generators memory usage increases at a rate of 520 KB per virtual user in addition to the fixed 270 MB. When load is generated on a very fast server with zero think time, the CPU utilization at the load generator increases at a rate of 11% per virtual user in addition to the initial 50% on an Intel(R) Pentium(R) 4 CPU 2.40GHz machine. It should be noted that the CPU utilization on the Intel(R) Xeon(TM) dual CPU (each hyper-threaded) 2.80GHz machine was much lower. Also, the CPU usage being

reported here is the worst possible CPU utilization profile as the measurements were performed for think time value zero and the server on which load was being generated was an extremely fast one with service demand of the transaction being profiled just 1.15 milli seconds. The profiler CPU usage is 0.15% of the server CPU on an Intel(R) Pentium(R) IV single CPU 1.50GHz machine.

Chapter 3

Experimental Results

This section illustrates the utility of AutoPerf in different modes of operation, mainly with the help of two systems: *SquirrelMail* - a webmail system, and *WebCalendar* - a PHP based calendar application. AutoPerf as discussed in section 2.3, consists of profilers deployed at the various server sites. The performance measurements performed by AutoPerf are an aggregate of those performed at the Master, which involve measures various client-side measures like throughput, response time, etc., and those performed at the various profilers. The measurements performed at a profiler could be service demand per transaction, CPU utilization, etc if it is a profiler for the Linux or Windows platform. If it is a Java Profiler (*JavaProf*[6]), average method level elapsed times at different physical as well as virtual resources are reported. Also, the message sequence chart, heap and non-heap memory usage and the garbage collection information is reported.

Sections 3.1, 3.2 and 3.3 discuss experiments performed using AutoPerf in it's various modes of operation.

3.1 Profiling mode, at a single load level

This is the simplest mode of operation of AutoPerf. In this mode of operation, AutoPerf generates load at a single load level. This section discusses the results obtained using AutoPerf's profiling mode of operation at a single load level. The experimental setup consists of a single machine which hosts a web server (*httpd*), an authentication server (*authdaemond*) and a courier TCP server which accepts IMAP connections (*couriertcpd*) as shown in figure 3.1. The various transactions possible in SquirrelMail are *Login*, *ReadMail*, *SendMail*, *DeleteMail* etc. The excerpts of results obtained by profiling the *ReadMail* transaction in *SquirrelMail* at different load levels (separately) using AutoPerf are summarised in table 3.1 - CPU service demand of each of the server-components, CPU utilization by each of the server-components, memory usage by each of the server-components, disk bytes read by each of the server-machines, disk bytes written by each of the server-machines, network bytes read by each of the server-machines and network bytes written by each of the server-machines - for the *Login* transaction and the *httpd* process (*Apache* web server). The resource profiles obtained at the other server-components (*authdaemond* and *couriertcpd*) are not shown in the table. AutoPerf also performs measurements of client-side measures which are not shown here to avoid replication of results in different profiling modes of operation. Client-side measures obtained using AutoPerf are discussed in section 3.2.

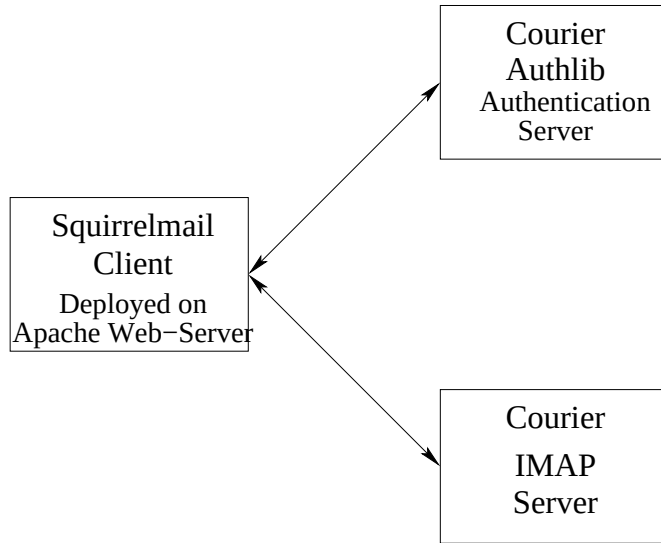


Figure 3.1: Deployment configuration of the webmail system.

