A Tool For Automated Resource Consumption Profiling of Distributed Transactions

B. Nagaprabhanjan and Varsha Apte

Indian Institute of Technology Bombay, India
nagaprabhanjan@it.iitb.ac.in, varsha@cse.iitb.ac.in

Abstract. In this paper, we present a tool, called Autoprofiler, that automates the discovery of resource consumption by transactions on distributed systems. Such information is required as input to performance analysis tools, which may be used for capacity planning, for re-architecting a distributed system, or to identify potential bottlenecks. Deriving this information using existing tools is a tedious and error prone process. In contrast, our tool requires minimal human intervention, and brings down the time required to profile complex distributed systems to a few minutes. It does this by co-ordinating the process of load generation and server resource profiling. Our tool also works with a Java profiler, called LiteJava Profiler, which we have built, to fully automate the process of resource consumption discovery for J2EE servers.

1 Introduction

With the advent of the Internet, the trend of business transactions between customers and enterprises has changed drastically. Now, a customer expects to perform a business transaction within seconds from home through the Internet. The IT infrastructure of an enterprise should be robust enough to provide such service to its customers. As distributed computing systems have been proven to be a low cost and high performance alternative to centralized systems, most of the Internet based services are supported by distributed systems. An enterprise can also choose to consolidate all their applications and services in high-performance data centers. In either case, a typical transaction always accesses multiple nodes before completion. For example, a transaction to transfer money from one account to another may access the authentication server for verifying the credentials of a person, the database server to access the accounts and the application server for processing the business logic.

In such a scenario, it is necessary for an enterprise or the owner of a data center to make sure that the user performance requirements of a service, such as response time, throughput, connection loss rate etc. are met. At the same time, it is desirable that the IT infrastructure is utilized well and the systems operate close to their capacity. The process of sizing the infrastructure so that service performance requirements are met, is called “capacity planning”. Capacity planning can be carried out if the resource requirements of individual applications are known, along with the usage volumes (or *load*) that the applications need to support.

One way to answer a capacity planning question, such as how many machines of a certain type are required to support a certain application, is the following: If resource utilization and load measurements are available from a system where this application is currently deployed, a simple linear scaling can be used for future capacity planning. E.g. if it is known that application server A deployed on Machine M, is supporting 1000 customers, with machine M's utilization level at 80%, then if the number of customers is projected to reach 2000, planning for one more machine of the same type will be needed.

However, this approach works only if the nature, or the *mix* of transactions coming to the system can be assumed to be unchanging. If the workload mix on a system changes, then linear scaling methods do not work. In such cases, a more detailed model of the system, and a fine-grained resource consumption profile of the transaction is required. That is, we need to know what the resource requirement of each *type* of transaction is, not the application server as a whole. More specifically, a complete resource consumption profile, such as the CPU time required by each transaction of the application, the memory requirement, the disk and network I/O, is required, along with a characterization of the workload (e.g. rate at which the requests for different transactions arrive). With these inputs, analytical queueing models (or simulation models), can be used to arrive at an optimal sizing and deployment configuration of the applications on the infrastructure [13].

Several modeling tools exist that accept the resource requirements, the message flows and the deployment configuration of applications on physical resources as inputs, and solve an underlying queueing model to provide performance measurements [13]. However, in realistic scenarios, these inputs may not be readily available, and effort must be made to explicitly discover them. Of the three mentioned above, the deployment details are the easiest to obtain; however, derivation of the message flows and resource requirement may be more involved. The process of discovering resource consumption for multiple transactions that access multiple servers can be quite tedious and error prone. Thus, there is a need for designing good software tools that make this process smoother, error-free and requiring minimal human involvement.

In this paper, we describe a tool that automates this process of resource usage profiling for distributed applications which have a Web-based front-end. Given a simple deployment description, and the URIs of the Web-transactions, the tool generates a resource consumption profile of all specified servers. The tool includes two components: the wrapper component, which we call the *Autoprofiler* which automates and co-ordinates the entire process; and a Java-specific profiler called the *LiteJava Profiler*. This was required since direct interaction with JVM internals is required to produce a fine-grained profile of a Java application.

We note here that none of the existing load generators are built for this purpose; they are primarily "performance testing" tools - a purpose which is distinctly different from resource profiling. In case of Java profilers, none of the exisiting Java profilers fit into the automated framework that we were building, therefore we built a custom profiler (LiteJava Profiler), which is flexible and scalable, to suit our requirements. In our experiments with Autoprofiler working with the LiteJava Profiler, the time required to profile a sample J2EE application called ECPerf reduced dramatically. Thus, our tool can significantly enhance the productivity of an IT enterprise.

