Feedback Performance Control in Software Services

Tarek F. Abdelzaher, John A. Stankovic, Chenyang Lu, Ronghua Zhang, Ying Lu
Department of Computer Science, University of Virginia
Charlottesville, VA 22904

Software systems are becoming larger and more complex. At the same time they are being deployed in applications where performance guarantees are required. Traditional approaches to providing these performance guarantees are not effective for a large class of software systems. Recently, control theory has been identified as a promising theoretical foundation for performance control in complex software applications such as real-time scheduling, Web servers, multimedia control, storage managers, power control in CPUs, and routing in computer networks. In this paper, we describe advances in the application of control theory to software systems. We demonstrate the formulation of software performance assurance problems as those of feedback control. We describe modeling the software system and provide an example of performance control in contemporary Web servers. We embody the control-theoretical methodology for software Quality of Service provisioning into a middleware called ControlWare. It provides a generic interface between computing and control subsystems of a software application, and automates many parts of the feedback control design and implementation for software systems. Middleware solutions like ControlWare are needed to resolve the systems challenges and provide analytic foundations to enable efficient software performance control in next generation performance-assured computing systems.

Software Performance Control

While many engineered physical systems carefully address quality assurance, historically, software system design evolved in a more ad hoc fashion, with less rigorous guarantees on performance and quality. Most software engineering research is concerned with tools and paradigms that facilitate the development of functionally correct software. Functional correctness was implicitly assumed to be an adequate software quality metric.

There are several notable exceptions to the assumption that functional correctness is adequate. For example, in most embedded control systems, an important factor affecting quality of software is the timeliness of software response. Here, a delayed, but functionally correct, reaction to physical events in the environment can be as devastating as a physical equipment malfunction. This observation gives rise to research which attempts to design software with predictable non-functional

The work reported in this paper was supported, in part, by the National Science Foundation under grants ANI-0105873, CCR-0093144 and CCR-0098269, by the DARPA NEST program under grant F33615-01-C-1905, and a MURI award N00014-01-1-0576.

properties such as timeliness, security, availability, and performance. These properties are often referred to as software *quality of service* (QoS) attributes.

Traditional performance analysis of embedded control applications relies on worst-case estimates of load and resource availability. A system whose performance is guaranteed in worst-case conditions will not violate its performance specifications under more favorable circumstances. The notion of quality of service guarantees, however, is needed today in a much larger class of applications, which, unlike closed embedded systems, operate in open unpredictable environments. The worst-case scenario in such environments is not known *a priori*, or is very pessimistic, rendering traditional worst-case analysis impractical.

The need for quality of service guarantees in open systems is fueled, in part, by the emergence of modern global communication networks such as the Internet, and the increasing proliferation of globally accessible digital services such as online banking, trading, and distance learning. Such services represent points of massive aggregation, which may suffer from unpredictable loads, potential bottlenecks, and security breaches such as denial of service attacks. Failure to meet acceptable performance specifications may result in loss of customers, financial damage, or liability violations. Existing approaches for designing performance-guaranteed computing systems which rely on *a priori* workload and resource knowledge are no longer applicable. Instead, the predominant practice for providing quality of service assurances today is overdesign. This practice results in costly systems with uncertain assurances regarding performance. Putting performance guarantees on a solid analytic foundation in such systems is an important challenge.

In this paper, we show how classical feedback control offers a solution to the problem of achieving performance guarantees for this new category of applications and discuss the challenges involved in realizing this approach. The main challenges in implementing a feedback-based QoS control solution in computing systems are (i) analyzing the software architecture with the purpose of modeling it as a feedback control system, (ii) mapping the particular QoS control problem into a system of feedback loops, (iii) choosing a proper actuator that can affect server resource allocation, and a monitor that can measure current performance reliably, and (iv) designing a controller for the server. Solving these challenges permits the mathematical basis of control theory to support the performance guarantees of software systems. We also demonstrate that a software system can be approximated by a linearized model and describe the needed software actuators and sensors. Importantly, we create a taxonomy of the most important QoS assurance problems in the software literature and describe how they can be translated into a feedback control formulation.

We also present some experimental results for a web-based application that shows that the control-theoretic approach is feasible for software systems. Although our approach can be applied to other services as well (and we provide brief explanations of those) we have chosen to focus on controlling the performance of web applications due to the increasing importance of the web, spurred by the phenomenal growth of the Internet. The web is the largest and most visible client-server application in existence today. It is an excellent vehicle for investigating fundamental problems of distributed client-server computing such as that of overload protection and providing performance guarantees.

In the rest of this paper, we describe the internal architecture of a typical web server, present its dynamic model, and elaborate on the equivalents of sensors and actuators in the computing system. We then map the most important software performance assurance problems into those of feedback control, and piece these elements together in a case study on web server performance control. We also describe briefly other examples of the use of control-theory for QoS guarantees. Finally, we introduce a middleware service that generalizes the control-theoretic approach, implementing a new paradigm for service performance guarantees in software systems. The paper concludes in with a brief discussion of important results and remaining challenges.

Inside a QoS-aware Service

A successful architecture for controlling performance of a computing system begins with an understanding of the dynamics that affect performance and the mechanisms available to manipulate those dynamics. In this paper, we are concerned specifically with performance attributes that involve a notion of time, since they lend themselves more naturally to a feedback-control framework. Generally, the most common of such performance attributes are classified into two categories depending on whether they are directly proportional or inversely proportional to time. The former category includes performance metrics such as queuing delays, execution latencies, and service response times. The second category includes metrics such as connection bandwidth, service throughput, and packet rate. We call them *delay metrics* and *rate metrics*, respectively. There are also *derived metrics* defined as ratios between other metrics, for example the relative delay of two traffic classes, or the hit ratio (i.e., the fraction of hit rate to the total request rate) of a cache.

Time-related performance attributes can generally be controlled by adjusting resource allocation. Queuing theory has often been used to predict performance given a particular resource allocation, or to determine how resources should be allocated to yield a particular performance level [25]. In general, if the service is modeled as a queuing system, allocating more resources to the server of the queue will reduce the mean service time, decrease the mean queuing delays, and increase the average service rate. Unfortunately, queuing theory generally requires assumptions about the input traffic arrival pattern which are not always accurate, leading to potentially poor predictions regarding performance. For example, a significant body of queuing literature applies to Poisson arrivals, whereas many common arrival patterns, such as those of web requests, are known to follow a heavy tailed distribution [8, 24]. Even when all assumptions are accurate, queuing theory, by nature, offers only predictions on average behavior. In a QoS-aware system, stronger guarantees are often required. For example, it is not enough that the frame rate of a streaming video be 30 frames/sec on average. Instead, guarantees are usually required on maximum deviation and recovery time from transient perturbations around the nominal rate.

Service Architecture

To design feedback loops for software performance control, we need to understand the basic components of a software system and how they interact. We focus on an important category of software systems where feedback control is particularly important; namely, software servers. In this section, we present the typical design of a multi-threaded server and describe how this design yields a system model suitable for feedback control.

Consider a distributed client-server system in which a succession of requests arrive at a server from clients across a communication network. The server acts as a point of aggregation of client requests. The performance observed for a request at the server depends on the path the request takes inside the server until it is served. In a typical multithreaded server, independently schedulable entities, called *worker threads*, execute the arriving sequence of clients' requests. Each thread implements a loop that processes incoming requests and generates responses.

