Evaluating Performance Attributes of Layered Software Architecture

Vibhu Saujanya Sharma[1], Pankaj Jalote[1], Kishor S. Trivedi[2]

[1] Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur, INDIA, 208016
{Vsharma, Jalote}@cse.iitk.ac.in
[2] Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA
Kst@ee.duke.edu

Abstract. The architecture of a software system is the highest level of abstraction whereupon useful analysis of system properties is possible. Hence, performance analysis at this level can be useful for assessing whether a proposed architecture can meet the desired performance specifications and can help in making key architectural decisions. In this paper we propose an approach for performance evaluation of software systems following the layered architecture, which is a common architectural style for building software systems. Our approach initially models the system as a Discrete Time Markov Chain, and extracts parameters for constructing a closed Product Form Queueing Network model that is solved using the SHARPE software package. Our approach predicts the throughput and the average response time of the system under varying workloads and also identifies bottlenecks in the system, suggesting possibilities for their removal.

1 Introduction

Software architecture is an important phase in software lifecycle as it allows taking early design decisions about a system. Moreover it is also the earliest point in system development at which the system to be built could be analyzed [7], [9]. Analysis of a system at the architectural level enables the choice of the right architecture for the system under consideration, thus saving major potential modifications later in the development cycle or tuning the system after deployment.

Out of the various attributes that could be assessed, performance attributes are most sought after in any software system. Performance is an umbrella term describing various aspects, such as responsiveness, throughput, etc. of the system. Assessing and optimizing these aspects is essential for the smooth and efficient operation of the software system. There have been many approaches [2],[3],[4],[5],[12] for performance evaluation of software systems, the pioneering work being done by C.U. Smith, [2] which introduced the concept of Software Performance Engineering (SPE).

Layered software architecture is a very prevalent software architectural style that is followed by almost all client-server and web based systems. Layered architecture

G.T. Heineman et al. (Eds.): CBSE 2005, LNCS 3489, pp. 66-81, 2005.

helps to structure applications that can be decomposed into n groups of subtasks in which each group is at a particular level of abstraction with well-defined interfaces [8]. The i^{th} layer could communicate with only the $(i-1)^{th}$ and the $(i+1)^{th}$ layer. Layered architecture is widely used in almost all web-based systems where performance is a critical factor. Hence, performance analysis of layered systems is of much importance to system architects. Moreover as a large number of layered systems already exist, performance predictions with varying number of clients or with the addition or scaling up of components in the system would be beneficial to system administrators, who manage such systems.

In this paper, we present an approach for performance evaluation of software systems following the layered architectural style. In the past SPE has been largely seen as an activity, which requires specialized skills and in-depth knowledge of both software architecture and performance modeling. Thus one of the motivating factors of our effort is to provide an approach that could be used by software engineers for designing new systems and system administrators for tweaking existing systems alike. The aim of the approach is to output the traditional performance parameters as well as suggest to the user bottleneck components that need to be scaled up. Our system thus removes the performance analyst from the loop in that the activities traditionally performed by him/her are automated.

Our approach consists of modeling the layered software system as a closed Product Form Queueing Network (PFQN) [1], and then solving it for finding performance attributes of the system. One of our aims is to ask for specifications that are easy to provide even for someone who is not an expert in this field. After getting the specifications we model the system initially as a Discrete Time Markov Chain (DTMC), with each layer in the system, corresponding to a state in the DTMC [13]. This DTMC is then analyzed to find the total service requirements of the software system over the different hardware nodes or machines. The closed PFQN model is then constructed using this information along with the specifications given by the user. Modeling machines having limited software resources such as threads is also performed at this stage using a hierarchical approach.

This closed PFQN model is then fed to SHARPE [11] which is a versatile software package for analyzing performance, reliability and performability models. The output from SHARPE is then further analyzed, and the results include the classical performance metrics such as the throughput and the average response time along with information about system bottlenecks and suggested scale-ups for them. Along with these, it predicts the improvement in system performance if the suggested scale-ups are done. This is done by reconstructing the model internally, accommodating the scaled-up components and solving it again, using our approach.

The tool, which we have developed as an implementation of this approach, requires minimal knowledge of queueing models or any other performance modeling techniques to use it. The specifications which are needed could be easily procured and hence the tool facilitates modeling new systems as well as helping in scaling existing systems.

2 Evaluating Performance of Software Architectures

System performance has become a major concern, as large, complex, mission critical and real time software systems proliferate in almost all domains. The inherent structural relationships among the components characterize the architecture that software system is following and usually could be classified into some well-known architectural styles [7], [8]. Since the performance attributes of a software system, such as number of jobs serviced per second (throughput), the average response time, etc. depend both on the time and resources spent in computation as well as in the communication between various components of the system, the architecture that a particular system follows has a lot of bearing on its performance. This forms the basis for the need of evaluating various software architectures for their performance attributes. In practice there are many questions that a performance assessment approach should answer. Some of the prominent ones are:

- What effect would varying the number of clients have on the throughput and the average response time of a particular system, if a particular architecture is followed?
- What would be the ideal number of clients the system would be able to handle before it saturates?
- Which software component should be allocated to which hardware node?
- What would be the bottlenecks in the system and how could they be removed?
- What would be the change in the performance attributes if a system component is enhanced or scaled up?