User	S	U	M	DR	DW	NR	NW
1	220	1	153884	688	3040	2860	2505
3	200	0.91	218624	528	11248	8855	7715
5	190	0.93	234880	568	20232	13896	12307
7	157	0.95	267240	608	25520	15775	14058
9	142	0.96	315796	630	25864	18703	16634

S CPU Service demand in milliseconds
U CPU utilization
M Memory usage by the server in bytes
DR Disk bytes read by the server-process
DW Disk bytes written by the server-process
NR Network bytes read by the server-process
NW Network bytes written by the server-process

Table 3.1: Resource usage at Apache web server (*httpd*) for *Login* transaction in SquirrelMail

3.2 Profiling mode, at multiple load levels

This mode of operation differs from that discussed in section 3.1 only in the number of load levels at which load generation and profiling takes place. In this mode, profiling of transactions takes place over a range of load levels which is used to observe anomalous changes in the behavior of the transactions. For example, a marked increase in CPU service demand, when a very low rise is expected with an increase in load level, could signal a possible bug in the system. An example of such a bug could be an inefficiently written query, which causes a marked increase in CPU service demand at the database server with an increase in load level.

In section 3.1 we discussed the various server-side profiles obtained by AutoPerf. So, here we discuss with another example only the client-side measurements like the throughput and response time values at different load levels reported by AutoPerf. The system under test is WebCalendar application. The experimental setup consists of a single machine which hosts a Web server (*httpd*) and a database server (*mysqld*). The various transactions possible in *WebCalendar* are *Login*, *ViewByMonth* (view calendar entry by month), *ViewByWeek* (view calendar entry by week), *ViewByYear* (view calendar entry by year), *AddEntry* (add entry to a particular date in the calendar), etc. The client-side measures obtained using AutoPerf during the profiling of the *AddEntry* transaction of *WebCalendar* with think time set to 2.5 seconds are summarised in figure 3.2. The point of saturation and the load level up to which load is to be generated is automatically determined by AutoPerf. AutoPerf also reports server-side resource profiles as were discussed in section 3.1.

AutoPerf along with JavaProf [6] can be very useful to report average method level elapsed times at different physical as well as virtual resources, message sequence chart, heap and non-heap memory usage and garbage collection information of the system under test, details of which are described in [6]. [6] describes profiling of a banking application named *Duke's Bank* deployed on *JBoss* container, using JavaProf and AutoPerf in profiling mode at multiple load levels. The method level profiling is done only when a single user is using the system (i.e. when load level is one) due to profiler restrictions[6], whereas memory usage and garbage collection information is reported at all load levels. Tables 3.2, 3.3, 3.4 and 3.5 describe the message sequence chart, simple task elapsed times, heap and non-heap memory details and garbage collection statistics respectively for the *Atm_Page* transaction (request for the withdrawal/deposit page) in the *Duke's Bank* application as reported by AutoPerf when used with JavaProf.

3.3 Capacity analysis mode

In this mode, a CBMG is given as input to AutoPerf, thereby summarizing the workload characteristics. AutoPerf automatically determines the maximum throughput and saturation load level. It also reports resource profiles collected at the server.

Table 3.6 shows the CBMG used for the experiment. User think time was set to 1.5 seconds. The observed throughput curve is shown in figure 3.3. Figure 3.4 and 3.5 summarise the observed service demand and CPU utilization of the web server respectively. (Resource profiles obtained, like in table 3.1 are not shown again.)

CurrAppServ	CurrResource	CallToResouce	CallToAppServ	CallName
zipf	JVM	SessionBean	zipf	JVM_17
zipf	SessionBean	EntityBean	zipf	SessionBean_0
zipf	EntityBean	Database	zipf	EntityBean_1
zipf	Database	EntityBean	zipf	Database_2
zipf	EntityBean	Database	zipf	EntityBean_3
zipf	Database	EntityBean	zipf	Database_4
zipf	EntityBean	SessionBean	zipf	EntityBean_5
zipf	SessionBean	EntityBean	zipf	SessionBean_6
zipf	EntityBean	Database	zipf	EntityBean_1
zipf	Database	EntityBean	zipf	Database_2
zipf	EntityBean	Database	zipf	EntityBean_3
zipf	Database	EntityBean	zipf	Database_4
zipf	EntityBean	SessionBean	zipf	EntityBean_7
zipf	SessionBean	EntityBean	zipf	SessionBean_8
zipf	EntityBean	Database	zipf	EntityBean_1
zipf	Database	EntityBean	zipf	Database_2
zipf	EntityBean	Database	zipf	EntityBean_1
zipf	Database	EntityBean	zipf	Database_2
so on till 208				

Table 3.2: Message sequence chart : *Atm_Page* transaction in “*Duke’s Bank*”[6]