Client requests are first read from an input queue (such as the server TCP socket queue) by a kernel entity that hands each request to a different worker thread. Worker threads which have requests to serve become runnable and are queued for the CPU. The order in which threads get the CPU to execute is determined by the CPU scheduling policy. This policy maintains a priority queue called the *ready queue*. The thread at the top of this queue executes for a particular time quantum or until it is blocked. Request processing by a worker thread typically entails access to one or more auxiliary server resources, the most notable being disk I/O. For example, in a web server, disk I/O is usually needed to read the requested web page from disk. Access to auxiliary resources blocks the calling thread, at which time it is queued for I/O until the awaited resource becomes available. Each resource usually has a queue which determines the order in which accesses are served. The resource is made available to the thread at the top of the queue, at which time the corresponding thread becomes runnable again and re-enters the CPU ready queue. When request processing is done, the worker thread sends a response back to the client. Sending the response entails enqueuing data into the outgoing packet queue for transmission on the network.

Figure 1-a illustrates the aforementioned main components of a software server, including the input client request queue, the CPU ready queue, the resource I/O queue, and the outgoing network queue. Numbered arrows depict the progress of requests from one queue to another. Worker threads are also shown. The black circle on each thread represents the current position of thread execution.

We are especially interested in the case were the server is operating at non-trivial load. Hence, response time is dominated by queuing delays rather than service times. For the server to yield acceptable performance it must be that the order of request service times is much smaller than the order of tolerable server response times. If it takes C_i units of time to serve request i, and if D_i is the maximum tolerable server response time, then typically $\forall i, j : C_i << D_j$. We call this condition the *liquid task model*. The model is representative of high performance servers which handle many thousands of requests per second. For example, in the case of web servers, a worst-case tolerable response time would be of the order of seconds to tens of seconds, whereas a typical request service time would be of the order of hundreds of microseconds to single milliseconds. The liquid task model represents the limiting case in which tolerable response times are generally finite, yet the service times are practically infinitesimal. More formally, in the liquid task model, $\forall i : C_i \rightarrow 0$, and $\forall i : C_i/D_i \rightarrow 0$. While this model is an idealization, results based on this model hold well in systems exhibiting a large number of small requests.

In a high performance server approximated by a liquid task model, the progress of requests through the server queues resembles a fluid flow. The service rate, $dN_k(t)/dt$, of stage k, defines the amount of flow through that stage, where $N_k(t)$ is the total number of requests served by this stage by time t. The different queues in Figure 1-a can therefore be modeled as capacities that accumulate the corresponding flows. The number of requests queued at stage k, denoted V_k, is a quantity akin to volume, given by $V_k = \int_{-\infty}^{t} (F_{in} - F_k)dt$ where F_k is the service rate of stage k, i.e., $F_k = dN_k(t)/dt$, and F_{in} is the request arrival rate to that stage. Queues also offer points at which flows, F_k, can be manipulated. Figure 1-b depicts the server from a control perspective where capacities are represented by water tanks. Observe that "valves" in Figure 1-b represent points of control, i.e., actuators, which manipulate the service rates F_k.[1]

It is important to notice that the fluid flow analogy does not make assumptions on how individual requests are prioritized. It simply allows us to describe the dynamics of request flows and queue fill levels. In contrast, queuing theory and real-time scheduling theory have well-understood foundations for relating aggregate metrics such as queue-length, total workload, or total utilization to delays seen by individual requests under a particular prioritization policy. Hence, combining control theory with real-time scheduling or queueing analysis, one can develop feedback loops to maintain appropriate queue fill levels that are guaranteed to produce the desired client delays under different request prioritization schemes. Most of today's servers implement FIFO queueing on resources (such as socket queues and semaphore queues) and are composed of pools of same priority threads or processes. Hence, in this paper, we assume that queues are FIFO. Multiple client classes may exist, however, each with their own queue of a different priority.

[1]We assume in this analogy that flow through the valve depends only on valve opening and not on the liquid level. In that, the arrangement is perhaps more akin to a pump.

Service Modeling

Internet servers described by the liquid task model have dynamics akin to those of flow due to their intrinsic queuing structures. This observation motivates the use of difference equations to model Internet servers. Let $y(k)$ denote the average performance (e.g., delay or throughput) reported at the kth sampling instant. It reflects a measurement carried out during the most recent sampling interval. Let $u(k)$ denote the control input at that sampling instant. A server can be modeled by an nth order ARMA difference equation relating $u(k)$ to $y(k)$. The system order n represents the length of history that determines the current server performance. Figure 1-b suggests that the difference equation can be derived from a state space representation of the server model:

$$\begin{aligned} \mathbf{x}(k) &= A\mathbf{x}(k-1) + bu(k) \\ y(k) &= C\mathbf{x}(k) \end{aligned}$$

where $\mathbf{x}(k)$ is the state vector, A, b, and C describe the system model. Representing software system dynamics by difference equation models has recently gained much popularity. For example, in [21] a model of a web proxy cache is derived, and in [14] a model of TCP dynamics is presented in the presence of network active queue management based on RED gateways [13].

Derivation of computing system dynamics can be difficult in the absence of complete knowledge of software system code due to hidden interactions between different components in such systems. For example, a semaphore used for synchronization can create a waiting queue that affects system dynamics. Yet, the presence of such a semaphore might not be discovered without access to system code. Furthermore, deriving system dynamics from first principles may be impractical in some situations for administrative reasons. For example, the model parameters may need to be recomputed every time the server's hardware or software configurations are changed (e.g., due to a system upgrade), since such changes may affect platform speed and consequently the rates and capacities in the system. Moreover, system administrators usually lack the expertise to perform control-theoretic modeling. Therefore, a more practical approach may be to perform automatic parameter estimation, e.g., using a least squares estimator. System identification requires adding software modules to instrument a running software system and iteratively estimate the parameters of difference equation models based on system input and output. A successful application of the approach is described in [20] for a web proxy cache to determine the relation between disk allocation and cache hit ratio.

A key prerequisite to system modeling is to define the actuators and their manipulated variables, i.e., the valves in Figure 1-b. These actuators represent the main interface between the control subsystem and the computing subsystem in the feedback control loop. In the next section, we describe the types of actuators in computing systems and their principles of operation.

Resource Allocation for QoS-Guarantees

Most time-related performance metrics in software systems can be tied to the state of the queues depicted in Figure 1. Delay metrics such as response time and latency are generally related to the queue (tank) fill levels. Higher fill-levels imply longer delays and vice versa. Rate metrics such as service throughput are generally related to the dequeue rates (i.e., flows through the valves). Higher rates imply higher throughput. Both sets of metrics can be affected by controlling the different flows in the system. The flow, i.e., the service rate, of a stage in the software service depends on the amount of computing resources allocated to this stage and the operation being performed. Allocating more resources will generally increase the flow the same way opening a valve would. Thus, to control performance, we need actuators which manage resource allocation or alter the functionality of the server in a way that manipulates the rate at which work is done. The feasibility of implementing such actuators in a computing system is one basic reason of why control theory is applicable. Actuators in computing systems fall into three basic types as described below.