In the recent past there have been some efforts towards answering such questions and concerns regarding software architectures. A methodological approach for evaluation of software architectures for performance was first proposed by C.U. Smith in her pioneering work [2] and later with L.G. Williams [10] and is called Software Performance Engineering (SPE). Two models represent the system in SPE: the software execution model and the system execution model. The software execution model is specified using Execution Graphs (EGs), which have nodes representing the components and arcs representing transitions. The system execution model is basically a queueing network model, which relies on workload parameters derived from EGs.

Following the SPE approach, Petriu and Wang have used UML activity diagrams for the software execution model, and UML collaboration diagrams for the system execution model [3]. The latter is modeled as a Layered Queueing Network (LQN) which differs from Queueing Network (QN) models in that servers can become clients of other servers in the model. An LQN model is represented as an acyclic graph whose nodes are software entities and hardware devices and whose arcs denote software requests. Menasc´e and Gomaa proposed a proprietary language called CLISSPE (Client/Server Software Performance Evaluation) [5] for the performance assessment of client server systems. The specifications in CLISSPE are fed to a compiler which generates the QN model which is then solved using a performance model solver. Some authors have also followed a Stochastic Petri Net based approach [4] for modeling the systems.

Our approach follows the basic SPE methodology with a focus on layered software architecture. We represent the software model in terms of a DTMC which is then transformed into a closed PFQN model. However, the QN model we propose to use also models software resource constraints such as limited number of threads at a particular machine. This allows for a model which is closely related to the real system and its constraints. As the architecture to be analyzed has been fixed to layered, specifying the software system becomes relatively simple, unlike in other approaches that follow some proprietary languages. This was also one of the aims we set forth while developing this approach. We also allow analysis of the software model on different hardware architectures, by letting the user specify the rating factor of the hardware under consideration, as compared to the one in which the resource demands of the components were specified for the specific architecture.

The inherent simplicity of specification makes our model feasible to be used in practice more effortlessly and even by those with relatively little knowledge of performance analysis techniques. Moreover the close modeling of real systems by our QN model, and the results we provide to the user, including the suggested bottleneck scale-ups and predicted performance improvements for the same, make our approach well suited to be used for large layered systems with real world constraints. We present the details of our approach in the following sections.

3 System Model and Assumptions

Our approach assumes that the computer system under consideration is an interactive system, wherein the system gives responses to the inputs given by the users. Further this interactive system follows the layered software architecture, with each layer interacting only with the adjacent layers. The user interacts with the first layer, which passes the request, if needed, to the second layer, which may pass it to the next layer, and so on. The last layer sends its response back, which then traverses through the layers, till the user gets the output. In general a layered software system could be visualized as in Figure 1.

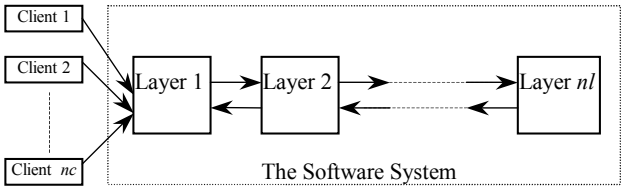

Fig. 1. The layered software architecture.

Note that different software components which constitute different layers, could be allocated to different (hardware) machines or some of them could be collocated on the same machine also. A layer acts as a functional unit providing some well defined services through appropriate interfaces and some computation and some I/O is done as the control passes through each software layer. The computation and I/O may be

intermixed and done repeatedly till the layer has completed its task. It then either returns the result, or passes the request to the next layer either on the same machine or on a different machine as the case may be. There might also be limitations on how many concurrent pending requests can exist for computation on a machine (which frequently is the case when the software server or the operating system of the hardware node, limits the number of threads or available connections.) We assume that a software component runs on only one machine at a time, i.e., there is no concurrency within a software component. However, several layers could be allocated to run on a single machine.

To analyze the performance of this architecture, we need to specify the average CPU time for a request in each layer and the average I/O time, which we call the Disk time in our model. Note that even though in a layer, the CPU work and I/O may be intermixed, for modeling purposes, they can be assumed to occur in an aggregate fashion, i.e., first all the CPU processing and then all the I/O processing. This combination does not affect the performance analysis and is based on the known insensitivity result of the product form solution of closed networks [1]. The allocation or deployment of the layers on different hardware nodes is also very important and is taken as a part of the specifications. The tool could be used to aid in choosing different allocations by comparing them.