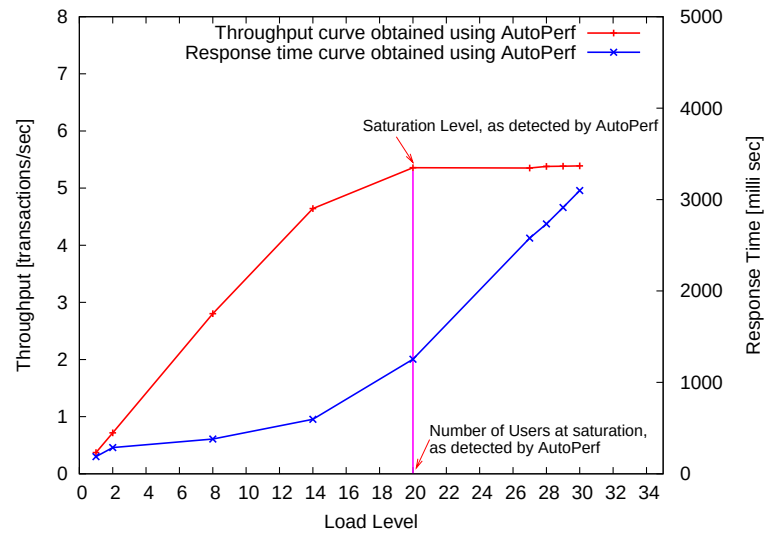


Figure 3.2: Throughput and Response time curves obtained using AutoPerf for the *WriteEntry* transaction in *WebCalendar*

OnAppSer.	CurrResource	CallToResouce	ToAppSer.	callName	ElapsedTime
					(milli sec.)
zipf	SessionBean	EntityBean	zipf	SessionBean_0	0.443404
zipf	EntityBean	Database	zipf	EntityBean_1	1.081198
zipf	Database	EntityBean	zipf	Database_2	0.461469
zipf	EntityBean	Database	zipf	EntityBean_3	0.114107
zipf	Database	EntityBean	zipf	Database_4	0.378689
zipf	EntityBean	SessionBean	zipf	EntityBean_5	0.905007
zipf	SessionBean	EntityBean	zipf	SessionBean_6	0.559039
zipf	EntityBean	SessionBean	zipf	EntityBean_7	0.945247
zipf	SessionBean	EntityBean	zipf	SessionBean_8	0.167965
zipf	EntityBean	Database	zipf	EntityBean_9	0.109672
zipf	Database	EntityBean	zipf	Database_10	0.445692
zipf	EntityBean	Database	zipf	EntityBean_11	0.106431
zipf	Database	EntityBean	zipf	Database_12	0.488048
zipf	EntityBean	Database	zipf	EntityBean_13	0.121841
zipf	Database	EntityBean	zipf	Database_14	0.711435
zipf	EntityBean	SessionBean	zipf	EntityBean_15	0.854809
zipf	SessionBean	JVM	zipf	SessionBean_16	0.335417
zipf	JVM	SessionBean	zipf	JVM_17	54.103405

Table 3.3: Simple task elapsed times : *Atm_Page* transaction in “*Duke’s Bank*”[6]

Load Level	HeapUsed	NonHeapUsed	HeapComit.	NonHeapComit.
	(KB)	(KB)	(KB)	(KB)
1	93643.634	54520.036	202768.384	54788.096
2	115486.416	54668.174	202768.384	54788.096
3	103280.364	54713.407	202768.384	54788.096
4	87187.594	54725.258	202768.384	54788.096
5	74828.555	54756.643	202768.384	54853.632
7	83819.256	54740.388	202768.384	54853.632
8	85356.092	54765.171	202768.384	54853.632
9	81747.429	54771.486	202768.384	55181.312
11	83456.867	54771.88	202768.384	54853.632
12	71832.47	54779.644	202768.384	55181.312
13	84027.338	54523.366	202768.384	54657.024
14	89175.411	54592.723	202768.384	54657.024

Table 3.4: Used Heap and Non Heap Memory at various load levels : *Atm_Page* transaction in “*Duke’s Bank*”[6]

Load Level	GC Frequency (per sec)	Percentage Runtime in GC (sec)
1	0.020025428	0.088470094
2	0.004409449	0.002813669
3	0.003131851	0.002501284
4	0.002803738	0.017103488
5	0.003700278	0.002786124
7	0.004028509	0.002746607
8	0.004599816	0.025363142
9	0.004855842	0.003185311
11	0.004575961	0.002981117
12	0.00513285	0.003661383
13	0.005074627	0.004313284
14	0.00480048	0.024357546

Table 3.5: Garbage collection statistics at different load levels : *Atm_Page* transaction in “*Duke’s Bank*”[6]

