RC23929 (W0604-027) April 6, 2006
Computer Science

IBM Research Report

Controlling Quality of Service in Multi-Tier Web Applications

Yixin Diao, Joseph L. Hellerstein, Sujay Parekh,
Hidayatullah Shaikh, Maheswaran Surendra
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Controlling Quality of Service in Multi-Tier Web Applications

Yixin Diao, Joseph L. Hellerstein, Sujay Parekh, Hidayatullah Shaikh, and Maheswaran Surendra
IBM Thomas J. Watson Research Center, Hawthorne, New York 10532, USA
{diao, hellers, sujay, hshaikh, suren}@us.ibm.com

Abstract

The need for service differentiation in Internet services has motivated interest in controlling multi-tier web applications. This paper describes a tier-to-tier (T2T) management architecture that supports decentralized actuator management in multi-tier systems, and a testbed implementation of this architecture using commercial software products. Based on testbed experiments and analytic models, we gain insight into the value of coordinated exploitation of actuators on multiple tiers, especially considerations for control efficiency and control granularity. For control efficiency, we show that more effective utilization of tiers can be achieved by using actuators on the bottleneck tier rather than only using actuators on the entry tier. For granularity of control (the ability to achieve a wide range of service level objectives) we show that a fine granularity of control can be achieved through a coordinated, cross-tier exploitation of coarse grained actuators (e.g., multiprogramming level), an approach that can greatly reduce controller-induced variability.

1 Introduction

Most Internet service sites have a multi-tier architecture that partitions the processing of web requests into stages for HTTP servers, application servers, and database servers. Recently, there has been interest in providing differentiated quality of service (QoS) for web requests as part of fee-for-service business models and other considerations. This paper describes a tier-to-tier management architecture that provides a transparent way to deploy and control QoS actuators in multiple tiers. We show the value of having actuators in multiple tiers in terms of resource efficiency and the ability to achieve service level objectives.

For a multi-tier system, each tier consists of one or more nodes (hardware entities) that are dedicated to a specific kind of processing. The first Edge Server tier provides load balancing and request routing; the second HTTP Server tier does HTTP parsing and response generation; the third Application Server tier contains application servers typically providing a J2EE environment for business logic; the fourth Database Server tier contains database server nodes that manage persistent data. Sometimes, there is a fifth tier as well if a separate storage system is used such as a storage area network. Client requests enter the first tier and are routed to an HTTP Server; some fractions of the HTTP requests also require processing by Application Servers. A fraction of the requests processed by Application Servers also require services from a Database Server. Because the inter-tier interaction is synchronous, threads/processes in upstream tiers are blocked while waiting for the completion of processing in downstream tiers. Thus, requests may simultaneously consume resources in the HTTP, Application, and Database server nodes.

The QoS of requests is managed by specifying performance targets or **service level objectives (SLOs)** such as desired response times that the Internet service system must enforce. Typically, SLOs are specified for service classes. For example, web requests to list the best selling books, recently announced children's toys, and the top renting action movies can all be classified as browse requests. Achieving SLOs requires one or more control mechanisms or actuators that provide a means to give preferential treatment to designated classes of requests. Examples of actuators include CPU priorities, JVM memory allocations, multiprogramming levels, server assignments, and request routing.

In such multi-tier systems, most proposals call for managing these actuators centrally in order to provide proper coordination [1, 2, 3, 4]. However, centralized management can impose significant coordination overheads due to messages exchanged, and there is concern about controllability (and stability) if there are communication delays [5]. In addition, because upstream threads wait for replies from downstream tiers, it is feasible to control the system QoS using actuators only at the entry tier [6] since it changes the effective arrival rate for all the downstream tiers. This approach is attractive since it can be implemented non-invasively (using a proxy as in [6]), and independent of the internal architecture of multi-tier systems. Our studies show that compared to using actuators at all the tiers, doing only

front-end control is a poor choice since it can significantly reduce performance and impair the ability to achieve service level objectives.

This paper describes a management architecture that supports distributed management of actuators in multi-tier systems. Our **tier-to-tier (T2T) management architecture** is structured so that each tier only communicates with its upstream and downstream neighbors. Such an approach avoids the congestion associated with centralized management and improves scalability. We implement the T2T management architecture in a testbed consisting of application and database tiers using the IBM WebSphere Application Server and DB2 Universal Database Server, and we demonstrate the value of using actuators in multiple tiers through the testbed experiments. Specifically, we gain insight into control efficiency and control granularity from analytic models calibrated from testbed data. For control efficiency, we show that more effective utilization of tiers can be achieved by using actuators on the bottleneck tier rather than only using actuators on the entry tier. These efficiencies result from avoiding queueing delays imposed on requests at non-bottleneck tiers. To the best of our knowledge, none of such analysis has been conducted in the existing literature, especially in a multi-tier system. Granularity of control is the ability to achieve a wide range of SLOs, something that can be difficult with coarse actuators such as multiprogramming level (a commonly used technique because of the ease with which it can be manipulated). Our results show that a fine granularity of control can be achieved through a coordinated, cross-tier exploitation of coarse grained actuators.