Input Flow Actuators

An input flow actuator manipulates the input workload of a server, F_{in}. In software systems, *admission control* is a primary input flow actuator that affects queue fill levels in the server. In the simplest case, admission control limits the number of clients who access the server concurrently by denying service to some of them hence reducing the load seen by the server. The actuator accepts as input a control variable m which determines the desired average flow. This is analogous to controlling the opening of input valve V1 in Figure 1-b and discarding all requests that overflow. A more intelligent admission control algorithm decides, on a request-per-request basis, whether service should be granted (request admitted) or denied (request rejected), perhaps depending on client identity, queue fill level, and server utilization. Input flow actuators are used when some clients are inherently more important than others such that denying service to less important clients is acceptable in favor of higher priority clients. Examples of the control-theoretic formulation of admission control schemes in real-time systems are described in [19, 27].

Quality Adaptation Actuators

A more flexible way of controlling flow is to alter the processing requirements of the request. For example, reducing the processing requirements increases service rate, i.e., increases $F_k = dN_k/dt$, which is desired when the server approaches overload. To reduce processing requirements, the service offers intermediate "degraded" service levels. In a web server, a degraded level of service can correspond to providing an abbreviated version of content, or a lower quality version of images and sound. The approach is often followed manually by editors of various sports and news web sites such as the Cable News Network, `www.cnn.com`, upon important breaking news. An example of that was the obvious abbreviation of the CNN front page in the first hours following the September 11th events in 2001. An important question is, how much content should be abbreviated and what fraction of clients need to receive the abbreviated version for the overload to be avoided. An automatic web content adaptation scheme that answers these questions by using feedback control theory is described in [3]. The scheme dynamically selects a suitable content version for each client on a request-per-request basic depending on load conditions. Content adaptation has also been discussed at length in multimedia applications. A good survey is found in [4].

In general, a quality adaptation actuator offers a flexible tradeoff between delay and quality. Since servers don't have unlimited queue space, some compromise is often unavoidable to prevent server queue overflow at extreme load conditions. An advantage of quality adaptation actuators over admission control schemes is that service is provided to *all* clients, albeit potentially at a degraded level. For effective load reduction, it is important that the actuator control the bottleneck resource. For example, if the bottleneck resource is the processor, CPU-intensive operations should be reduced. If the bottleneck is the network, the number of sent bytes should be reduced. The identity of the bottleneck can be measured dynamically by measuring the utilization of different resources.

To approximate flow, a quality adaptation actuator should provide a continuous range of service rates, F_k, even when the server exports only a finite (small) number of different service levels. Consider a general server with M discrete service levels that differ in their consumption of the bottleneck resource. Let these levels be numbered $1, ..., M$ from lowest quality to highest quality. The level 0 may be added to denote the special case of request rejection. The actuator accepts as input the control variable m in the range $[0, M]$. An essential challenge in actuator design is that of unique mapping from m to the manipulated variables, namely, the fraction of clients served at each QoS level to produce smooth flow control.

To resolve the aforementioned challenge, the authors of [2] propose to decompose the fractional value of input, m, into an integral part I and a fraction F, such that $m = I + F$. The two nearest integers to m (namely, I and $I + 1$, where $I < m < I + 1$) determine the two most appropriate service levels at which clients must be served under the given load conditions. The fractional part F determines the fraction of clients served at each of the two levels. In effect, m is interpreted to mean that a fraction $1 - F$ of clients must be served at level I, and a fraction F at level $I + 1$. The scheme works well when the input average request arrival rate does not change quickly.

Server utilization increases (possibly nonlinearly) when m is increased and vice versa. At the upper extreme, $m = M$, all requests are given highest quality service. At the lower extreme, $m = 0$, all requests are rejected. Hence, the actuator changes the amount of load on a server with discrete service levels in a continuous fashion, depending on its input m.

Resource Reallocation Actuators

A different way of controlling flow (other than quality adaptation) is to alter the amount of resources available to the processing of the request queue. This important category of resource management actuators typically arises in servers with multiple classes of clients. In such servers, it is often desired to maintain some constant ratio between the performance of different client classes. The computing resources are usually partitioned among these classes such that the sum of all partitions is constant and equal to the total resource capacity of the service. We call this condition, the *constant-sum* invariant. Individual actuators can alter the partitioning, but the total resource allocation must remain constant. We call such actuators, *resource reallocation actuators*.

In a multi-class server, a separate control loop is often associated with each client class. An actuator in each loop alters the resources allocated to it. An important design objective of this approach is to ensure that individual loops may operate in isolation without violating the constant-sum invariant. We demonstrate how the above is achieved with an example. Consider a multi-class service, which offers some classes of requests better performance than others by allocating resource space among different request classes appropriately. Let the measured performance of request class i be H_i. One can think of H_i as the output of the ith control loop. A very common model for discrimination in a multi-class environment is the relative differentiated services model [11, 12]. In this model, a differentiation policy specifies that the measured performance of different classes should be related by the expression: $H_1 : H_2 : ... : H_n = C_1 : C_2 : ... : C_n$, where C_i is the importance weight of class i. In a control-theoretic formulation, we define the *relative* performance, R_i, of class i as $R_i = H_i/(H_1 + H_2 + ... + H_n)$. It determines how the class is performing relative to other classes. The desired relative performance of class i should be $R_{i_{desired}} = C_i/(C_1+C_2+...+C_n)$. It represents the set point for the class. This formulation is akin to ratio control [29]. The difference $R_{i_{desired}} - R_i$ is the performance error e_i of class i. Note that the aggregate performance error of the system is always zero, because $\sum_{1\leq i\leq n} e_i = \sum_{1\leq i\leq n}(R_{i_{desired}} - R_i) = \frac{\sum_{1\leq i\leq n} C_i}{C_1+C_2+...+C_n} - \frac{\sum_{1\leq i\leq n} H_i}{H_1+H_2+...+H_n} = 1 - 1 = 0$.

The aforementioned problem formulation was used in the differentiated caching services architecture described in [20, 21]. The authors developed a web proxy cache that can give some web content preferential treatment. In this example, the performance metric being controlled is cache hit ratio and the resource being allocated is disk space. There are n content classes in the system.

Each class of content i is assigned a different amount of cache storage s_i, such that $\sum_i s_i$ is the total size of the cache. A controller is invoked at fixed intervals for each class at which it corrects resource allocation to that class based on the measured performance error. To compute the correction $\delta s_i[k]$ in resource allocation, the controller uses a linear function $f(e_i)$ where $f(0) = 0$. If the computed correction $\delta s_i[k]$ is positive, the space allocated to class i is increased by $|\delta s_i[k]|$. Otherwise, it is decreased by that amount. Since the function f is linear, $\sum_i f(e_i[k]) = f(\sum_i e_i[k])$. Since we showed that $\sum_i e_i[k] = 0$, it follows that $\sum_i f(e_i[k]) = f(0) = 0$. Hence, the sum of corrections across all classes is zero, i.e., the constant sum invariant is maintained.

Many computing systems with multiple client classes use similar per-class feedback loops [7, 16]. The difference between these systems lies generally in the resource being allocated by the actuators. For example, instead of manipulating storage allocation, actuators can be built to manipulate the number of worker processes (and hence CPU capacity) allocated to a class [16], or the fraction of network link bandwidth allocated to a flow [7].