	Entry	Login	ViewByYear	ViewByWeek	ViewByMonth	Exit
Entry	0	1	0	0	0	0
Login	0	0.1	0.4	0.1	0.2	0.2
ViewByYear	0	0.1	0.2	0.2	0.4	0.1
ViewByWeek	0	0.1	0.3	0.4	0.1	0.1
ViewByMonth	0	0.1	0.3	0.2	0.2	0.2
Exit	0	0	0	0	0	0

Table 3.6: Customer Behavior Model Graph used during capacity analysis of *WebCalendar*

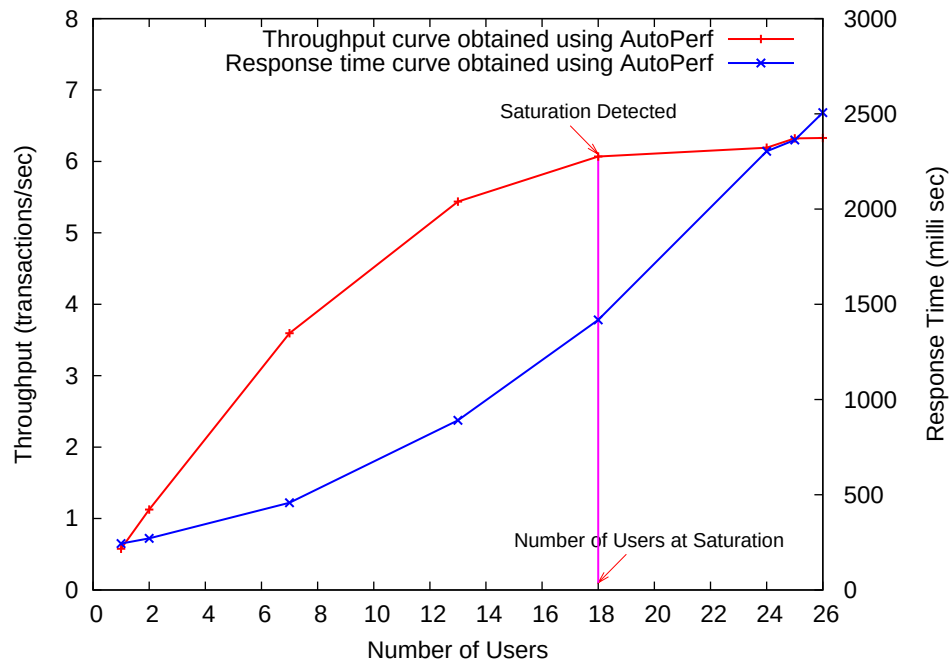


Figure 3.3: WebCalendar: Throughput curve obtained by AutoPerf during capacity analysis

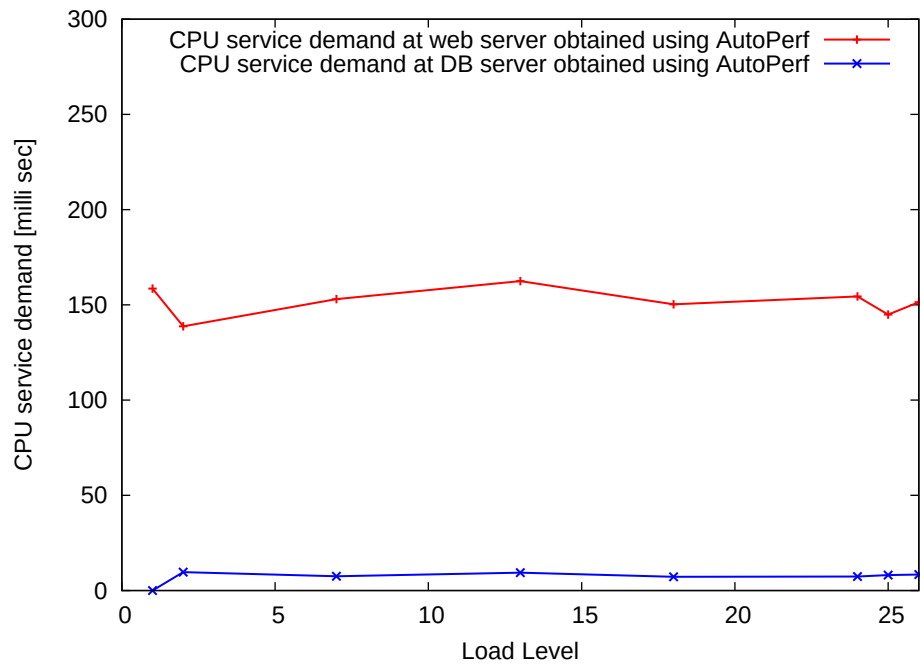


Figure 3.4: WebCalendar: CPU service demand at web server and database server obtained by AutoPerf during capacity analysis