The remainder of this paper is organized as follows. Section 2 describes the tier-to-tier management architecture and overviews the testbed setup and experiments. Section 3 studies efficiency considerations in actuator placement. Section 4 studies the effect of control granularity. The conclusions are contained in Section 5.

2 Tier-to-Tier Management Architecture

This section describes the T2T management architecture that provides a decentralized approach to achieving SLOs in multi-tier systems. Figure 1 displays the components of this architecture. There is an InterTier Manager that manages QoS interactions between tiers. This includes: (a) collecting measurements from downstream nodes and forwarding them to upstream nodes, (b) accepting SLOs for downstream nodes from upstream nodes and resolving conflicts, and (c) communicating reconciled goals to downstream nodes. The Upstream InterTier Manager and Downstream InterTier Manager in the figure are both instances of InterTier Managers with this function.

Each node in the T2T management architecture is structured as indicated in the blow-up in tier m in Figure 1. There are three layers: tier function (e.g., HTTP processing, database processing), T2T QoS management, and node services (e.g., QoS actuators for adjusting process CPU priorities). There are four interacting components in the T2T QoS management. The Upstream Proxy provides the interface between T2T components in a node and its upstream InterTier Manager. The abstract resource manager supports the abstract resource management interface provided to the Autonomic Controller. This interface includes APIs for: adding/removing a service class to manage, setting the resource share for the service class, and querying the performance metrics for the service class. The Autonomic Controller (a) accepts SLOs for a service class and (b) uses performance measurements of the node and downstream delays to control local resources and specify SLOs for downstream tiers. The Downstream Proxy: (a) provides the interface between T2T components in a node and its downstream InterTier Manager and (b) provides measurements of downstream delays.

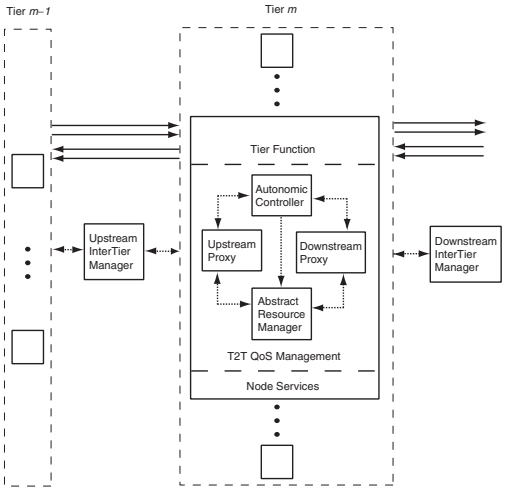


Figure 1. *Tier-to-tier architecture for a multi-tier system. T2T provides coordinated control of actuators in multiple tiers in which each tier only knows its upstream and downstream neighbors so as to avoid centralized knowledge of actuators.*

Normal operation of the T2T QoS management layer is as follows. The Upstream InterTier Manager receives an SLO from tier $m-1$ and forwards this to the appropriate nodes in tier m, where the Upstream Proxy provides the SLO to the Autonomic Controller. The SLO contains information about performance metrics available on that node and target values (e.g., response times for request entering the node should be less than 2 seconds). This information is used to subscribe to metric values from the Abstract Resource Manager. The Autonomic Controller also subscribes

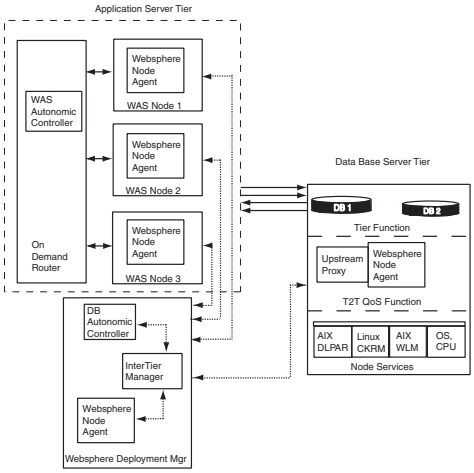


Figure 2. *Instantiation of the T2T architecture in Figure 1 for a two tier system.*