This section discussed the natural existence of actuators in computing systems, which makes it possible to implement the "valves" which appear in Figure 1-b. Another important cornerstone of applying feedback control to computing systems, is the existence of a natural translation from common QoS assurance problems into those of feedback control. This topic is covered in the next section.

QoS Mapping

A cornerstone of a control theoretic paradigm for QoS guarantees in software systems lies in the ability to convert common resource management and software performance assurance problems into feedback control problems. One can think of each QoS control problem as having a corresponding control loop instantiation that describes how this particular QoS control problem is solved using feedback control. We call such an instantiation, a *control loop template*. In this paper, we describe control loop templates for the main QoS control problems such as absolute convergence guarantees, performance isolation, statistical multiplexing, prioritization, relative differentiated service guarantees, and optimization guarantees. The fundamental building block in these templates is one that implements the basic (absolute) convergence guarantee. Interconnecting such blocks together can lead to formulating more complex guarantees such as relative guarantees, prioritization, and optimization as feedback control problems.

The Absolute Convergence Guarantee

Since it is impossible to achieve absolute guarantees in a system where load and resources are not known *a priori*, we define the absolute guarantee problem as one of *convergence* to a specified performance. The statement of the problem is to ensure that a performance metric, R, (i) converges within a specified exponentially decaying envelope to a fixed value, $R_{desired}$, and that (ii) the maximum deviation $R_{desired} - R$ is bounded at all times, as shown in Figure 2-a.

The absolute convergence guarantee is translated into the control loop shown in Figure 2-b. The loop samples the measured performance, compares it to the desired value $R_{desired}$, and uses the difference to induce changes in resource allocation via the actuator $A(R)$. The absolute convergence guarantee loop is the elementary building block from which all other QoS assurances follow. In the context of time-related performance metrics, it is interesting to classify the convergence guarantee loops depending on the performance variable being controlled. As is the case with physical plants, the controlled output of interest affects the model of the system and whether the control loop is linear or not.

- *Rate and queue-length control:* To a first approximation, rate metrics and queue length are easiest to control because they result in linear feedback loops. The (flow) rate can be controlled directly by the actuators. Queue length can be linearly controlled by controlling the flow. The simplest example of a loop of this category is a server utilization control loop. Such a loop, for instance, can be used to maintain high server utilization while avoiding overload. In this section we make several uses of utilization control as a basic building block for other types of guarantees.

- *Delay control:* Delay guarantees are more difficult to provide. This is because delay is inversely proportional to flow. If a request arrives at a queue of length Q, with a dequeue rate of r, the queuing delay, d, of the request is $d = Q/r$. In general, since the rate r changes over time, the delay of this request is $d = \int_0^Q dq/r$. The inverse relation between the manipulated variable (rate) and the delay makes the control loop nonlinear. Note that, while level control (i.e., queue fill-level control) is a very common problem in physical plants, one generally does not care how long it takes a particular liquid molecule to leave the tank. Thus, the non-linear delay control problem is less commonly encountered in physical process control. We believe that an efficient solution to this type of non-linearity will have great impact on software QoS-control research.

The Resource Reservation Guarantee

In some applications it is necessary to guarantee that certain resources be reserved. The absolute guarantee solution, described above, is particularly useful for resource reservation for the purpose of isolating applications that share common resources. For example, in a web hosting application, the web server may offer a “virtual-estate”, such as percentage of server capacity, for sale to hosted web sites. The particular capacity allocation purchased by a given site can be enforced using the control loop in Figure 2-b. The amount of server capacity owned by the site will represent the utilization control loop set point. The actual amount of resources used can be measured dynamically. The difference controls the actuator, which in this case can be an admission controller for requests for the particular site. One control loop is needed per web site.

The Prioritization Guarantee

To see how prioritization can be implemented using feedback control, consider a sampled system in which a large number of new service requests is introduced at each sampling time. Requests are classified by priority and are enqueued in a separate queue depending on their class. The scheduling policy depletes one queue at a time in priority order during the sampling interval. By the end of the sampling interval some queues will be fully depleted, at most one queue will be partially depleted (that's the one the scheduler was working on when the next sampling time arrived), and the rest will remain untouched. To emulate this effect with control loops, imagine each queue being a separate pipe with its own control valve. Given server capacity, the valves of queues that can be fully depleted are saturated in the open position. The valve of the next queue is controlled to dequeue exactly the right fraction of requests within the sampling interval to fully utilize any leftover server capacity. All the remaining (lower priority) valves are closed. Below, we illustrate a feedback control scheme that achieves the above. Note that the scheme is unconventional in that at most one loop is operating at any given time outside of saturation limits. The rest are saturated in either the fully open or the fully closed position as explained above. Integrator anti-windup is used to prevent unreasonable integral action buildup in the controller during saturation periods.

Let there be M priority classes defined within a server, such that priority 1 is highest, and priority M is lowest. Collectively, clients of the server are allocated a target utilization U^*. This capacity should be made available to clients in priority order. A resource allocation loop can be created which gives the entire server capacity to the highest priority class. The unused capacity of each class is measured and treated as the set point for the resource allocation to lower priority classes. If this capacity is not enough, these clients are degraded or rejected accordingly by the actuator in their utilization control loop. The architecture is described for a two class server in Figure 2-c. One control loop is needed per class.

For illustration, consider the operation of a system with two priority classes. When the high-priority class is getting more than its resource share, the controller error of the high-priority loop is negative. In this case (i) the class is controlled to reduce its consumption to the allotted share, and (ii) the lower priority class has a zero resource allocation. The first condition is ensured by the loop of the higher-priority class. The second is ensured by the other loop since its set point will be negative in this case, causing its actuator (e.g., the admission controller) to saturate at a fully closed position. Conversely, when the higher priority class does not consume all its allocated resources, the error (left-over resources) is positive, which is the set point to the lower priority loop. The actuator of the high-priority class is saturated at a fully open position admitting all requests of this class. The loop for the lower priority class makes sure only as many requests are admitted as permitted by the left-over resources from the high-priority class. Hence, the loops emulate priority semantics.

The Statistical Multiplexing Guarantee

One shortcoming of resource reservation is that it can lead to unnecessary resource underutilization when some resource owner does not use all the resources allocated to it. This is objectionable if some other owner is starving for resources in the meantime. Hence, a statistical multiplexing scheme is needed whenever spare capacity exists on the machine. Individual resource allocations should be enforced only when the entire machine is overloaded. These two requirements can be simultaneously satisfied by the control loop set in Figure 2-d.

To describe how statistical multiplexing is achieved, assume that the controller output in the utilization control loop of a server i is m_i. All surplus requests are treated as a lower priority server, called the best effort server. Let the controller output of the utilization control loop of the best effort server be m_b. Sharing of spare capacity is achieved using a variation of a min-max scheme [29] in which the objective is to minimize degradation (loss) of premium traffic in the worst-case. In this scheme, the service level of a request for a given first class server i is determined by the actuator of premium traffic if $m_i > m_b$ and with the actuator of best effort traffic otherwise. Thus, the request is handled according to the higher of m_i and m_b. When the first class server is overloaded while the machine as a whole is not, $m_b > m_i$. Consequently, incoming requests are served with quality determined by m_b which is higher than that warranted by m_i thus utilizing excess machine capacity. On the other hand, if the machine is overloaded, $m_b < m_i$. Consequently, the quality of content delivered by server i is determined by m_i. Thus, the individual server is prevented from exceeding its capacity allocation. The mechanism allows smooth and informed switching between a mode of operation where an individual server i is allowed to exceed its capacity allocation and a mode of operation where it is required to stay within its original capacity.