In addition, we need to model the impact of connectors between the layers that are allocated on different machines. We capture this by the size of the request, the capacity of the connector, and the probability with which a request is sent by one layer to another layer. To model the load on the system, we require the range of the number of clients that the system might be subjected to. In addition, an estimate of the number of requests generated per unit time by each client is also needed. Overall, the following properties about the architecture are needed:

- The range of the number of clients [*ncmin, ncmax*] accessing the system and the average think time of each client *ttc*.
- The total number of layers in the software system *nl.*
- The relationship between the machines and the software components, i.e., which software layer is located to which machine and the number of machines. Thus corresponding to each machine *j* we have a set *L(j)* containing indices of the layers residing on *j* , $0<j\leq nm$ and *nm* is the number of machines.
- The number of CPUs and disks on each of these machines and thread limitations if any, or *nCPU(i), ndisk(i)* and *tlim(i).*
- The uplink and downlink capacities of the connectors connecting machines running adjacent layers of the system and the size of the packets going on these links or *capup(i), capdn(i), psizup(i)* and *psizdn(i)* where $0<i\leq nm$. Note that *capup(1)* and *capdn(1)* are the *total* uplink and downlink capacities respectively, of the connector(s) joining all the client to the machines.
- The service time required to service one request by a software layer given that it is using a standard CPU and a standard Disk for the purpose or *CPU(i)* and *disk(i)* where $0<i\leq nm$.
- Forward transition probabilities $p_{x(x+1)}$, i.e., the probability that a request being serviced by layer *x* would need the service of layer *(x + 1)* next.

- The rating factors *fcj* and *fdj* of the CPU and Disks respectively of each machine, which are present in the system, with respect to a standard CPU and Disk as considered above in 6.

4 Analyzing Performance

For analyzing the performance of a layered software system, we follow these main steps, which are discussed in detail in the ensuing sections:

- Constructing the DTMC model.
- Determining the queuing network model parameters.
- Modeling thread limitations.
- Queuing model solution and outputs.

4.1 Constructing the DTMC Model

We model the software system following layered architecture using a DTMC [13]. The state of the application at any time is given by the component or layer in execution at that time. Moreover, transitions between states represent transfer of control from one layer to the other. Assume that the DTMC to be analyzed has *k* states. Then the DTMC is characterized by a *k* by *k* transition probability matrix **P** = [p_{ij}]. All elements of a row in **P** add up to one and $0 \leq p_{ij} \leq 1$. We can calculate the expected total number of visits to a state j starting from state 1, V_j [1] by $V_j = \sum V_i p_{ij} + q_j$ where, q_j is the probability of starting in state *j*. Thus visit counts to a particular state could be obtained by solving a system of (n-1) linear equations. We can model a layered software system with *nl* layers as shown in the Figure 1 using a DTMC with *2nl*+2 states as shown in Figure 2.

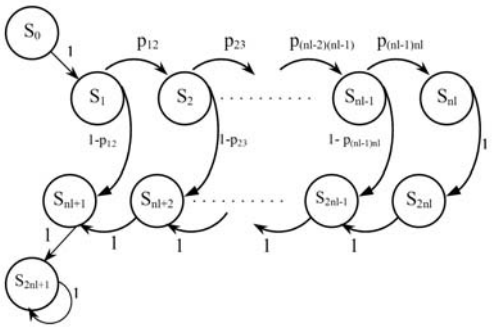

Fig. 2. The DTMC model for a layered software system.

The transition from state S_0 to S_1 represents a client sending request to the first layer with probability 1. The completion of a request's service by the layered system is denoted by a transition to state S_{2nl+1}, which is an absorbing state. Note that as the system is layered in nature, only transitions between adjacent layers are possible.

There is no incoming edge to S_0, which is the initial state, and no outgoing edge from S_{2nl+1}, which is the absorbing state. The states S_i and S_{nl+i} ($0 < i \leq nl$) represent control flow arriving to layer i in the forward and return paths of the request respectively. In the forward path, upon receiving service at layer i, the request can proceed further to the next layer with probability $p_{i(i+1)}$, or may return with the probability $(1-p_{i(i+1)})$. Also note that we have:

$$0 \leq p_{i(i+1)} \leq 1 \qquad 0 \leq i < nl$$

4.2 Determining Model Parameters

The layer to layer transition probabilities provided by the user and shown in Figure 2 can also be seen as a (2*nl*+2) by (2*nl*+2) transition probability matrix of a DTMC with (2nl+2) states given by:

$$P = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & \ldots\ldots & 0 & 0 & 0 & \ldots\ldots & 0 \\
0 & 0 & p_{12} & 0 & 0 & \ldots\ldots & 1-p_{12} & 0 & 0 & \ldots\ldots & 0 \\
0 & 0 & 0 & p_{23} & 0 & \ldots\ldots & 0 & 1-p_{23} & 0 & \ldots\ldots & 0 \\
\cdot & & & & & & & & & & \cdot \\
\cdot & & & & & & & & & & \cdot \\
0 & 0 & 0 & 0 & 0 & \ldots\ldots & 0 & 0 & 0 & \ldots\ldots & 1 \\
0 & 0 & 0 & 0 & 0 & \ldots\ldots & 1 & 0 & 0 & \ldots\ldots & 0 \\
0 & 0 & 0 & 0 & 0 & \ldots\ldots & 0 & 1 & 0 & \ldots\ldots & 0 \\
\cdot & & & & & & & & & & \cdot \\
\cdot & & & & & & & & & & \cdot \\
0 & 0 & 0 & 0 & 0 & \ldots\ldots & 0 & 0 & 0 & \ldots\ldots & 1
\end{bmatrix}$$