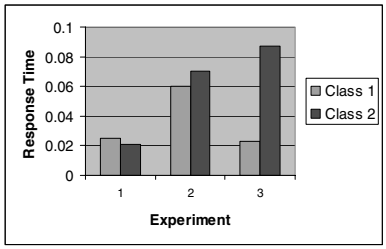


Figure 3. *Testbed measurements illustrating the ability of the T2T Testbed to achieve SLOs.*

to metrics provided by the Downstream Proxy regarding the contribution of downstream tiers to delays. Based on these data, the Autonomic Controller constructs an SLO for the downstream tier. This SLO is communicated to the Downstream Proxy, which in turn provides it to the Downstream InterTier Manager.

The Autonomic Controller monitors the performance metrics indicated in the SLO. If the Autonomic Controller determines that the SLO metric is too high or too low, the Autonomic Controller determines whether the problem is local to its node or is due to downstream tiers. For the former, the Autonomic Controller invokes the Abstract Resource Manager to adjust local resource allocations (e.g., CPU priorities). If the latter is the case, the Autonomic Controller specifies a new SLO for the downstream tier.

In practice, the architecture in Figure 1 must be packaged in an existing system. To understand the issues in doing this, we built a two tier testbed consisting of an Application Server Tier and a Database Server tier. As depicted in Figure 2, IBM's Websphere Application Server (WAS) is used for the Application Server and installed on four nodes, and the IBM DB2 Universal Database Server is employed for the Database Server Tier and installed on one node. The Abstract Resource Manager function is instantiated by a Websphere Node Agent. The WAS Autonomic Controller is packaged in the On Demand Router (ODR), a component that assigns incoming requests to WAS nodes. Since there are only two tiers, we packaged the InterTier Manager with the DB Autonomic Controller and incorporated them into the Websphere Deployment Manager, an existing administrative component that, among other things, provides configuration information. This structure had the additional advantage of minimizing impact on the Database server. Note that the Websphere Node Agents in the WAS nodes communicate with the Deployment Manager to provide measurement data. The DB tier Node Services adjust the CPU shares through different OS level mechanism such as dynamic logical partitions (DLPAR) and class-based kernel resource management (CKRM).

We demonstrate the ability of the T2T management architecture to provide effective control by conducting experiments on the testbed in Figure 2 using the industry standard benchmark *Trade* [7] (a database intensive workload). There are two service classes in our experiments. Class 1 has a response time SLO of 25 msec; class 2 is a best effort class without specific SLO. In the first experiment depicted in Figure 3, load is light, and class 1 achieves its SLO. In the second experiment, load is heavy, and we use the same actuator settings as in the first experiment. We see that class 1 has a response time of 60 msec, which is well above its SLO. In the third experiment, actuator setting have been adjusted, and we see that class 1 again has a response time of 25 msec. Note that the response time of class 2 has increased substantially from experiment 2 to experiment 3, which is not surprising since preferential treatment is given to class 1 so that its SLO can be achieved.

3 Control Efficiency

Besides the management architecture, placement of control actuators also affects the quality of controlling multi-tier web applications. In this section we study efficient control actuator placement. We start from providing a brief description and motivation of the problem. Afterwards, we investigate the control scheme based on an analytical model calibrated from testbed data, and show that we get better performance with actuators in multiple tiers.

Internet service requests for a multi-tier system can receive service from multiple tiers. Although it may be sufficient to only have control actuators at the bottleneck tier, it is preferred to have actuators at all tiers as the bottleneck tier can vary for different workloads. However, such a scheme can increase the implementation cost since not all

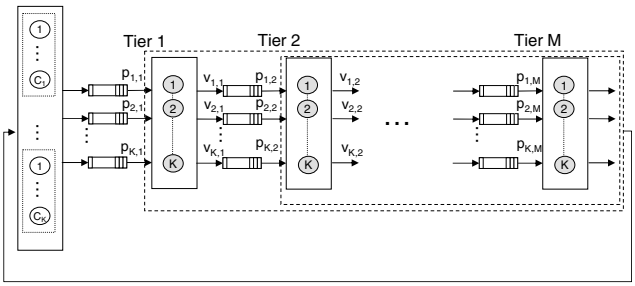


Figure 4. *Architecture of a closed queueing model for multi-tier web applications with multiple service classes.*