The Relative Guarantee

In some applications it is desired that the ratio between the performance of different classes of work be fixed, e.g., it may be that the delays of two traffic classes in a network should be fixed at a ratio of $3:1$. This fixed ratio is a good candidate for the performance set point, R. We can therefore pose relative guarantees as a variation of the ratio control problem [29]. In general, if there are n request classes in the system, and if H_i is the measured performance of class i, the relative guarantee specifies that $H_1 : H_2 : ... : H_n = C_1 : C_2 : ... : C_n$, where C_i is the weight of class i. One way to achieve this guarantee is to formulate per-class relative performance set points in the form $R_{i_{desired}} = C_i/(C_1 + C_2 + ... + C_n)$ and compare each against $R_i = H_i/(H_1 + H_2 + ... + H_n)$. Note that this loop (same as in Figure 2-b, but with the set point defined above) is nonlinear, since the sensor divides outputs of different loops to provide feedback on the relative performance of individual classes, as well as because flow is inversely proportional to delay.

It is possible to formulate the relative guarantee problem in a way that avoids the former nonlinearity. In this case, each output H_i is multiplied by the constant $W_i = C_1C_2...C_n/C_i$. Note that when the performance metric is satisfied (i.e., when $H_1 : H_2 : ... : H_n = C_1 : C_2 : ... : C_n$), the ratio H_i/C_i is equal for all classes. Thus, the products H_iW_i are equal for each class i. Let the set point be the average of these products, i.e., $R_{desired} = \sum_{i=1}^{n}(H_iW_i)/n$. Hence, when the performance metric is satisfied, all H_iW_i are equal to the set point. When the metric is not satisfied, the error e_i of class i is given by $e_i = R_{desired} - H_iW_i$. A control loop is designed per class with the objective of driving the error of that class to zero. The control loop in this case is linear, since W_i for a given class is constant. However, the control loops of different classes are coupled through their common set point which is a weighted sum of the outputs of all loops.

Finally, yet another way of defining the control loops is to define set points on pair-wise relative performance metrics C_i/C_{i-1}. In this case only $n-1$ loops are needed. Each loop reduces the corresponding error $e_i = C_i/C_{i-1} - H_i/H_{i-1}$ to zero. In this case, each loop decides on the ratio of resource allocation between two adjacent classes. Resources are then globally reallocated such that (i) all pair-wise ratios are satisfied, and (ii) the sum of allocated resources is equal to the total resource capacity.

The Utility Optimization Guarantee

Another type of performance guarantee addressable using a control-theoretic framework is that of utility optimization. Following a microeconomic model [22], consider a computing service which produces an amount of work w. Let the benefit per unit of work be k. Hence, the total utility U produced by the service is $U = kw$. Let the resource consumption of the service be some nonlinear function, $g(w)$, which represents a measure of cost. It is desired to achieve the maximum net profit, i.e., maximize $kw - g(w)$. Assuming a concave cost function, $g(w)$, the profit is maximized when the marginal utility is equal to the marginal cost, or when $\frac{dg(w)}{dw} = k$. The equation can be solved for w which then becomes the control set point, R. In a computing example, w, may be the desired server utilization, the desired workload size, or other metrics depending in the problem formulation. The resulting feedback loop is illustrated in Figure 2-b, where the set point is derived as described above.

In summary, the important point is that we have identified the main types of performance guarantees that cover the performance requirements of many software systems. For each type of guarantee we have specified a feedback control loop template that can be used as a basis to provide that type of guarantee. We believe that these templates, and the nonlinearities they exhibit, provide an interesting opportunity for exploring new control techniques which target the characteristic couplings and nonlinearities of loops implementing guarantees on time-related software performance metrics.

Feedback-Based Solutions

In this section, we bring together the concepts introduced above by describing several successful applications of the control-theoretic approach in the software domain. We first give a detailed example involving Web servers, and then briefly introduce other applications on Internet servers, CPU scheduling, networking, and microprocessor architectures. The success of feedback control theory in these different domains demonstrates the generality and strength of the control theoretic framework when applied to computing systems.

A Detailed Example on Relative Delay Guarantees in Web Servers

As the Internet becomes commercialized, it generates an incentive for e-businesses to provide guaranteed shorter service delay to premium customers, either because they pay higher fees or are more important to the business than the other customers. Web servers have to handle disturbances caused by highly unpredictable workload. For example, the population in each customer class often varies dramatically at run-time, yet offered performance has to remain constant.

In this section we show an example of applying feedback control theory to provide relative delay guarantees on Web servers. Every HTTP request over the Internet belongs to a class i. A desired relative delay W_i is assigned to each class i. The service delay $C_i(k)$ of class i is defined as the average delay of established connections of class i measured within the kth sampling period $((k-1)T, kT)$ sec, where T is the constant length of the sampling period. A relative delay guarantee requires that $C_j(k)/C_i(k) = W_j/W_i$ for any classes j and i. For example, if class 0 has a desired relative weight $W_0 = 1$, and class 1 has a desired relative weight $W_1 = 2$, the relative delay guarantee requires that the delay of class 0 should be half that of class 1.

Sensors and Actuators

The first step in the feedback control design is to identify the controlled and manipulated variables in Web server systems. We focus on the Apache server and HTTP 1.1 requests. Apache is currently the most popular Web server software. The server runs a pool of server processes listening to a common TCP port. Each server process can only handle one TCP connection at any time instant. The HTTP 1.1 protocol works as follows. 1) A client (e.g., a Web browser) establishes a TCP connection with a server process; 2) the client submits an HTTP request to the server over the TCP connection; 3) the server processes the request and generates a response; 4) the server sends the response back to the client over the TCP connection; 5) to amortize the overhead of TCP connection establishment, the TCP connection is left open after the response has been transmitted in anticipation that the same TCP connection can be reused by a following HTTP request from the same client; if a new HTTP request arrives within a configurable keep alive interval (e.g., 15 sec), the TCP connection is kept open; otherwise the connection is closed. This feature is called "persistent connections".

From the clients' perspective, the delay of an HTTP request includes three components: the connection delay on the server for establishing a TCP connection with a server process, the processing delay on the server for local computations (network protocol processing, open/read files, and running CGI scripts), and the network delays for transmitting the TCP connection request, the HTTP request, and the response over an established TCP connection. In this example we focus on controlling the connection delay. The connection delay can be significant in overload conditions when all server processes are tied up with existing connections. When the server is highly loaded, all new TCP connection requests are queued in the TCP listen queue until they are accepted by a server process. The connection delay of a service class can be controlled by manipulating the process budget $B_i(k)$, i.e., the number of server processes allocated to class k in the sampling period $(kT, (k+1)T)$. Increasing the process budget of a class leads to a shorter connection delay for this class.