As mentioned in Section 4.1, we could calculate the visit counts to each of the states *V(i)*, by solving the set of linear equations given in the previous section. If we assume that between each pair of layers an imaginary connector is present, then assuming the connectors to be bi-directional, the number of visits to this connector in the forward and the backward direction would be same as each request going forward will eventually return. For any layer *i* in the system, the fraction of all the arriving requests that proceed to the next layer is the same as $p_{i(i+1)}$ in the DTMC as shown in Figure 2. Hence, the average number of visits to a uplink connector *i*, joining layer *i*, to layer *(i+1)*, is given by $V(i)p_{i(i+1)}$. The backward or return visits to the (downlink) connector will also be the same. Hence, we could calculate the total visits to these imaginary connectors joining the layers.

However, note that not all of these imaginary connectors are present in the actual system. The chosen allocation of software layers on different machines determines the connectors that are present between layers, which lie on adjacent machines. Only these are the connectors that physically exist and need to be considered in the queueing model. Hence, we can get the visit count to the uplink and downlink connectors respectively between machines j-1 and j as *Vconup(j)* and *Vcondn(j)*. Once the visit counts for the different layers and the connectors are calculated, the next step is to find the total service requirements on the actual CPUs and Disks on the machines as well as for the connectors that join these machines. For any machine *j (0<j≤ nm)* we have the total CPU and Disk service requirements given by

$$tmcpu(j) = f_{cj} \sum_{i \in L(j)} V(i) \times cpu(i)$$

$$tmcdisk(j) = f_{dj} \sum_{i \in L(j)} V(i) \times disk(i)$$

The above equation simply states that the total CPU or Disk service requirement at a particular machine is given by the sum of individual CPU or Disk service requirements of all the layers present on that machine, multiplied by the rating factor of that hardware device. For the connector *j* with uplink and downlink capacities *capup(j)* and *capdn(j)* respectively, we can compute the average delays caused due to each request as:

$$delayup(j) = psizeup(j) \div capup(j)$$

$$delaydn(j) = psizedn(j) \div capdn(j)$$

where, *psizeup(j)* and *psizedn(j)* are the uplink and downlink average packet sizes on the connectors. Each time the connector is visited, the above delays occur depending upon whether the request is going from a lower indexed machine to a higher indexed machine or vice versa. So the total average service requirement at connector j would be given by:

$$tconup\ (j) = Vconup(j)\ \cdot delayup\ (j)$$

$$tcondn\ (j) = Vcondn(j)\ \cdot delaydn\ (j)$$

The performance model that we generate is a closed product form queueing network wherein queueing stations represent the connectors, CPUs and Disks. This is shown in Figure 3. The clients are modeled by an Infinite Server (IS) [11] which allows a new request to be generated after an average of *ttc* seconds of the completion of the previous request. The total number of jobs in the closed PFQN is kept same as the number of clients.

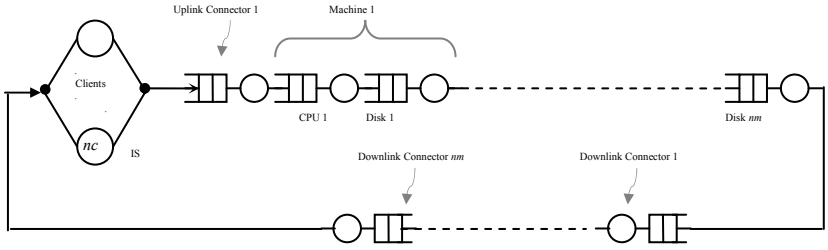

Fig. 3. The closed PFQN model of the system.

The connectors are modeled as FCFS stations [11] with rate as the reciprocal of the total average service requirements at that connector. The CPU for a machine is modeled by an FCFS station with CPU service rate for that machine. However, if there is more than one CPU present at a single machine, these are modeled as a multiple-server (MS) [11], with each server in MS having the rate as the CPU service rate for that machine. Thread limited systems are discussed in detail in the next section. The disk is modeled by an FCFS station with service rate same as the reciprocal of the total average disk service requirement at that machine. If there is more than one disk present at the machine, then all these disks are modeled as separate FCFS stations with equal probability of transition from the CPU to each of the disks. These are not

modeled as a multi-server (unlike the case of CPUs), as a request would not go to just any free disk, but is targeted to some specific data, on a particular disk.

4.3 Modeling Thread Limited Systems

In real systems there exist machines that have some software resources such as number of available threads in limited quantity. So there is an upper limit on the number of jobs that a particular machine can handle. We can visualize this as in Figure 4.

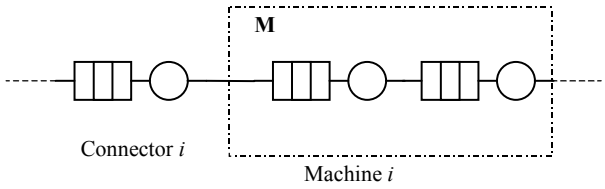

Fig. 4. Limited threads on a machine.