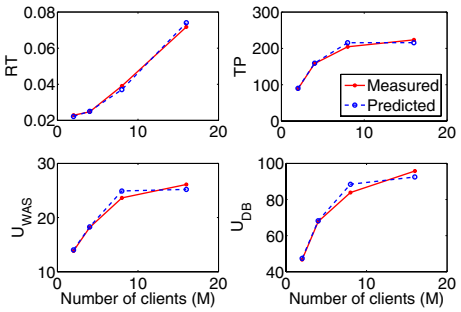


Figure 5. *Modeling results for a multi-tier system.*

tiers have the actuators already. An alternative is to have actuators only at the entry tier. The rationale comes from the cross-tier dependency of a multi-tier system: since the threads/processes in upstream tiers are blocked while waiting for the completion of processing in downstream tiers, differentiated service can be achieved through controlling the threads/processes at the entry tier which changes the effective arrival rate for all the downstream tiers including the bottleneck tier. This alternative seems appealing since it can reduce the actuator implementation cost and make this control approach general.

3.1 Modeling Multi-Tier Systems

In the following, we investigate different control strategies based on an analytical model. Using a model-based approach facilitates the generality of the results, and allows us to explore more system/workload variability and to avoid corruption from measurement noise.

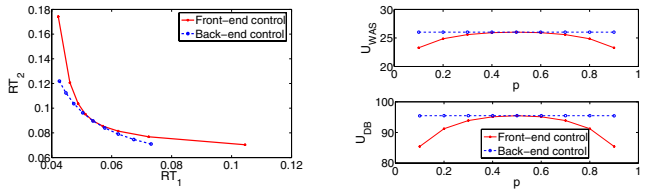
Figure 4 illustrates the architecture of the closed queueing network model for M tiers of servers with K service classes. The behaviors of workload and concurrent sessions are modeled using a machine repair model [8], which is a closed queueing network consisting of two service centers. As indicated in Figure 4 by the solid box on the left, the service center 1 contains as many servers as concurrent clients–the served concurrent clients and their corresponding servers are marked by $1, 2, \ldots, C_k$ for service class k and grouped by a dotted box. The service center 2, as indicated by the outer dashed box on the right, includes multi-tier servers to service client requests. A request sent by a client in service center 1 enters the queue in front of the service center 2, and is serviced by servers in multiple tiers after admission. The completed request comes back to the client in service center 1, and this client will issue a new request after a think time Z_k for class k. (We have also modeled the system as an open queueing model for the situations where the number of clients and think time are difficult to obtain. In the interest of brevity, however, we do not discuss the open queueing model in this paper.) The QoS control actuators are represented by $p_{k,m}$ for class k on tier m. For Application Server tier, the QoS actuator can be the number of threads. For Database Server tier, the QoS actuator can be the CPU shares.

Due to cross-tier dependency, a server (and its server threads) can be in one of the following states: 1) using CPU or disk of this tier, 2) waiting for CPU or disk of this tier, 3) waiting for response from a lower level tier, 4) idle after completing the request service. In a multi-tier system a service request can only be completed after it receives services from all related tiers; this means the service rate of any tier not only is affected by its own tier service capacity (and resource contention) but also depends on the service rate of successive tiers, so that a tandem queue model is inappropriate (which fails to represent such dependency). The cross-tier dependency is modeled through the layered queueing model and solved iteratively [1, 9]. In addition, we use a scaling factor $v_{k,m}$ to model the behavior that the requests processed at one tier may generate multiple requests for the lower tier; the value of $v_{k,m}$ can be obtained through monitoring the throughput of each tier.

We calibrate the model from the experimental data as shown in Figure 5. It indicates a close fit between the measured data (marked by the asterisks and solid lines) and the predicted values (marked by the circles and dashed lines) of response time (in the unit of seconds), throughput, WAS utilization (in the unit of percentage), and DB utilization (in the unit of percentage) for different numbers of clients. Although only a small number of clients are present in the system, their think time is set to close to zero. This results in a large range of workload intensity (from 45% to 95% utilization for the database tier). More details on the modeling algorithm and validation results can be found in [10].