A relative guarantee scheme is used to provide relative delay guarantees in the Apache server. The connection delay ratio of two adjacent classes i and $i-1$ is controlled by a controller CR_i. For CR_i, the controlled variable is the desired connection delay ratio $V_i(k) = C_i(k)/C_{i-1}(k)$ between classes i and $i-1$, and the reference is the desired delay ratio W_i/W_{i-1}. The manipulated variable is the process budget ratio $U_i(k) = B_{i-1}(k)/B_i(k)$ between class $i-1$ and i. Note that the class of the denominator and the numerator is reversed in our notations of $V_i(k)$ and $U_i(k)$ due to the inverse relation between service rate and delay. At run-time, the controller CR_i periodically updates the process ratio $U_i(k)$ and invokes an resource reallocation actuator to allocate server processes to different classes according to $U_i(k)$. The goal is to control the delay ratio $V_i(k)$ to remain close to the reference $W_i(k)/W_{i-1}(k)$.

System Identification

As an instance of the server architecture described earlier, the Web server has dynamics caused by the queuing of TCP connection requests. To compute the dynamic model between the connection delay ratio and the process budget ratio, we explore the system identification approach. We approximate the server with an ARMA difference equation:

$$V(k) = \sum_{j=1}^{n} a_j V(k-j) + \sum_{j=1}^{n} b_j U(k-j)$$

In an n^{th} order model, $2n$ parameters $\{a_j, b_j | j = 0..n\}$ need to be estimated. We estimate the open loop dynamics of the Apache web server using pseudo-random digital white noise as input. The input signal randomly changes the process ratio between two different levels. A least squares estimator is invoked periodically to estimate model parameters at every sampling instant.

We conduct system identification experiments to model a real Apache web server on five Linux PC's connected with a 100 Mbps Ethernet. A Linux PC runs an Apache server together with our system identification software, and four Linux PC's run a SURGE workload generator [5] to simulate 400 clients. Figure 3-a shows that the estimated parameters of a second-order model at successive sampling instants in a 30-minute run (the system identification is started at 2 min after the run starts to give SURGE time to fully start up). The estimations of the parameters (a_1, a_1, b_1, b_2) converge to (0.74, -0.37, 0.95, -0.12).

To verify the accuracy of the model, we re-run the experiment using a white noise input with a different seed and compare the actual delay ratio to that predicted by the estimated model. Figure 3-b shows that the prediction of the estimated model is consistent with the actual relative delay throughout the 30 minute run. These results demonstrate that the real Apache server can be modeled as a second-order difference equation. Experiments also show that an estimated first-order model had larger prediction error (Figure 3-c) than the second-order model, while an estimated third-order model (Figure 3-d) did not improve the modeling accuracy. We choose the second-order model is as a tradeoff between the model accuracy and complexity.

Note that while the general server model shown in Figure 1-b suggests a high-order system, in general the server is dominated two important bottlenecks. The first is the client request queue on the server's input port. Hundreds of requests are often queued there until they are dequeued by some server process. The second queue is the CPU ready queue. More than one hundred server processes are usually dequeuing requests concurrently. The ready queue, therefore, has a large number of entries. Together the two queues give rise to a second order system as verified by the above experiment.

Evaluation of the Closed-Loop Server

We use the digital Root-Locus method to design a PI controller for the difference equation model. The PI controller guarantees stability, zero steady-state error, and has a settling time of 4.5 min. The feedback control loop is implemented by modifying the Apache server software. The detailed system identification, design and implementation is described in [16].

Due to the highly unpredictable Internet client access patterns, it is critical for a Web server to achieve robust relative delay guarantees in the face of disturbances caused by changing workloads. We run experiments on the Linux PC testbed to compare the performance of the open-loop server with the closed-loop server with changing client populations. Both the open-loop and the closed-loop servers are tested with the same workload. Each experiment starts with the nominal workload generated by 200 basic (class-1) clients and 100 premium (class-0) clients. To test the servers' disturbance rejection capabilities, the number of premium clients is suddenly increased from 100 to 200 at 870 sec.

The open-loop server has a fixed process budget ratio that is hand-tuned to 0.83 through extensive system profiling based on a nominal workload. The process budget ratio of the closed-loop server is arbitrarily initialized to 1 without hand-tuning. The reference to the controller is set to 3, which requires the connection delay of the basic clients to be three times of that of the premium clients.

The performance of the open-loop server shown in Figure 4-a. The open-loop server performs well in the beginning of the experiment because its process budget ratio has been fine-tuned for the nominal workload. However, after the number of premium clients suddenly increases to 200 at 870 sec, the connection delay of the premium clients increases significantly. Consequently the connection delay ratio drops from the reference and remains below 1 (i.e,, the basic clients receive a shorter delay than the premium clients!) in the rest of the experiment. This result shows that the open-loop server cannot reject disturbances caused by a changing workload.

The performance of the closed-loop server is shown in Figure 4-b. In the beginning of the experiment, the connection delay ratio of the closed-loop server converges to around the reference by adjusting its process budget ratio. In response to the population change at 870 sec, the feedback control loop allocates more processes to the premium clients while reducing the number processes from the basic clients. By 1140 sec the connection delay ratio settles around the reference and stays close to it in the rest of the run. This result demonstrates that the closed-loop server is able to reject the disturbance caused by instantaneous changes in client populations. The recovery time is close to the designed settling time of 4.5 min. The server remains stable throughout the run, and the connection delay ratio remains close to the reference at steady state.

In summary, we have shown the effectiveness of feedback control theory in achieving QoS guarantees in a representative Internet server. We successfully developed effective software sensors and actuators to instantiate the relative guarantee control template. We have shown that the dynamics of a complex Internet server can be modeled through system identification. The implementation and evaluation on real server systems demonstrate that feedback control loops can provide robust QoS guarantees and disturbance rejection capabilities in Internet servers.

Other Applications in Computing Systems

Feedback control theory has been successfully applied to a wide spectrum of computing systems. The application domains cover different resources that need to be controlled (network bandwidth, CPU cycles, storage spaces, and I/O bandwidth), as well as different metrics and types of QoS guarantees. We now briefly summarize some recent application examples, providing references for more detailed information.

Internet Servers

In addition to the Web example presented earlier several recent papers [1–3] presented a control theoretical approach to web server resource management through Web content adaptation. Feedback control loops were developed to achieve absolute convergence guarantees, capacity reservation, and prioritization on request rate and delivered bandwidth for different service classes. In [9], MIMO control was designed to achieve absolute convergence guarantees on both memory and CPU utilizations in web servers. In other work, feedback control and fuzzy control were applied to control the input queue length in Lotus e-mail servers with admission control [10, 23].

CPU Scheduling

Feedback control theory was used for CPU scheduling in real-time systems such as multimedia and embedded control systems. Steere et. al. [28] developed a feedback based CPU scheduler that coordinates the CPU cycles allocated to the consumer and supplier threads to guarantee the fill level of buffers. In [18, 19], feedback control real-time scheduling algorithms were developed to provide deadline miss ratio guarantees for real-time applications with unknown task execution times. Feedback control real-time scheduling has also been extended to handle distributed systems

[27]. All the above scheduling algorithms are designed for absolute convergence guarantees.