Here M is the upper limit on the number of jobs in that machine. We use a hierarchical combination of models so in the upper level model, the dotted box is represented a flow equivalent server. The lower level is modeled as a closed form PFQN. This is done by converting the subset of the model, which is shown inside the dotted region, to an equivalent closed PFQN. The rate of the flow equivalent server then equals the throughput E[T(n)] of this inner PFQN, which is directly proportional to the utilization of the same[1]. The flow equivalent server should continue to serve jobs only until limit M and till this point its rate keeps on increasing. However, after this point the rate no longer increases with increase in the number of jobs, and the jobs have to wait before getting into this inner PFQN or equivalently this thread limited server. So the rate of the flow equivalent server is given by:

$$flrate(n) = E[T(n)] \quad if\ n \le M$$
$$= E[T(M)] \quad if\ n > M$$

Such hierarchical models are easily specified and solved by SHARPE. Note that the flow equivalent server is represented as a load dependent server (LDS) in SHARPE.

4.4 Model Solution and Outputs

The performance model would consist of an upper level PFQN and if flow equivalent servers are present, some lower level (inner) PFQNs along with output statements. This has to be fed to SHARPE for getting the throughput of the whole system for the range of clients specified by the user. Other measures such as the average response times, saturation number and bottlenecks would be found using the throughput along with the specifications provided by the user.

In case of thread limited servers present in the system, each of those will have a separate PFQN representing the flow equivalent server, and the corresponding LDS

will be present in the top level PFQN. So in general the SHARPE input file will have the specifications of each of the inner PFQNs, if any, followed by the function that calculates their service rates, and the specification of the top level PFQN, followed by the output section. The model is fed to the SHARPE engine, which analyzes it and predicts the throughput of the whole system. The average throughput at each of the servers in the model for different number of clients is also calculated. This is required as unlike the FCFS or MS, the total service requirements per visit at the LDS change with varying number of clients in the system. This in turn affects the bottleneck analysis as is explained in the next section.

As part of the outputs, the approach provides for the throughput and the average response times of the whole system for the range of clients mentioned by the user. The approach also provides the saturation number of the system, which is the number of clients beyond which the system starts to saturate, i.e., the servers in the system start getting busy almost at all times and server utilization (the probability of finding a server busy) reaches unity. In practice the system should be running with the number of clients below the saturation number. Along with these it provides the bottleneck analysis as explained in section 5.

5 Bottleneck Analysis

One important part of our analysis is bottleneck analysis of the whole system. The bottleneck node of the system is defined as the one at which the total service requirement is the largest or the relative utilization (the probability of finding a server busy) is the highest. Hence in a closed PFQN, bottleneck nodes can be found out even before solving the queueing network model, by comparing the total service demands on the various nodes. However, if there is a thread limited server in the network, the total service requirements at this node is dependent upon the instantaneous number of jobs on that server (hence the name load dependent servers). Thus in such networks, we need the throughput values for the lower level closed PFQN for calculating the total service requirements at that node for varying number of clients from the SHARPE engine.

Bottleneck nodes are the ones, which are most busy (or have high relative utilizations), and most jobs will tend to queue up at these servers. They will cease to be the bottleneck if they are scaled up so that they no longer have the highest total service requirements (or the highest relative utilization). We provide information about the first as well as the second bottleneck(s) in the system. In general one can determine the minimum scaling up factor *Scale(k)* of the k^{th} bottleneck by the formula:

$$Scale(k) = \frac{\text{Avg. service requirements per job of } bottleneck(k)}{\text{Avg. service requirements per job of } bottleneck(k\text{-}1)}$$

One thing to be noted is that the bottleneck(s) of the system might change with the number of clients in the system. This is so, because the average service demands and hence the average service times per job of the thread limited servers are load dependent and would change with the number of clients in the system. Hence, we give the bottlenecks for the whole range of the clients as mentioned by the user.

Bottleneck information is then used to analyze the effect of scaling the bottleneck server up, on the average throughput and average response time of the whole system. This is done by iterating once again through the performance model generation phase with the bottleneck node's service rate scaled up by the suggested value, and then feeding the model to SHARPE and analyzing throughput again. Thus one can get the percentage change in system performance for the suggested scale up, and decide if the scale up is worthwhile for the system. Bottleneck analysis gives the user a good idea of the amount of improvement in system performance if he/she chooses to put in effort in scaling up / improving the bottleneck of the system.

6 An Example

We have implemented our approach as a web based tool, which automates our approach. The implementation of the tool was partially done as an undergraduate project [14], and was extended later on. The tool completely automates the task of performance analysis of layered software systems. This web tool is based on cgi scripting and renders html forms for the user to fill in the specifications of the layered system under consideration. The tool then uses the specifications and constructs a DTMC internally, representing the software system, calculates the service requirements and generates a SHARPE input file, which has the system model as a closed PFQN. This is then fed to SHARPE, which works as our model solver and backend. The results from SHARPE are further analyzed, for bottlenecks. The scale up for the primary bottleneck in the system is found and the QN model is reconstructed and again fed to SHARPE to get the new performance attribute values.