3.2 Selecting Efficient Control Actuators

This section studies the effect of control actuator placement based on the calibrated model built above. While different workload scenarios have been considered, below we



(a) Response time *(b) Utilization*

Figure 6. *Performance comparison of different control actuator placement when the Database tier is the bottleneck.*

illustrate using two service classes with different numbers of clients $C_1 = 6, C_2 = 10$; the service rate for the WAS tier is $x_{1,1} = x_{2,1} = 856$ and the service rate for the DB tier is $x_{1,2} = x_{2,2} = 373$. Clearly, the Database Server tier is the bottleneck tier. Figure 3.2(a) compares the response time (in the unit of seconds) from different control actuator placement. The horizonal axis is the response time for class 1, and the vertical axis is the response time for class 2. The solid line indicates the effect from the front-end control, which uses WAS tier actuators to control the fraction of threads dedicated to class 1 ($p_{1,1}$) versus class 2 ($p_{2,1}$); the asterisks indicate nine $p_{1,1}$ values from 0.1 to 0.9 (while $p_{2,1}$ is computed as $1 - p_{1,1}$. No service differentiation is provided at the back-end Database Server tier, that is, we represent the DB server with a single server and a first-come-first-served single queue. Note that this is not a symmetric curve because two classes have different numbers of clients. Next, we consider the back-end control scheme as represented by the dashed line in Figure 3.2(a), where no service differentiation is provided in the front-end WAS server and the DB tier actuators adjust the fraction of CPU shares allocated for each class ($p_{1,2}$ and $p_{2,2}$). Similarly, the circles indicate nine $p_{1,2}$ values from 0.1 to 0.9 and $p_{2,2} = 1 - p_{1,2}$. Note that no request will be dropped even if the control occurs at the back-end tier; thus, no consumed resource will be wasted.

As shown in Figure 3.2(a), although front-end control at the entry tier can still differentiate the service, it is not as effective as the back-end control executed at the bottleneck Database Server tier. Given any response time value for class 1 (or class 2), the back-end control can always achieve an equal or smaller response time for the other class than that can be achieved by the front-end control. The efficiency of the back-end control can be further illustrated from Figure 3.2(b), which compares the utilizations (in the unit of percentage) of the WAS server and DB server from the same model-based study. The back-end control at the bottleneck tier shows more effective utilizations not only at the bottleneck tier, but also at the entry tier.

These efficiencies of bottleneck tier control result from avoiding queueing delays imposed on requests at non-bottleneck tiers. Due to request synchronization between multiple tiers, the thread in the WAS server cannot process a new request when it is waiting for request completion in the DB tier. When the DB tier is the bottleneck tier, using front-end control causes the requests from the low priority class to queue prior to the WAS tier; this is conducted through limiting the allocation of WAS tier threads. This reduces the effective arrival rate to the DB tier so that more DB tier resources can be utilized by the high priority class. However, when DB tier resources become available, they cannot be utilized immediately to service the low priority class. Even if requests can be dequeued immediately at the front tier, they can only be serviced at the DB tier after they complete service at the Application Server tier first. The WAS tier service time introduces a delay, which lowers the utilization of the database server.

The effect of the delay can be further illustrated through an M/M/1 model. As shown in Figure 7(a), if the control is executed in the front-end to block the requests and build the queue, a delay (z) will occur after the request departs from the queue and before it is serviced by the back-end server. This delay comes from the service time at the front end and the dependency between layers. Since this delay occurs prior to the back end server, we call it back-end delay and its effect on the response time T_b can be expressed as

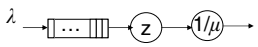
$$T_b = \frac{\frac{1}{\mu} + z}{1 - \lambda\left(\frac{1}{\mu} + z\right)} \tag{1}$$

where λ is the arrival rate, μ is the service rate, and z is the delay. In contrast, as shown in Figure 7(b), if the control is executed in the back-end (i.e., achieving service differentiation by manipulating the DB tier resources), the requests will be blocked and the queue be built just before the back-end server, and the front-end service delay (WAS tier service time) will occur before the request enters the queue. We call it front-end delay. Its effect on the response time T_f can be expressed as

$$T_f = \frac{\frac{1}{\mu}}{1 - \frac{\lambda}{\mu}} + z \tag{2}$$

Since $T_b > T_f$, back-end control is a more effective approach than front-end control when back-end is the bottleneck tier.

As shown in Figure 3.2, this effect is more apparent when a longer queue is formed as in the two ends of the curve; the back-end control can still maintain the same high utilization of the server, while the front-end control results in lower server utilization due to the effect of delay, which is exaggerated when the queue is longer. The front-end control behaves the same as the back-end control in the middle of the curve where there is no service differentiation. Also note that the front-end control affects the utilization of both tiers because an ineffective control brings down the throughput.