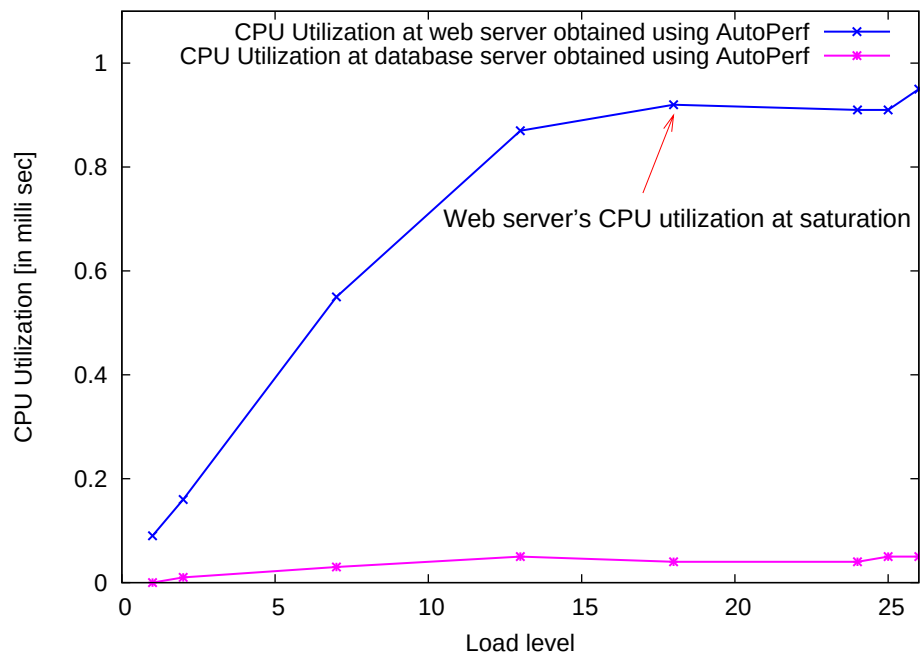


Figure 3.5: WebCalendar: CPU utilization at web server and database server obtained by AutoPerf during capacity analysis

Chapter 4

Validation of AutoPerf

This section describes the validation of the tool using two different approaches.

4.1 Validation using Mercury Load Runner

Figures 4.1 and 4.2 show the throughput and response time curves obtained using AutoPerf and Mercury Load Runner for a *Login* transaction in WebCalendar with a think time of 400 milli seconds. As can be seen, the curves obtained using Load Runner and AutoPerf are comparable. Thus, we successfully validate the throughput and response time measures obtained using AutoPerf.

4.2 Self-Validation

As discussed in section 2.3, AutoPerf consists of three modules viz. controller, load generator and profilers. The Master determines the observed throughput and the slave profilers determine the resource utilizations and service demands at the servers independent of the measurements at the Master. Table 4.1 shows the observed CPU service demand and the CPU utilization for a *Login* transaction in SquirrelMail at the Apache web server (*httpd*) and the observed throughput when load was generated with think time 3 seconds. According to utilization law [9],

$$Utilization\ of\ resource\ i = System\ Throughput * Service\ Demand\ of\ resource\ i$$

Load Level	CPU utilization at web server	CPU service demand at web server in seconds	Throughput in transactions/second
1	0.2	0.69	0.28
2	0.21	0.40	0.53
3	0.37	0.47	0.78
4	0.49	0.50	1.02
5	0.52	0.49	1.08

Table 4.1: AutoPerf results while profiling a *Login* transaction in SquirrelMail.