Consider an example of a 4-layered software architecture. Suppose that the current plan is to have these 4 layers run on 3 machines, with layer 1 running on machine 1, layer 2 on machine 2, and layers 3 and 4 on machine 3. Further assume that there is a limit of 25 threads on machine 2. In the baseline hardware configuration that we are planning for be that each machine has 2 CPUs and machines 1 and 2 have 1 Disk each, while machine 3 has 4 disks available. The data about the software architecture and the hardware that is being planned is summarized in Table 1.

Table 1. Example software and hardware specifications.

Number of Layers	4	Number of Machines	3
Layer 1 runs on Mc	1	Layer 2 runs on Mc	2
Layer 3 runs on Mc	3	Layer 3 runs on Mc	3
No. of CPUs on Mc 1	2	Thread Limit on Mc 1	No
No. of CPUs on Mc 2	2	Thread Limit on Mc 2	25
No. of CPUs on Mc 3	2	Thread Limit on Mc 3	No
No. of Disks on Mc 1	1	No. of Disks on Mc 2	1
No. of Disks on Mc 3	4		
Total capacity of connector joining Clients and Mc 1 Uplink/Downlink	56/512 Kbps	Capacity of connector joining Mc 1 and Mc 2 Uplink/Downlink	1/1 Mbps
Capacity of connector joining Mc 2 and Mc 1 Uplink/Downlink	1/1 Mbps	Capacity of connector joining Mc 2 and Mc 3 Uplink/Downlink	1/1 Mbps

Now we have to provide the system execution behavior estimates. We have taken the parameters resembling those, which might characterize a distributed transaction processing system following a 4 layered architecture. We take an example of an ADSL connection between the clients and the system as is evident from the asymmetry in the uplink and downlink capacities of the link joining clients to machine 1. Further, assume that the client request does much of its CPU processing in layers 2 and 3 and does most of its I/O operations in layer 4. Also, assume that 60% of all requests coming to layer 1 need to go to higher layers for service and similarly 90 % and 100 % of the requests which reach layer 2 and 3 respectively require service from higher layers. The values of the relevant parameters are as in Table 2.

Table 2. Example system execution behavior estimates.

Data packet sizes: (in bytes)			
From Client to Mc 1	250	from Mc 1 back to Clients	2000
From Mc 1 to Mc 2	250	From Mc 2 back to Mc 1	1000
From Mc 2 to Mc 3	250	From Mc 3 back to Mc 2	1000
Times per visit: (in secs)			
CPU time/visit of Layer 1	0.01	CPU time/visit of Layer 2	0.03
CPU time/visit of Layer 3	0.06	CPU time/visit of Layer 4	0.01
Disk time/visit of Layer 1	0.02	Disk time/visit of Layer 2	0.02
Disk time/visit of Layer 3	0.002	Disk time/visit of Layer 4	0.20
Probability of request flow from:			
Layer 1 to higher Layers	0.60	Layer 2 to higher Layers	0.90
Layer 3 to higher Layers	1.00		

Suppose we specify to the tool that we want to estimate the performance of this system for 1 to 75 clients, each client having an average think time of 1 sec and the rating factor for all the devices as unity. The tool does the analysis and gives the output as in Figure 5 and 6.

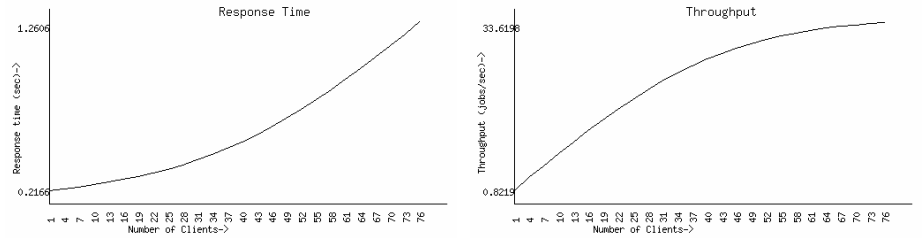

Fig. 5. Analysis output: The average response time and throughput graphs.

From Figure 5, we can see that the model suggests that the average response time is quite low initially, but then after around 8 clients the average response times starts to increase very fast. Similarly if we study the throughput graph, this shows the throughput constantly increasing with the number of clients initially, but then almost becoming constant if the number of clients is increased further, the maximum throughput of the system being about 34 jobs/sec. These two observations are due to the same phenomenon of the onset of saturation of the system - in this case occurring around 11 clients in the system. Below 11 clients, there are practically no queues at

each of the machines, and so the average response time is fairly low, and the throughput of the system, increases almost linearly as new clients are added. But as the number of clients increase further, queues start to build up at different service centers, and jobs have to wait for other jobs to complete service, before they could be taken up. The servers in these conditions are busy almost all the time and server utilizations (the probability of finding the server busy) reaches unity. Thus congestion builds up in the system, and hence the average response time of the system keeps on increasing thereon as more and more clients are added. Because of the same reason, the throughput of the system reaches a limiting level, and does not increase after that (as most of the servers are already busy processing to their limit) and so we get the flat region in the throughput graph.