The rest of the paper is as follows. In Section 2, we present some commonly available load generators and resource profilers and discuss their limitations. In Section 3 we present Autoprofiler, a framework to automate the process of load generation and resource profiling. We discuss the LiteJava Profiler in Section 4. In Section 5, we present the results of some preliminary experiments conducted with the tool and discuss some observations that we found in Java based applications. We conclude in Section 6 with a discussion on future work.

2 Background and Motivation

In order to better understand the need for automating the process of resource consumption profiling, consider the scenario in Figure 1. Here, we have deployed the *ECPerf* [1] application on the JBoss application server on one machine and the PostgreSQL database server on another machine. The *ECPerf* application

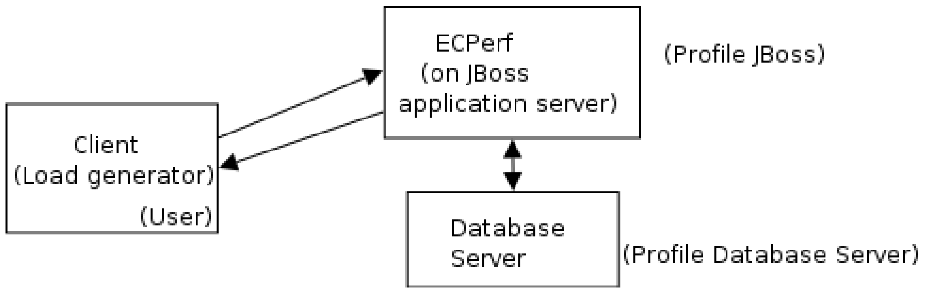

Fig. 1. A Resource Usage Profiling Scenario

supports services such as creating a new order, getting the status of an order, getting the status of a customer, cancelling an order, scheduling a work order, updating a work order etc. [1].

Now in order to find, for e.g., the CPU time consumed by each of the above transactions on each of the servers, we need to do the following:

- Generate the requests for a particular transaction using some load generator.
- After some warm-up time, start measurements on both machines to profile the corresponding processes.
- When the load generation is over, stop profiling the processes on the server machines.

– Collect the statistics from the load generator as well as the profiling processes. Correlate it (e.g. at request rate x, CPU utilization is y).
– Carry out the calculations required to get the resource consumption values for a single request.
– Repeat these steps for each transaction that the application supports.

Currently, in order to carry out the above steps, we need two distinct kind of tools *viz. load generators* which generate load on the system and *server profiling tools* which give us the resource consumption measurements from the server. However, the co-ordination between the load generation and the server resource profiling process needs to be done manually. Furthermore, the actual calculations needed to derive per-transaction resource consumption, also must be done manually. This increases the time required to generate a complete resource consumption profile of the server and increases the possibility of errors that occur in the co-ordination process. As the service becomes more distributed in nature, with a large number of components, the process of manually profiling becomes more time consuming, tedious and error prone.

Most of the work mentioned above can and should be automated. This is what we have aimed for in our tool *viz.* Autoprofiler. The tool should generate the load for the transaction. After detecting sufficient warm up for the transaction, it should automatically start measurement at the server end. When the load generation process is over, it should get the resource consumption details from the servers, normalize the values and display the same. In case the server is a Java based server, the tool should also interact with the Java profiler to get fine grained details.

2.1 Existing Tools

There are a number of commercial Web load generators available in the market, e.g. *PureLoad*[2], *openSTA*[6] and *Httperf*[3]. Products such as *Silk Performer*[4] and Mercury LoadRunner[5] are commercially available enterprise class tools. However, these are all primarily *performance testing* tools that focus on recording and analyzing client-side performance measures. Although some do provide consolidated views of client-side performance measures along with server side resource utilization measures, none of them do any co-ordination or calculations necessary to provide per-transaction resource consumption details.

Some common OS utilities provide comprehensive resource consumption information. *Top* displays the dynamic values of the system state, such as CPU utilization, memory consumed etc. on a per process basis. Additionally, utilities such as *iostat*, *vmstat*, *netstat*, *sar*, *ps* provide information similar to *top*. The *Linux Trace Toolkit* [7] can give finer information such as time spent in I/O etc. by a process.