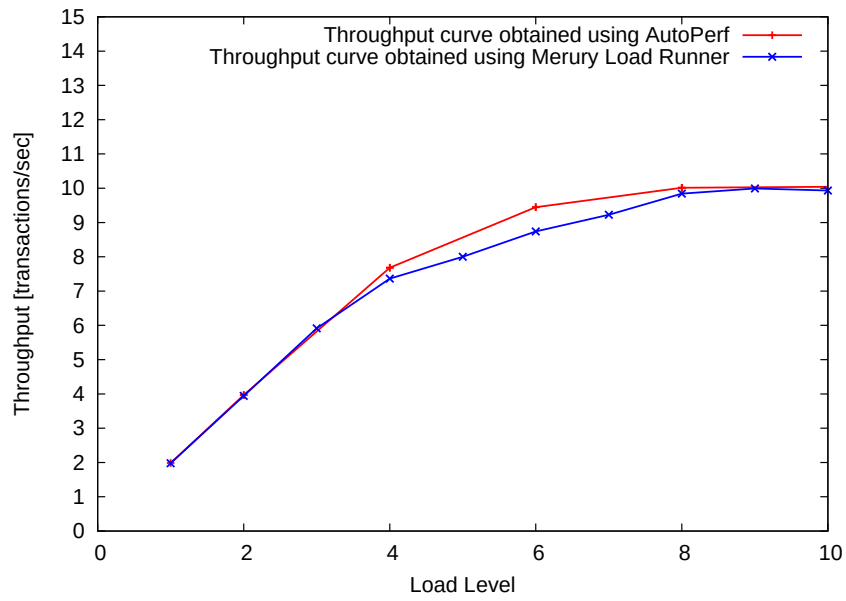


Figure 4.1: Throughput curves obtained using AutoPerf and Mercury Load Runner

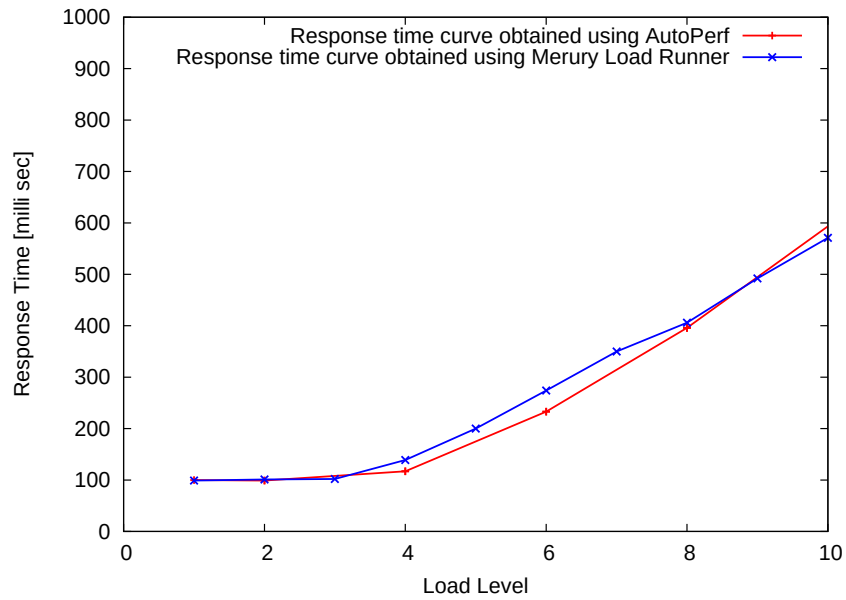


Figure 4.2: Response time curves obtained using AutoPerf and Mercury Load Runner

AutoPerf can be validated by applying utilization law to the values shown in table 4.1. This can serve as a primary validation, because the measures at client and server are obtained in a completely independent manner by the master and slave modules respectively and the throughput values have been validated as described in section 4.1.

Chapter 5

Future Work

- Real Servers would generally be deployed over a distributed system consisting of high configuration machines and with enough load balancing measures taken so that a particular resource on some machine does not get overloaded. Such systems would have very high capacity as compared to a client emulating virtual users over a single machine. Generating load up to the system saturation load level could require more than one client machine over which virtual users would be emulated by AutoPerf. AutoPerf should itself determine the maximum number of virtual users that can be emulated on a machine, so that the client machine itself does not become a bottleneck.
- During load generation, executing the same URL by the load generator, may cause the result of the URL/request to get cached at the server. As a result, the processing of the request is done only at the time of the first request and for all further requests the cached results are sent in response to the request. This will lead to misleading results. Hence, support for dynamic URL generation is very useful, thus causing different URLs to be requested each time. The current support for dynamic URL generation is minimal, only to illustrate the AutoPerf's capability of having such a feature. However, a more powerful way of specifying the URLs to be dynamically generated needs to be developed and supported. This could include an iterator to iterate over a range of integer/float values of variables or a way to read values of variables from files. The variables could then be used to generate dynamic URLs.
- It is important to define confidence in the obtained measures, if the results are to be properly analyzed, interpreted and used. For this, AutoPerf needs to support confidence intervals.
- AutoPerf should have a provision to specify unsuccessful responses, (For example, web responses from a particular server are unsuccessful if they contain strings of the form "*ERROR*".) so that AutoPerf can differentiate between successful responses and failed ones. Error handling is hardcoded (for the string "ERROR" in the server response) currently for the system under study.