The graphs are very useful for a system architect as they show precisely how many clients would the system be able to handle efficiently. Practically a system should never be running in a saturated condition as then the system performance degrades very fast. These graphs could be used to ascertain the kind of the average response times and throughput the system would deliver for the specified range of clients and whether that meets the desired performance criteria or not. The system architect could also get an idea about the performance of the system, in conditions of excessive loads.

As shown in the bottleneck analysis in Figure 6, the tool predicts the primary and secondary bottlenecks in the system along with the minimum scale-up needed so that they no longer are the bottlenecks. As mentioned earlier, as the bottlenecks in the system may change upon changing the number of clients in the system, hence the tool provides the bottleneck analysis for the whole range of clients. In Figure 6, the same is shown for 43 clients. Moreover the tool iterates upon the analysis once more with the primary bottleneck scaled-up, and then shows the improvements in throughput, average response time, and saturation number of the system.

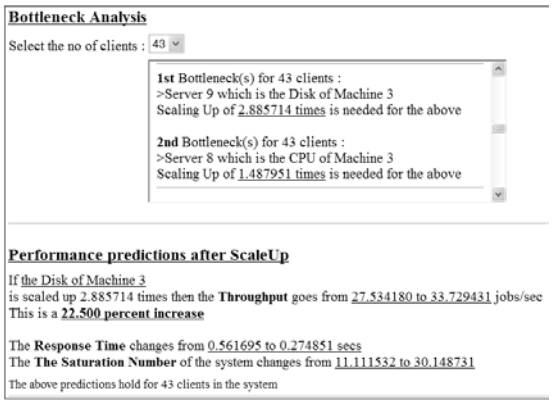

Fig. 6. The bottleneck analysis output.

In this example system, we can see the improvements - by scaling the disk of machine 3, we could get an improvement of more that 22.5% in the throughput of the system and about a 50% decrease in average response time. Figure 7 illustrates the comparative improvements using graphs, showing the system throughout and the

average response time variation with load, before and after the suggested scale-up. We can see that if the suggested scale-up is done, then the system can sustain more number of clients, without degradation as compared to the initial configuration.

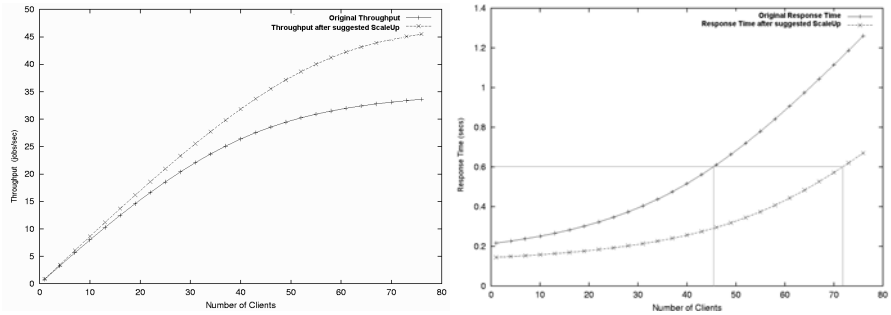

Fig. 7. The effect of system scaleup as suggested by our tool on average system throughput and response times.

From the capacity planning point of view, say if an average response time of upto 0.6 seconds is deemed acceptable for the system, one can observe from the average response time variation graph in Figure 7 that the capacity of the system increases from 46 to 72, if the suggested scaleup is done. One can also use to the tool to see the possible effects of any changes in the system hardware or software. We used the tool to evaluate the above example system but with the number of concurrent threads in Machine 3 limited to 25. The tool shows that this restriction causes the maximum throughput of the system to go down to 20 jobs/sec from 34 jobs/sec. In addition, the tool could be used for comparing the effect of different deployments of the layers on the available machines, on the overall system performance. One such comparison for the example system is shown in Table 3 which shows that the first deployment scheme is significantly better than the others in terms of maximum average throughput.

Table 3. Effect of different layer deployments on maximum average throughput.

Layer(s) deployed on			Maximum average throughput (jobs/s)
Machine 1	Machine 2	Machine 3	
1st	2nd	3rd ,4th	33.62
1st	2nd ,3rd	4th	19.84
1st ,2nd	3rd	4th	29.81

7 Discussion and Conclusion

In this paper we presented an approach of performance evaluation of systems following layered architecture. The approach deals with first constructing a DTMC model of the software system, using the specifications user has provided. This model is then solved to get the total visit counts to different layers of the system and calculate total service requirements of the system on the hardware over which the software system is deployed. These are then used to construct a closed PFQN model for the system. This

model also takes care of limited software resources as threads on a particular machine. The PFQN model is solved using SHARPE as the backend, the outputs of which are analyzed, and various performance metrics such as throughput and the average response times, and saturation number are provided. Moreover bottleneck analysis is done and the minimum scale up for the bottleneck node in the system is suggested. The approach also allows for a prediction of the improvement in system performance if the scale up is actually done. There are two major applications of this approach: First is in architecting and deploying new layered systems and the second is in tweaking or upgrading or scaling up existing systems.