There are a number of profilers available for the Java environment. The *Extensible Java Profiler* (EJP)[8] is a Java profiling tool that enables developers to test and improve the performance of their programs running on the JVM. It has filtering capability allowing one to log only methods of specified packages.

Yourkit Java Profiler (YJP) [9] gives useful profiling information regarding the heap (memory). Using this profiler, one can get the CPU and memory allocation information about the application. It has support for partial profiling which means that profiling can be enabled or disabled as and when required. Details about more profilers can be found in [10]. The main drawback of these profilers is that they are written for simple to moderate-sized applications, and meant to be used in an interactive manner by humans. Hence, for e.g., they have very user-friendly graphical interfaces. However, they fail to scale up when deployed and run in a J2EE environment, requiring overheads such as a long start-up time and a large amount of disk space for the large amount of data generated. Furthermore, they are not built for automation, which makes them unsuitable for our purpose.

3 Autoprofiler

The Autoprofiler is a distributed tool with a master-slave architecture, that co-ordinates the process of load generation and resource profiling. In master mode, the tool generates the load and is responsible for co-ordinating the process. In slave mode, the tool records the resource utilization information and sends this to the master when asked for. This basic co-ordination process for a general distributed system is depicted in Figure 2.

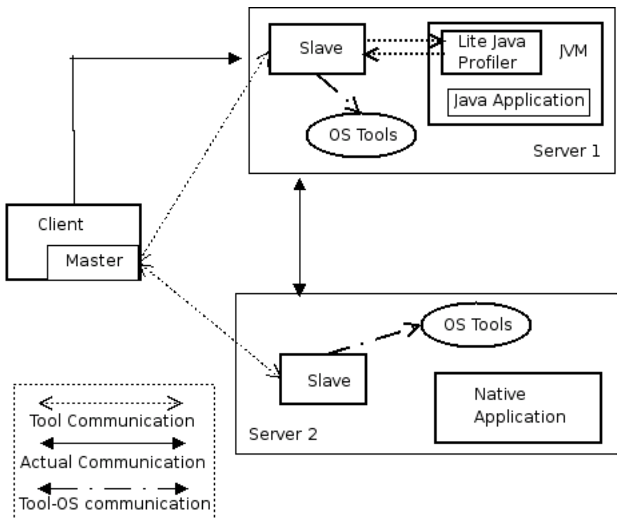

Fig. 2. Autoprofiler: Architecture and process co-ordination

Here, the master resides at the client side and the slaves at the servers. The client side is the one where load is generated and the server side is the one where the application servers reside that serve the requests.

In the first version of the tool, we have focussed mainly on resource profiling. We aim to get only raw service times for requests and do not want to put the resources under contention. In order to ensure this, the tool generates the requests sequentially; that is, only after the reply for a request arrives, it generates the next request. These raw service times will serve as input to queueing models from which the performance attributes of the system such as the average number of jobs, average waiting time etc. can be derived.

We now discuss the architecture of the tool in detail. As the tool works in two modes viz. master and slave, we discuss their architectures seperately.

3.1 Master architecture

The tool, as a master, does the following:

- Reads the information about the transactions from an XML file.
- Generates the corresponding HTTP requests.
- Co-ordinates the process of resource profiling by issuing commands to the slaves.
- Displays the resource profile summary on the master terminal.

We elaborate further on some of the elements of the Master.

Input Specification Formats: The tool needs inputs such as the description of the transactions, the software servers and their deployment details. We have created an XML DTD that allows us to specify this information. Figure 3 shows the XML DTD which we explain in detail as follows:

Transaction Specification: As can be seen from the DTD, the basic information about a transaction comprises of:

- The transaction name.
- The Web interface for the transaction.
- The information about the applications, and their deployment information, that are accessed by the transaction.

The Web interface comprises of the node and port on which the Web server is running. This information is given as url and port respectively. It also includes the Uniform Resource Identifier (URI) information which specifies what needs to be accessed.

Specification of URI: A static URI is specified as uri in the XML file. If variable URIs are to be sent (e.g. URIs with "name-value" pairs), the name of a file of such URIs is given instead, as uriFile in the XML specification.