Bibliography

- [1] Flood - a profile-driven HTTP load tester. <http://httpd.apache.org/test/flood/>.
- [2] Httpperf. <http://www.hpl.hp.com/research/linux/httpperf/>.
- [3] JMeter - Apache JMeter. <http://jakarta.apache.org/jmeter/>.
- [4] Mercury LoadRunner.
<http://www.mercury.com/us/products/performance-center/loadrunner/>.
- [5] Torture-Testing Web Servers. <http://stein.cshl.org/~lstein/torture/torture.html>.
- [6] Rishi Gupta. JavaProf: A resource usage profiler for Java software systems. Master's thesis, IIT Bombay, 2007.
- [7] JupiterResearch. Retail Web Site Performance, Consumer Reaction to a Poor Online Shopping Experience. Technical report, 2006.
- [8] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. Tata McGraw-Hill, 2000.
- [9] Daniel A. Menasce and Virgilio A. F. Almeida. *Scaling for E-Business*. Microsoft Press, 2000.
- [10] B. Nagaprabhanjan. Automated fine grained resource discovery of distributed transactions. Master's thesis, IIT Bombay, 2005.
- [11] B. Nagaprabhanjan and Varsha Apte. A Tool For Automated Resource Consumption Profiling Of Distributed Transactions. In *International Conference On Distributed Computing and Internet Technology*. Springer-Verlag, 2005.
- [12] Vaibhav Selot. Automated Tool For Resource Profiling And Capacity Analysis of Distributed Systems. Master's thesis, IIT Bombay, 2006.
- [13] Shirang Shirodkar and Varsha Apte. AutoPerf: An Automated Load Generator and Performance Measurement Tool for Multi-tier Software Systems. In *Proceedings of the 16th international conference on World Wide Web*. ACM Press, 2007.
- [14] Daniel Tkach. Application Server Platform Management Cost Comparison. Technical report, 2005.
- [15] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons, 2002.
- [16] Rukma Prabhu Verlekar and Varsha Apte. A Methodology and Tool for Performance Analysis of Distributed Server Systems. In *Proceedings of ICSE Conference*, 2006.

Appendix A

AutoPerf: Sample Input File

In this report we discussed various modes of operation of AutoPerf. In all its modes of operation, AutoPerf reads as input an XML file which primarily contains the *workload description*, *user behavior description* and *deployment configuration*. Here we will see a simple sample input to AutoPerf.

```
<AUTOPROF>
```

```
<LG>
```

```
<transaction>
```

```
  <name>Login</name>
```

```
  <url method="post">http://10.129.112.167/login.php?login=admin</url>
```

```
</transaction>
```

```
<transaction>
```

```
  <name>ViewEntryByYear</name>
```

```
  <sequencelist>
```

```
    <sequence name="y1">0,1,2</sequence>
```

```
    <sequence name="y2">0,1,2,3,4,5,6,7,8</sequence>
```

```
  </sequencelist>
```

```
  <initialisation>
```

```
    <url method="post">http://10.129.112.167/login.php?login=admin</url>
```

```
  </initialisation>
```

```
  <url>http://10.129.112.167/year.php?year=20${y1}${y2}</url>
```

```
</transaction>
```

```
<transaction>
```

```
  <name>ViewEntryByWeek</name>
```

```
  <sequencelist>
```

```
    <sequence name="y1">0,1,2</sequence>
```

```
    <sequence name="y2">0,1,2,3,4,5,6,7,8</sequence>
```

```
    <sequence name="month">01,02,03,04,05,06,07,08,09,10,11,12</sequence>
```

```
    <sequence name="d1">0,1,2</sequence>
```

```
    <sequence name="d2">1,2,3,4,5,6,7,8,9</sequence>
```

```

</sequencelist>

<initialisation>
    <url method="post">http://10.129.112.167/login.php?login=admin</url>
</initialisation>
    <url>http://10.129.112.167/week.php?date=20${y1}${y2}${month}${d1}${d2}</url>
</transaction>

<transaction>
    <name>ViewEntryByMonth</name>
    <sequencelist>
        <sequence name="y1">0,1,2</sequence>
        <sequence name="y2">0,1,2,3,4,5,6,7,8</sequence>
        <sequence name="month">01,02,03,04,05,06,07,08,09,10,11,12</sequence>
        <sequence name="d1">0,1,2</sequence>
        <sequence name="d2">1,2,3,4,5,6,7,8</sequence>
    </sequencelist>

    <initialisation>
        <url method="post">http://10.129.112.167/login.php?login=admin</url>
    </initialisation>
    <url>http://10.129.112.167/day.php?date=20${y1}${y2}${month}${d1}${d2}</url>
</transaction>

<farmer>
    <name>Farmer1</name>
    <thinktime>1000</thinktime>
    <executioncount></executioncount>
    <usetransaction>Login</usetransaction>
</farmer>

<farmer>
    <name>Farmer2</name>
    <thinktime>2000</thinktime>
    <executioncount></executioncount>
    <usetransaction>ViewEntryByMonth</usetransaction>
    <usetransaction>ViewEntryByYear</usetransaction>
</farmer>

<farm>
    <name>Message</name>
    <usefarmer count="2">Farmer1</usefarmer>
</farm>

</LG>

<Profiler>