As COTS based development is becoming very popular these days, it is commonplace today to use components as black boxes for the desired functionality. In such cases the designer could use our approach and estimate the performance characteristics of the final system he wishes to build using those components. Moreover it would also allow the architect to know about the possible bottlenecks in the system and how much scaling up is needed for those components.

The second application as mentioned before is when an existing system has to be scaled up or some additional software or hardware component has to be added. The system administrator should have some idea of the change in the system performance due to the change in system hardware and/or software configuration. Our approach could be used to ascertain that. The system administrator need not have an in-depth knowledge of performance evaluation techniques for this and our tool could be employed for the same by providing some specifications, which are easy to get.

At the software architecture phase, the actual components, which would constitute the layers, would not be present (unless COTS approach is used) so, the service requirements of different components at different devices have to be estimated from previous experience with similar software components. If off the shelf components are being used, the CPU and I/O times taken by particular layers to execute once could be assessed by using built in tools provided by various operating systems like *iostat* or *sar*. Tests will have to be conducted for each layered component separately as the above mentioned tools do not provide application level break-up of the measurements. For testing purposes if a single layer is run on a single machine one can get the total CPU and I/O times for say n executions and then get the average CPU and I/O times per execution.

Each proposed component in a layer could be examined to find the components with which it interacts with a non-zero probability and known operational profiles of similar systems might be used to estimate the associated transition probabilities between the layers. For existing systems, techniques mentioned in [6] might be used to ascertain the transition probabilities. Specifications such as the capacity of connectors, the number of CPUs and disks or thread limitations are all system characteristics. Some measurements will be needed for the packet sizes on the connectors joining various machines and could be ascertained by using a suitable network analysis utility.

There are still lots of avenues in this approach for future work. Our approach is limited only to layered systems at present. We believe that this could be extended to general software architectural patterns also. However, the aim while doing so would be to keep the specifications needed as simple and practical as possible so that the approach is easily adoptable in practice. One other major extension could be to mod-

ify the approach so that it allows for optimizing the use of various system resources to provide the maximum possible performance. This would be beneficial to system architects as well as system administrators and will allow them to minimize investment and maximize performance of their systems. One of the many aims of this approach is to provide a thorough performance evaluation of the layered software system under concern. Moreover the approach is helpful both at the time of architecting new systems as well as scaling up or improving existing systems. The tool that we have built implements our approach and is very simple to use. We hope that our approach would help software engineers and performance experts to architect better layered systems, as well as allow those who are not specialists in this field to perform performance evaluation of their systems.

References

1. K. S. Trivedi, "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", *John Wiley and Sons*, 2001.
2. C.U. Smith, "Performance Engineering of Software Systems", *Addison Wesley*, 1990
3. Dorina C. Petriu, X. Wang, "From UML descriptions of High-Level Software Architectures to LQN Performance Models", *Proceedings of AGTIVE'99*, Springer Verlag LNCS 1779, 1999.
4. P. King, R. Pooley, "Derivation of Petri Net Performance Models from UML Specifications of Communication Software", *Proceedings of XV UK Performance Engineering Workshop*, 1999.
5. D. A. Menasc´e, H. Gomaa, "A Method for design and Performance Modeling of Client/Server Systems", *IEEE Transactions on Software Engineering*, Vol. 26, No. 11, pp. 1066–1085, 2000.
6. K. Goˇseva–Popstojanova and K. Trivedi, "Architecture–based approach to reliability assessment of software systems", *Performance Evaluation*, 45:179-204, 2001.
7. L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice". *SEI Series in Software Engineering*, Addison-Wesley, 1998.
8. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "Pattern-Oriented Software Architecture, Volume 1: A System Of Patterns" , *John Wiley and Sons*, 2000.
9. M. Shaw, D. Garlan, "Software Architecture, Perspectives On An Emerging Discipline", *Prentice-Hall Inc.*, 1996.
10. C. U. Smith, L.G. Williams, "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives", *IEEE Transactions On Software Engineering, Vol 19, No7, Pages 720-741*, 1993.
11. R.A. Sahner, K.S. Trivedi, and A. Puliafito, "Performance and Reliability Analysis of Computer Systems: An Example-Based approach Using the SHARPE Software Package", *Kluwer Academic Publishers*, 1996.
12. Dorin Petriu, Murray Woodside, "Software Performance Models from System Scenarios in Use Case Maps", *Proc. Performance TOOLS 2002, London,* 2002.
13. Swapna S. Gokhale, W. Eric Wong. K. S. Trivedi, and J.R. Horgan, "An Analytical Approach to Architecture-Based Software Reliability Prediction", *IEEE Int. Comp. Perf. and Dependability Symposium*, Durham, NC, USA, Sept. 1998.
14. M. Vikram, P. Kant, "Evaluation of Layered Architecture Software Systems for Performance Attributes using Closed Product Form Queuing networks", *B.Tech Project Report, CSE, Indian Institute of Technology Kanpur,* India, 2003.