Node information: The tool also needs to know the information about the nodes that are accessed by a particular transaction so that it can communicate

```
<!DOCTYPE services [
<!ELEMENT services (TransInfo+)>
<!ELEMENT TransInfo (Name, WebInterface, NodesInfo+)>
<!ELEMENT WebInterface (url,port,(uri|uriInfoFile))>
<!ELEMENT NodesInfo (nativeNode|javaNode)>
<!ELEMENT nativeNode (Node, Process)>
<!ELEMENT javaNode (Node, Process, ComponentInfo?)>
<!ELEMENT ComponentInfo (Component+)>
<!ELEMENT Component (Name,Interface+)>
<!ELEMENT Node (#PCDATA)>
<!ELEMENT Process (#PCDATA)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT port (#PCDATA)>
<!ELEMENT uri (#PCDATA)>
<!ELEMENT uriInfoFile (#PCDATA)>
<!ELEMENT Interface (#PCDATA)>
]>
```

Fig. 3. The XML DTD

with the slaves at the respective nodes. The tool identifies two different types of nodes. One is a *Java node* and the other one is a *native node*.

A *Java node* is one which hosts a Java based application server such as *Tomcat*, *JBoss* etc. On a *Java node*, we have the Java based profiler running inside the JVM which does profiling on a per method basis. A *native node* is one which hosts any other conventional application server. Irrespective of whether a node is a *Java node* or a *native node*, the name of the process which serves the transaction on that machine needs to be specified. If there is more than one process serving the transaction on a given node, then the two processes are mentioned as seperate entries. If the node is a *Java node*, then more information is specified. We will discuss this in Section 4. If a node has both Java and native servers running on it and they need to be profiled, then they are specified as two separate entries in the XML file.

Procedures done by the Master: After reading the input in the XML format, the Master carries out the following procedures:

Request Generation: The request generator generates the requests sequentially. It reads the transaction information and frames the URL accordingly depending on the type of transaction.

Warm up Detection: We use the technique of *moving average* to detect the steady state of the server. As the request generator receives the responses from the server, it records the response times. We define a window of size w and calculate and store the average of the last w values. If the normalized difference between the two averages is less than a given ϵ, then we conclude that the server has sufficiently warmed up and we start taking measurements from that point onwards.

If the value of w is small, then we may see too many fluctuations between successive values and if the value of w is large, then we may not see any fluctuations at all. We have chosen w to be 5, which successfully detected warm-up in our experiments. This method is loosely based on Welch's method for discarding the initial transient and detecting the steady state [11].

If the successive moving averages do not differ by more than $15\%(\epsilon \leq 0.15)$, then we conclude that the server has sufficiently warmed up.

Master-Slave Co-ordination: The main purpose of Autoprofiler is to co-ordinate the process of load generation with the resource profiling. When the load generator detects sufficient warmup of the server, commands are sent to the slaves to start profiling. When the load generation process is over, the master again sends commands to get the profiling data from the slaves.

The profiling process can be summarized as follows:

- Master reads the transaction information from the XML file.
- Master starts generating the load.
- When the server has warmed up, commands are sent to the slaves to start profiling.
- When the load generation process is over, commands are again sent to slaves to get Che profiling data.

As can be seen, the entire profiling process as described in Section 2, that was being done manually is now fully automated by the Autoprofiler.

3.2 Slave Architecture

The tool when working as a slave, interacts with the OS tools as well as the in-process Java profiler to control the profiling. It works in the passive mode; that is, it waits for the master to send commands and then acts on them.

Interaction with the OS tools: The resources that are profiled by the tool are CPU time consumed, disk I/O, network I/O. Since the tool is developed for a Linux environment, we use tools that are available with every Linux distribution. We use ps for the CPU time consumed, vmstat for the details about disk I/O and netstat for the details about network I/O. Since the values given by the tool are cumulative, the slave takes the snapshot of the values at the beginning of the profiling process and at the end of the profiling process.

Interaction with the Java profiler: The slave controls the in-process Java profiler. The slave is responsible for initializing and resetting the profiler state, sending the necessary data to the profiler and receiving the data from the profiler. The Java profiler is explained in detail in the next section. Figure 2 shows the overall architecture of Autoprofiler integrated with the Lite Java Profiler.

4 Lite Java Profiler (LJP)

We have developed an in-process Java profiler, called the LiteJava Profiler, (LJP), which can give fine grained information such as per method CPU information, memory allocated in the JVM and garbage collection information by using JVMPI [12]. LJP interacts with the AutoProfiler slaves, and hence fits into the overall automated framework. LJP is "lighweight"; i.e., it has low profiling overhead, does not generate unreasonably large amounts of data, and does not require disk I/O.