```

```

JavaNode">
  <Node>10.129.112.167</Node>
  <Process>httpd</Process>
  <Port>20210</Port>
</NodeInfo>

<NodeInfo type="nonJavaNode">
  <Node>10.129.112.167</Node>
  <Process>mysqld</Process>
  <Port>20211</Port>
</NodeInfo>

</Profiler>

<warmup>true</warmup>

</AUTOPROF>

```

As shown, a transaction definition consists of the name to the transaction, which acts as an identifier for that transaction and the URL of the web-request.

```

<transaction>
  <name>Login</name>
  <url method="post">http://10.129.112.167/login.php?login=admin</url>
</transaction>

```

Sometimes, executing a transaction requires some series of other transactions to be executed before it. An example of such a scenario can be seen when a transaction requires us to first log into the system. We can specify all the transaction that need to be executed before the transaction that is to be profiled in the ‘initialisation’ tag, as shown below.

```

<transaction>
  <name>ViewEntryByYear</name>
  <sequencelist>
    <sequence name="y1">0,1,2</sequence>
    <sequence name="y2">0,1,2,3,4,5,6,7,8</sequence>
  </sequencelist>

  <initialisation>
    <url method="post">http://10.129.112.167/login.php?login=admin</url>
  </initialisation>
  <url>http://10.129.112.167/year.php?year=20${y1}${y2}</url>
</transaction>

```

All the measurements are begun only when all the ‘initialisation’ transaction are executed. Also, URLs can be generated dynamically using variables (*y1*, *y2* in this example) as shown.

We define various transactions which are part of the system under test. We can then define a 'farmer' as a list of transactions to be profiled. The input XML file can contain more than one farmer definitions. However, which farmer definition to pick for profiling is decided by the `<usefarmer>` tag.

The following line tells AutoPerf that the load level is 2, i.e. 2 concurrent users will be using the system under test.

```
<usefarmer count="2">Farmer1</usefarmer>
```

The `<execution\_count>` tag can be left blank as shown here. In such a case, number of repetitions of the transaction(s) being profiled is decided by AutoPerf itself. However, if a value is specified in the `<execution\_count>` tag, then each user generated those many number of transactions.

The deployment configuration is specified using the `<NodeInfo>` tags as shown above. The following snippet forms the basic unit of deployment configuration:

```
<NodeInfo type="nonJavaNode">
  <Node>10.129.112.167</Node>
  <Process>mysqld</Process>
  <Port>20211</Port>
</NodeInfo>
```

The `<port>` tag here refers to the port on which the profiler deployed on the machine 10.129.112.167 listens for commands from the Master. 'mysqld' is the process that constitutes the server and will be profiled. The 'type' field indicates the type of profiler it is - A Java profiler (*JavaProf*[6]) which performs fine-grained profiling or a profiler for Linux or Windows platform.

Appendix B

AutoPerf: Usage

As discussed in section 2.2, AutoPerf can operate in three modes. The following sections describe the commands to invoke AutoPerf in each of these modes.

B.1 Profiling mode and at a single load level

AutoPerf can be invoked in this mode by the following command:

```
java Master
```

This will generate load only at the level specified in the input XML file using the `<usefarmer count=XX>` tag. This will cause only the specified transaction to be requested and profiled.

B.2 Profiling mode and at multiple load levels

AutoPerf can be invoked in this mode by the following command:

```
java Master -c
```

This will generate load at multiple load levels up to the point of saturation. Load generation begins at the load level specified in the input XML file using the `<usefarmer count=XX>` tag. This will cause only the specified transaction to be requested and profiled.

B.3 Capacity analysis mode

AutoPerf can be invoked in the capacity analysis mode of operation by the following command:

```
java Master -c -p
```

This will generate load at multiple load levels up to the point of saturation. Load generation begins at the load level specified in the input XML file using the `<usefarmer count=XX>` tag. In this mode a mix of transactions are requested and profiled. The load generated is a mix of transactions as is summarised by the CBMG.