(a) Front-end control (back-end delay)

(b) Back-end control (front-end delay)

Figure 7. *Different control and delay positions as illustrated in an M/M/1 model, where λ is the arrival rate, μ is the service rate, and z is the delay.*

Table 1. *Assessment of the Markov Model. Entries are in the form measured/predicted.*

C	RT	TP	U_{WAS}	U_{DB}
4	0.024/0.14	158/226	18/19	68/97
8	0.039/0.030	204/233	24/20	84/99
16	0.071/0.060	223/233	26/20	96/100

4 Control Granularity

Section 3 discusses control efficiency using actuators with continuous settings such as CPU shares. Unfortunately, many common actuators have discrete settings. An excellent example of this is multiprogramming level (MPL), which is one of the most widely used actuators. This section addresses the performance implications of using actuators with discrete setting, especially MPL, and the benefit provided by using discrete actuators in a coordinated way across multiple tiers.

MPL regulates the number of concurrently executing requests for a service class, and hence controls service rates. For example, if the MPL for class 1 is 3 and the MPL for class 2 is 2, then class 1 receives 60% of the server capacity and class 2 receives the remaining 40% (assuming heavy load and that the server being controlled is a bottleneck). Another example of a discrete actuator is the assignment of requests to nodes in distributed systems. In this case, we increase service rates for a class by reserving more nodes for requests made by that class. A discrete valued actuator can only achieve a limited number of SLOs. We can achieve additional SLOs by dynamically switching between actuator settings, a technique that results in **controller induced variability**. Controller induced variability is very undesirable, especially for interactive users. We show that finer grain control is achieved by using actuators at multiple tiers, and hence control variability is reduced.

We define the control granularity of an actuator (or set of actuators used in combination) for a performance metric as the smallest increase in the metric that can be achieved in units of percent. We will qualify this by system state, typically by the response time of the system. Let R be the current value of the performance metric. Then, the control granularity g is defined in terms of the R' that is the smallest increase in R. That is,

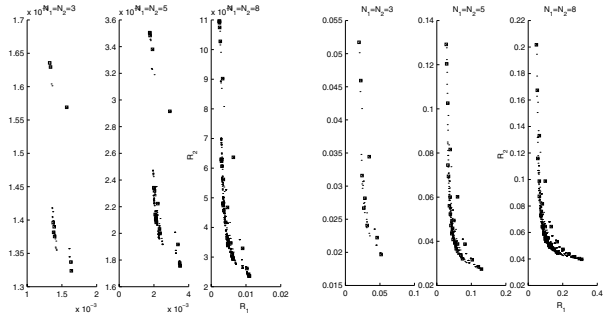
$$g = \min_{R'>R} \frac{R' - R}{R} \times 100 \tag{3}$$

To study the control granularity of MPL, we must consider the performance achieved for all possible combinations of MPL for the service classes on each tier. This is a large number. For example, consider a two tier system with two service classes. If each service class has five customers, there are $5^4 = 625$ possible settings of the MPL actuators.

It is impractical to conduct such a large number of experiments on a testbed. Hence, we develop a model of a system using T2T management of QoS. Since we must model MPL explicitly, the layered queueing network described previously in Section 3 is not sufficient. Rather, we use a closed queueing network with M tiers and K classes in which class k has N_k customers, and there are per class settings of MPL in each tier. For simplicity, we assume that there is a single server at each tier (although it is not difficult to generalize to multiple servers). In our model, a customer has at most one request outstanding and so we use the terms customer and request interchangeably. A customer is in one of the following states: (1) thinking (no request pending); (2) waiting to execute at tier m; and (3) executing at tier m. Note that if a customer is waiting to execute or executing at tier m, then it cannot be waiting to execute or executing at any other tier. After a customer completes its execution at tier m, the customer continues to tier $m + 1 \leq M$. Once a customer enters the tier m server, it consumes an MPL slot until it has completed its execution at tiers $m + 1, \cdots, M$. An execution completion at tier M results in a *service completion*, and the customer returns to the thinking state.

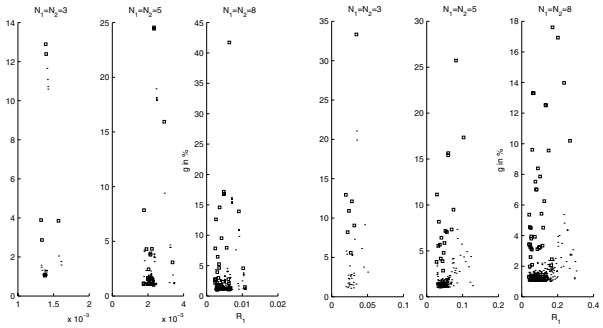
By assuming that think times and execution times are exponentially distributed, the above queueing network is a Markov Model. We use μ_{km} to denote the service rate of class k requests at tier m. The mean think time of a class k customer is $1/\mu_{k0}$. The state transitions in this model consist of (a) arrivals at tier m as a result of an execution completion at tier $m - 1$ and (b) execution completions at tier M, which then results in service completions in tiers $1, \cdots, M-1$. Table 1 compares testbed measurements with predictions obtained from the Markov Model on response time (in the unit of seconds), throughput, WAS utilization (in the unit of percentage), and DB utilization (in the unit of percentage) for different numbers of clients. While the fit is not exact, it is close enough to ensure that the Markov Model provides reasonable insights.