Storage Management

Storage is another critical resource in many server systems. In [21], a relative guarantee template was developed to provide a relative hit ratio guarantee in a Web proxy cache through a resource reallocation actuator that dynamically changes the disk space allocation for different service classes. Recently adaptive control was applied to the same the Web proxy cache to improve its portability and robustness via automatic controller tuning [20].

The Aqueduct system [17] featured a feedback control loop that regulated the speed of background data migration in enterprise storage servers while bounding the performance impact on front-end applications. Aqueduct provided absolute convergence guarantees on the I/O latency of front-end applications during data migration.

Network Routers

At the network layer, control theory was applied to packet flow control in Internet routers. Hollot et. al. [14] applied control theory to analyze the RED active queue management on IP routers. Their control analysis provided guidance for tuning the RED algorithm, which had previously been a difficult problem. Recently, a feedback-based algorithm was developed for quantitative assured forwarding services [7] to provide absolute and proportional differentiation of loss, service rates, and packet delays on Internet routers. Li and Nahrstedt [15] developed a hierarchical architecture that integrates an upper-layer fuzzy control and a lower-layer packet rate control to achieve absolute convergence guarantees on tracking precision in distributed visual tracking systems.

Microprocessor Architecture

Feedback control theory has also been applied in microprocessor architectures. In [26], absolute convergence guarantees were given on CPU chip temperatures by applying control-theoretical techniques to microprocessor thermal management.

Middleware for QoS Control

As shown above, over the last several years an increasing number of individual feedback control solutions have been developed for various software performance assurance problems. A natural question is whether any generalization of these solutions is possible. Indeed it seems that a small set of QoS guarantee types cover a wide range of applications, and for each of these classes of guarantees, we can define control theoretic templates that are amenable to on-line solutions. In computer systems when many applications can make use of similar services, these services are often provided in terms of middleware. In our work, we have developed a middleware called ControlWare [30] that supports multiple types of QoS guarantees, each based on feedback control.

ControlWare

ControlWare is a middleware QoS-control architecture based on control theory, motivated by the needs of performance-assured Internet services. ControlWare allows the user to express QoS specifications off-line, maps these specifications into appropriate feedback control loop sets, tunes loop controllers analytically to guarantee convergence to specifications, and connects loops to the right performance sensors and actuators in the application such that the desired QoS is achieved. One main novelty of our middleware lies in isolating the software application programmer from control-theoretic concerns while utilizing this theory to achieve the desired QoS guarantees. At the same time, ControlWare isolates the control engineer from the software task of interfacing the controller to the controlled software system and designing software performance sensors and actuators. A Conceptual representation of ControlWare components used in the process of designing software performance-assurance loops is shown in Figure 5-a.

ControlWare contains a library of macros written in our topology description language, each formulating a particular type of QoS guarantees as a feedback control problem. The library is extensible in that a control engineer can transform a new guarantee type into a macro that describes the corresponding loop interconnection topology and stores that macro in the middleware's library. Currently, the library includes macros for absolute convergence guarantees, relative differentiated service guarantees, prioritization, and optimization guarantees. Each macro, like a block diagram, includes components such as sensors, actuators, and controllers. ControlWare contains a library of common sensors and actuators that can be used in these software control loops. The library is extensible in that it is possible for an application programmer to add new sensor and actuator types.

Overall, ControlWare can express many common guarantee types required in performance-assurance software by casting them appropriately as feedback control problems. Once the control loops are instantiated from the QoS specification, the middleware uses textbook techniques to estimate system models and determine appropriate feedback controller parameters for guaranteed convergence of the control loops to the specified performance.

SoftBus: The ControlWare Backbone

A number of implementation challenges had to be overcome to create ControlWare, including interoperability. To promote interoperability, the engineering community standardized open layered interface architectures such as the Fieldbus [6], which greatly simplify the interconnection of sensors, actuators, and controllers in a digital control system. Similarly, a crucial step in developing an open middleware layer for software QoS control is to provide a similar generic Application Programming Interface (API) and communication backbone to interface the computing and control subsystems. This backbone must be appropriate for distributed software rather than a physical bus. We call this backbone, a *SoftBus*.

ControlWare implements a SoftBus which provides a common interface for efficient information exchange between software performance sensors, actuators and controllers across machines and address spaces. The sensors, actuators and controllers are viewed as interchangeable plug-in modules. Note that, these modules are not physical devices (like a temperature sensor), but rather software components which conceptually act like a sensor or actuator (e.g., a software load sensor might invoke an operating system call to measure CPU utilization). Modules connected to SoftBus need not know each other's locations and need not worry about distributed communication. Underneath the common API, different information exchange mechanisms are developed for different situations. This layered architecture is depicted in Figure 5-b.

Plug-in Sensors and Actuators

In Softbus we support two types of software sensors and actuators: *passive* and *active*. A *passive* sensor or actuator is just a function or software component that returns sample data or accepts a command when called by the controller. An *active* sensor or actuator, in contrast, is a process or thread which may be running in its own address space. It is usually awakened periodically by the operating system scheduler to perform sensing or actuation. For example, an idle CPU-time sensor may be implemented as an active sensor process which runs at the lowest priority and computes the percentage of time it has been executing. If a controller needs to communicate with such a sensor, some kind of IPC instead of a direct function call should be used. Controllers are designed completely independently of the identity of the sensors and actuators. From the controller's perspective, the software system being controlled acts like a regular physical plant.

Sensors typically amount to a modest instrumentation of application code. For example, a sensor measuring the request rate on a particular site can be implemented as a simple counter. A sensor measuring delay can be implemented as a moving average of the difference between two timestamps. Often the measured metric is already available as a variable maintained by the controlled software service (e.g., queue length) or the operating system (such as CPU utilization). All one needs to do to implement the sensor is to pass the value to the middleware. The main challenge, however, is the design of the actuator. To meet this challenge, our middleware includes a generic resource manager (GRM) that can be thought of as an all-purpose actuator, or a "generic control valve". The actuator interfaces to any resource queue (i.e., "pipe") such that resource allocation (i.e., "flow") can be controlled in a simple, unified, yet customizable manner. In the following we shed some light on the generic resource manager in ControlWare.

Generic Resource Manager

Our generic resource manager (GRM) is designed for use with Internet servers such as web servers, DNS servers, mail servers, and proxy cache servers. As mentioned earlier, in these applications, it is the order and quantity of resource consumption that decide the quality of the service. Therefore, the actuator must control the access to the resources quantitatively. The actuator uses quota, which is one of the knobs it exports, to represent the resource allocated to each class. It can change the QoS of different classes by changing the quota dynamically at runtime. The definition of "quota" depends on the type of resource under consideration. Generally, server resources can be categorized into two categories according to their access pattern:

- Time-multiplexed resources. Such resources as CPU cannot be accessed by multiple users at the same time. Instead, access has to be serialized and multiplexed in time. For such resources, the GRM maintains per-class queues. A classifier classifies the requests and puts them into the different queues. The quota refers to the rate of dequeue of each class as a fraction of server capacity.

- Space-multiplexed resource. Memory and disk space are examples of such resource. They can be used by multiple users at the same time. Typically, each class has a quota indicating the amount of the resource dedicated to it. A queue may or may not be necessary in this scenario. In many cases, requests are admitted until the quota is used up at which point further requests are simply rejected. An interesting alternative is that when some class's quota is used up and a new request from this class arrives, a previously allocated resource amount can be revoked from other clients and transferred to this new request. Cache management is an example of this phenomenon, where newly cached pages replace older pages in the cache.