4.1 Features

We wrote LJP to overcome the limitations of the existing Java profilers. The enhanced features that are unique to our tool are:

– *Partial profiling:* This means that the profiling process can be stopped and started whenever required. This is necessary especially in J2EE kind of environments where continuous profiling can result in very large overhead which can affect the measurement process.
– *Remotely controllable:*. LJP can be controlled from a remote machine. This feature is required for automating the process of resource profiling.
– *Dynamic filter support:* LJP supports dynamic filtering. Filters are set to profile only selected methods and hence reduce the profiling overhead. The profilers we surveyed either had static filtering option where it obtains the filtering information once at start-up or provided filtering only at the display stage.
– *No File I/O:* LJP does not do any file I/O since it can cause unnecessary overhead. It maintains in-memory data structures to store all the intermediary data.
– *Garbage collection information:* As the garbage collection process happens asynchronously in the JVM, it can indeed affect the response time of a transaction. LJP separately profiles the garbage collection process and reports the statistics.

4.2 Implementation Details

Partial profiling and Remote controllability When the JVM is started, the LJP spawns a separate thread and starts listening for commands from the master. The LJP works with the JVMPI, which generates various events. Only the class_load event is enabled in the beginning (i.e. before LJP is asked to start profiling). This lets LJP build the mapping between the method names and method IDs. This is stored internally by the LJP as a hash map, which is required because method events only include method IDs. Other events, such as method entry and exit, object allocation and garbage collection that are used for the actualy profiling are enabled and disabled based on commands from the master. Thus there is no unnecessary start-up overhead. The master communicates with the LJP through the slave. This "partial" profiling results in huge saving of overheads, since one can choose not to profile the thousands of methods that are called at start-up by the J2EE application server such as JBoss, and instead start profiling only when transactions start coming to the server.

Dynamic Filtering Support Filters are necessary to avoid unnecessary profiling overhead, so that, for e.g., one can profile only the application-specific methods, and not the internal methods called by the application server. However, when profiling the server with minimal human interaction, the flexibility of changing the filter without re-starting the JVM and the application server, is required. Thus dynamic filtering support has been provided in LJP. It uses a set of in-memory data structures to maintain the filter information, thus no files need to be read. The filter information is provided in the form of class names and the corresponding method names. Whenever such a method executes, its execution time is recorded. The specific components and their corresponding methods are specifed in ComponentInfo element in the XML file.

Garbage Collection As already mentioned, garbage collection can happen asynchronously during the load generation period and hence can affect the response time. We capture the garbage collection information and report it along with the other profile details. This is done by enabling the gc_start and gc_finish events in the JVMPI.

5 Experiments and Observations

To test the tool, we profiled *ECPerf*, a sample J2EE application which was written to serve as a benchmark for evaluating different J2EE servers. We deployed ECperf application on the JBoss application server. The experimental setup is as depicted in Figure 1.

The database server is hosted on a uni-processor Pentium 4 machine and the JBoss application server on a dual processor Pentium 4 machine. All transactions access the database server for retrieving data. The resource consumption details of various transactions for ECPerf are as shown in Table 1.

Here S_t is the service time in milliseconds per transaction, N_i and N_o are number of bytes sent and received on the network, D_r and D_w are number of disk bytes read from and written to disk[1]. The values given by the Java profiler for individual transactions are shown in Table 2.

The entire profiling process, which generated 8 set of resource consumption measures, for 7 transactions, on 2 servers, took less than 10 minutes with the aid of the tool. If done manually, this experimentation, and the generation of results, can consume days.

[1] vmstat and netstat give values on an overall basis - not per-process. Thus the values obtained from vmstat and netstat include background disk and network activity. In the future versions of the tool, this "noise" will be removed.

T_N	R_t	JBoss Server					PostgreSQL				
	(ms)	S_t	N_i	N_o	D_r	D_w	S_t	N_i	N_o	D_r	D_w
		(ms)	(bytes)				(ms)	(bytes)			
Order Status	492.6	480.5	147.4	25.6	962.5	120.0	7.5	143.1	15.5	798.7	87.7
Customer Status	654.5	633.3	196.7	26.3	131.5	70.0	9.4	195.1	96.5	146.7	50.7
Cancel Order	620.8	616.5	196.4	28.6	172.5	30.0	3.5	184.1	11.5	652.7	273.7
Schedule Work Order	560.6	533.5	164.4	24.6	137.5	89.0	10.5	160.1	13.8	555.7	69.7
Update Work Order	572.4	566.67	168.4	26.6	106.5	49.0	6.5	165.12	10.5	603.7	47.7
Complete Work Order	1041.4	1033.3	302.4	45.6	233.5	273.0	10.5	304.12	19.5	1156.7	1501.7
Cancel Work Order	577.9	540.5	178.4	26.6	150.5	18.0	12.5	172.12	19.5	528.7	38.7