Figure 8 uses the Markov Model to plot response times



(a) Low Think Rate (b) High Think Rate

Figure 8. *Response times for a two class, two tier system. Squares indicate instances of one tier actuation.*



(a) Low Think Rate (b) High Think Rate

Figure 9. *Control granularity for a two class, two tier system. Squares indicate instances of one tier actuation.*

(in the unit of seconds) for different numbers of customers. Figure 8(a) plots response times for low think rates (i.e., light workload), and Figure 8(b) plots response times for high think rates (i.e., heavy workload). ("Low" and "high" are based on utilizations observed in our testbed.) In each figure, there are plots in which $N_1 = N_2 \in \{3, 5, 8\}$. The horizontal axis is the response time of class 1 (R_1) and the vertical axis is R_2. Each dot represents a different setting of the four MPL actuators (2 on each tier). The squares indicate actuator settings in which only the entry tier is used to control MPL. We refer to as **one tier actuation**. It is apparent that MPL settings that provide lower R_1 result in larger R_2, and the converse is true as well. The resulting curve appears to be a parabola that is symmetric around the line $R_1 = R_2$. Observe that there is a large number of MPL settings close to the center of the parabola, and few settings as we move further from the center of the parabola. Overall, there are far fewer settings for one tier actuation than settings that use actuators on both tiers.

We gain further insights by plotting g. Recall that g depends on state, which in our case means (R_1, R_2). From Figure 8, feasible values of (R_1, R_2) mostly lie on a parabola. Hence, state can be reasonably approximated by R_1 since R_2 is a function of R_1. Thus, Figure 9 plots R_1 on the horizontal axis and g on the vertical axis. As before, the dots indicate that MPL is controlled at both tiers, and the squares indicate instances of one tier actuation. Observe that for all plots, one tier actuation results in a significantly larger granularity of control.

If one tier actuation is used, then the control granularity may be coarse. Hence, we must oscillate between MPL settings to achieve intermediate values of response times. For example, consider Figure 8(b) in which $N_1 = N_2 = 3$. If one tier actuation is used, then there is no single setting of MPLs that provides a class 2 response time of 0.05. However, there are MPL settings such that $R_2 \in \{0.046, 0.053\}$. Hence, we achieve an average of $R_2 = 0.050$ by switching between these MPL settings in a way so that $p = 0.57$ of the time $R_2 = 0.053$ and $1 - p$ of the time $R_2 = 0.046$. In fact, the Autonomic Controller in Figure 1 would do exactly this.

Our motivation for studying granularity of control is that if it is too coarse, then we expect a substantial amount of controller induced variability. We investigate this using a simple analytic model. Let x_t be the value of the performance metric (e.g. response time) at time t. We want to set our actuators in a way that at steady state $x_t = x^*$. However, due to the granularity of control of the actuators employed, the steady state value of x_t is either x and x' such that $x < x^*$ or $x^* < x'$. That is, from Equation (3), $g = \frac{x'-x}{x}100$.

To analyze controller induced variability, we make the simplifying assumption that the x_t are independent random variables and that measurement and control intervals are long enough so that it suffices to use steady state values. The distribution of x_t changes over time as a result of changes in control settings needed to enforce SLOs when control granularity is coarse. We consider two settings of the actuators. In the first, the steady state value of x_t is x, and the variance of x_t is c^2x^2, where c is the coefficient of variation for x_t. In the second setting of the actuators, the steady state value of x_t is x' and its variance is $c^2x'^2$. Let p denote the fraction of time that actuators are set to achieve x' and so $(1 - p)$ is the fraction of time that actuators are set to achieve x. The variance of x_t is $\sigma^2_{x_t} = (1-p)c^2x^2 + pc^2x'^2 + (1-p)(x-x^*)^2 + p(x'-x^*)^2 = x^2\left((1-p)c^2 + pc^2(1+g)^2 + p(g-p)^2 + (1-p)(pg)^2\right)$, where $x^* = x(1-p) + px(1+g) = (1+pg)x$. Thus, the fractional increase in variance f_v due to controller induced variability is

$$f_v = \frac{(1-p)c^2 + pc^2(1+g)^2 + p(g-p)^2 + (1-p)(pg)^2}{c^2(1+pg)^2} - 1 \quad (4)$$