Table 1. Results

Transaction Name	Method Info			GC Time
	Method Name	CPU time (ms)	Memory consumed (kilobytes)	(ms)
Order Status	getOrderStatus()	32.7	16	43
Customer Status	getCustomerStatus()	48	26	44
Cancel Order	cancelOrder()	43.6	33	43
Schedule Work Order	scheduleWorkOrder()	24	17	42
Update Work Order	updateWorkOrder()	30.6	17	42
Complete Work Order	completeWorkOrder()	81	45	42
Cancel Work Order	cancelWorkOrder()	43	24	45

Table 2. Results from Java Profiler

Resource profiles such as this one can lead to helpful insights about the bottlenecks in a distributed system. For example, let us compare the CPU time taken by the application methods in the JBoss server (as shown in Table 2), with the total time consumed by the *java* process (as shown in Table 1). We can derive the time taken by the JBoss server methods by calculating the difference in these two measurements. We conclude that the JBoss application server contributes a large CPU time overhead (90% of the total time) while servicing a request. Similarly, we can observe that garbage collection time is more or less the same for all transactions.

6 Conclusion and Future Work

In this paper, we presented a tool that automates the process of fine-grained resource consumption profiling of distributed transactions. We believe that this tool represents a different paradigm - one in which performance measurement is done for the purpose of performance *modeling* and *prediction*, not simply for the sake of performance "testing". Existing tools do not significantly ease the process of gathering inputs that a performance analyst or a modeling tool would need, if performance is to be predicted beyond that which the load generator can measure. Sophisticated capacity planning requires the use of performance models, and performance models require inputs such as per-transaction and per-server resource consumption details.

We have built such a tool, and demonstrated that it can significantly reduce the time to produce such data, which will result in greater productivity of the performance analyst, and ultimately will cut costs of running large data centers.

Our experimentation also showed that resource profiling can reveal valuable insights about the performance characteristics of the applications under study - e.g. that application server overhead in J2EE servers can be large.

The tool can be enhanced in several directions. First, we can add innumerable features to it which enhance its usability for realistic Web-based systems. Second, the tool can be made more "intelligent" by having it automate several tasks that are a part of a measurement study; e.g. automatically characterize maximum throughput, find the bottleneck components, the maximum number of users that can be supported by the system, etc. Third, we can extend the tool so that it can work on high-level measurements obtained from a *production* environment. Generating resource profiles from measurements made in an uncontrolled environment is an interesting research problem that can be pursued further.

References

1. Sun Microsystems: The ECperf benchmark for evaluating J2EE servers (2003), http://java.sun.com/j2ee/ecperf/index.jsp.
2. Minq Software AB: Pure load (2003), http://www.minq.se.
3. David Mosberger: httperf (2003), http://www.hpl.hp.com/personal/David_Mosberger/httperf.html.
4. Segue Software Inc: Silk performer (2002), http://www.segue.com.
5. Mercury Interactive Corporation: LoadRunner (2002), http://www.mercury.com/us/products/.
6. openSTA: openSTA Software Testing Architecture (2002), http://www.opensta.org/.
7. OperSys: Linux Trace Tool kit (LTT) (2002), http://www.opersys.com/LTT.
8. Sabastien Vauclair: Extensible Java Profiler (2003), http://ejp.soureforge.net.
9. Yourkit: Yourkit Java Profiler (2003) http://www.yourkit.com.
10. B Nagaprabhanjan: Automated and Fine grained Resource Consumption Discovery of Distributed Transactions. Master's thesis, K.R.School of Information Technology, IIT Bombay (2004).
11. Law, A.M., Kelton, W.D.: Simulation modeling and analysis (2003).
12. Sun Microsystems: The Java Virtual Machine Programming Interface (JVMPI) (2004) http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.
13. Rolia, J.A., C.Sevcik, K.: The method of layers. In: IEEE Transactions on Software Engineering archive. Volume 21, Issue = 8 (August 1995). (1995) 689 – 700.