We use Equation (4) to study controller induced variability as a result of the control granularities in Figure 9. We see that if one tier actuation is used, then g often exceeds 25% and may be larger than 30%. With $p = 0.5$ and $c = 0.25$ (a low variance system due to good regulation), f_v is between 10% and 20%, which is a substantial increase in variance. We can gain further insight into the factors affecting f_v by examining Equation (4). Note that $f_v = 0$ if either $p = 0$ or $p = 1$. This makes sense since in both cases x^* can be achieved by a fixed setting of the actuators. Further, note that if we have a very fine granularity of control, which means that $g \approx 0$, then $f_v = 0$. Also, note that f_v increases with g and c. The former is intuitive since a large g means that there are big changes in the performance metric in order to enforce the SLO. The latter is a consequence of being able to better detect changes in the variability of x_t if the coefficient of variation of x_t is small. The effect of changing p is more complicated. In general, values of p such as $p = 0.5$ that are distant from both x and x' result in the largest f_v.

5 Conclusions

The tier-to-tier (T2T) management architecture supports decentralized management of actuators in multi-tier systems. The appeal of a decentralized approach is reducing message overheads and communication delays that occur in centralized schemes. We describe the elements in the T2T management architecture and a testbed implementation consisting of application and database tiers using the IBM WebSphere Application Server and DB2 Universal Database Server. Based on testbed experiments and analytic models, we gain insight into the value of coordinated exploitation of actuators on multiple tiers, especially considerations for control efficiency and control granularity. For control efficiency, we show that more effective utilization of tiers can be achieved by using actuators on the bottleneck tier rather than only using actuators on the entry tier. These efficiencies result from avoiding queueing delays imposed on requests at non-bottleneck tiers. Granularity of control is the ability to achieve a wide range of service level objectives, something that can be difficult with coarse actuators such as multiprogramming level (a commonly used technique because of the ease with which it can be manipulated). Our results show that a fine granularity of control can be achieved through a coordinated, cross-tier exploitation of coarse grained actuators. Doing so can greatly reduce the controller-induced variability that results from oscillating between actuator settings in order to achieve service level objectives with coarse grain actuators.

Our future work will provide more insight into the tradeoffs between centralized and distributed control of actuators on multiple tiers, especially regarding to the message overheads and communication delays and for a three-tier architecture. Further, we will investigate Autonomic Controller design that involves the interaction between control schemes used at different tiers (where bottleneck may appear at more than one tier) and consider experimental assessment for heterogenous workloads.

Acknowledgements

The authors would like to thank Mike Frissora and Jim Norris for their assistance on setting up the multi-tier system testbed.

References

[1] D. Menasce, D. Barbara, and R. Dodge, "Preserving QoS of e-commerce sites through self-tuning: A performance model approach," in *Proceedings of 2001 ACM Conference on E-commerce*, 2001.

[2] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Proceedings of the 2nd International Conference on Autonomic Computing*, (Seattle, WA), June 13–16 2005.

[3] S. Ranjan, J. Rolia, H. Fu, and E. Knightly, "QoS-driven server migration for internet data centers," in *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)*, (Miami Beach, FL, USA), pp. 3–12, 2002.

[4] M. Karlsson, C. Karamanolis, and J. Chase, "Controllable fair queuing for meeting performance goals," in *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation, Juan-les-Pins, France*, pp. 278–294, 2005.

[5] F.-L. Lian, J. R. Moyne, and D. M. Tilbury, "Network design consideration for distributed control systems," *IEEE Transactions on Control Systems Technology*, vol. 10, no. 2, pp. 297–307, 2002.

[6] A. Kamra, V. Misra, and E. M. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," in *Proceedings of the Twelfth International Workshop on Quality of Service (IWQoS 2004)*, (Montreal, Canada), 2004.

[7] IBM WebSphere Software. http://www.ibm.com/software/webservers/appserv/benchmark3.html.

[8] S. S. Lavenberg, ed., *Computer performance modeling handbook*. Orlando, FL: Academic Press, INC, 1983.

[9] J. A. Rolia and K. C. Sevcik, "The method of layers," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689–700, 1995.

[10] Y. Diao, J. L. Hellerstein, S. Parekh, H. Shaikh, M. Surendra, and A. Tantawi, "Modeling differentiated services of multi-tier web applications," in *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (Submitted)*, 2006.