In summary, the generic resource manager understands the notion of *traffic classes*, and exports the abstraction of *resource quota* to represent the amount of logical resources allocated to a particular class. The action of the manager lies in controlling resource quota allocations.

The structure of the generic resource manager is shown in Figure 6-a. In the figure, the Classifier and Resource Allocator are provided by the application. The Resource Allocator does resource allocation. The Queue Manager maintains one queue for each class, governed by a certain queuing policy. The Quota Manager maintains a resource quota for each class.

To use GRM, the application must export three interfaces: *allocProc*, *rejectProc* and *revokeProc*. *allocProc* is called to execute application-specific resource allocation methods. When a previously arrived request has to be rejected, *rejectProc* is called to do clean up work (for example, close the network connection). *revokeProc* is called when a resource previously allocated to some clients needs to be revoked (such as the case with cache replacement policies).

Figure 6-b summarizes the interaction between GRM and the application. When some resource is requested by the application, the request is first classified by the Classifier. After that, the request is passed to GRM by calling *insertRequest*. GRM controls resource allocation by checking the request against two constraints: (i) the queue length, and (ii) the quota constraint. If the queue for the given class is empty and the class has quota, the request is satisfied immediately via the function call *allocProc* to the resource allocator, and the quota is updated accordingly. If the request can't be satisfied immediately, it will be buffered in the its queue. When some resource becomes available, the application calls *resourceAvailable* to notify GRM, which will try to satisfy as many pending requests as possible.

It is important to mention that quota is a purely logical concept. Unlike the traditional resource reservation system, in our middleware the mapping of quota to physical resource consumption need not be known. In effect, the GRM is a logical queuing, admission control, and resource allocation policy interface with a back-end that is capable of executing a primitive service function such as assigning a request to a service process. The GRM generalizes the expression of various resource allocation policies in a common framework and makes it possible to control logical quota allocations by simple feedback controllers such that performance constrains are met. The control loop is guaranteed to converge because of the way controllers are designed which is the advantage of using a control-theoretic approach. Most importantly, the physical mapping of quota to actual resource consumption need not be known for correct operation, which separates this approach from resource reservation systems.

In summary, ControlWare is shown to be a very novel middleware that supports QoS guarantees based on control theory. While the middleware exists there are many open questions and improvements possible. For example, control templates for distributed systems, for adaptive controllers, and when faults exist are some of the interesting questions that must be answered and then added to the ControlWare libraries.

Conclusions

In this paper, we presented the necessary foundations for using a control-theoretic framework to achieve QoS guarantees in software systems. With the growing popularity of the Internet, providing performance guarantees in open software systems has become increasingly important. We illustrated the successful application of the control framework by practical examples in which guarantees were provided within a software service. We analyzed the general architecture of software servers for purposes of feedback control. We have shown that high-performance servers can be approximated in practice by a liquid task model. The model gives rise to a fluid representation of workload in which the server is modeled as a cascaded flow through a pipeline of tanks of different capacities. We identified typical actuators that control these flows within the server and illustrated software protocols that implement these actuators. We also presented a taxonomy of the most important QoS assurance problems in software literature and described how they can be translated into a feedback control formulation. Having shown the feasibility of a control-theoretic formulation of software assurance problems, the feasibility of linear modeling, and the feasibility of actuation in practical application scenarios, we believe that the foundation is now in place towards applying the wealth of control-theoretic results in the Internet application domain.

There are many remaining issues and challenges that warrant further research. For example, how to model non-linearities peculiar to computing systems when they cannot be adequately handled by the linearization techniques we employed? How can these nonlinearities be accounted for in controller tuning? The most widely-spread nonlinearity in software systems is the inverse relation between rate and delay that complicates the analysis of delay control loops; a key step towards reasoning about and controlling temporal behavior. While, in this paper, we focused on fixed-parameter controllers, of significant interest is the possible application of adaptive control and robust control techniques to handle parameter variations and load uncertainties. Another interesting possibility is the use of a predictive control framework, or one where queueing-theoretic prediction is integrated with feedback-based correction to achieve the desired software performance. Finally, further examples, theoretical foundations, experimental evidence, and practical experience are needed to establish the true potential and limitations of applying feedback performance control to different computing systems. This remains an important focus of our future research.

References

[1] T. F. Abdelzaher. An automated profiling subsystem for qos-aware services. In *IEEE Real-Time Technology and Applications Symposium*, Washington, D.C., June 2000.

[2] T. F. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service*, London, UK, June 1999.

[3] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, January 2002.

[4] C. Aurrecoechea, A. Cambell, and L. Hauw. A survey of QoS architectures. In *4th IFIP International Conference on Quality of Service*, Paris, France, March 1996.

[5] P. Barford and M. E. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of Performance '98/ACM SIGMETRICS '98*, pages 151–160, Madison, WI, 1998.

[6] A. Chatha. Fieldbus: the foundation for field control systems. *ontrol Engineering*, 41(6):47–50, May 1994.

[7] N. Christin, J. Liebeherr, and T. F. Abdelzaher. A quantitative assured forwarding service. In *IEEE Infocom*, NEW YORK, NY, June 2002.

[8] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):55–79, December 1997.

[9] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury. Mimo control of an apache web server: Modeling and controller design. In *American Control Conference*, Anchorage, Alaska, May 2002.

[10] Y. Diao, J. L. Hellerstein, and S. Parekh. Using fuzzy control to maximize profits in service level management. *IBM Systems Journal*, 2002.

[11] C. Dovrolis and P. Ramanathan. Proportional differentiated services, part ii: Loss rate differentiation and packet dropping. In *International Workshop on Quality of Service*, Pittsburgh, PA, June 2000.

[12] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional differentiated services: Delay differentiation and packet scheduling. In *SIGCOMM*, pages 109–120, 1999.

[13] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[14] C. V. Hollot, V. Misra, D. Towsley, and W. Gong. A control theoretic analysis of red. In *IEEE Infocom*, Anchorage, Alaska, April 2001.

[15] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications*, September 1999.

[16] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *IEEE Real-Time Technology and Applications Symposium*, TaiPei, Taiwan, June 2001.

[17] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *USENIX Conference on File and Storage Technologies*, Monterey, CA, January 2002.

[18] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. The design and evaluation of a feedback edf control scheduling algorithm. In *IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1999.

[19] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Journal of Real-Time Systems*, 23(1/2), May 2002.

[20] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework and its application to differentiated caching services. In *International Conference on Quality of Service*, Miami Beach, FL, May 2002.

[21] Y. Lu, A. Sexana, and T. Abdelzaher. Differentiated caching services; a control-thoeretical approach. In *Proceedings of the 2001 International Conference on Distributed Computing Systems*, pages 615–622, 2001.

[22] W. A. McEachern. *Economics*. South-Western College Publishing, 5th edition, 1999.

[23] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[24] V. Paxton and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[25] T. G. Robertazzi. *Computer Networks and Systems: Queuing Theory and Performance Evaluation*. Springer Verlag, 3rd edition, 2000.

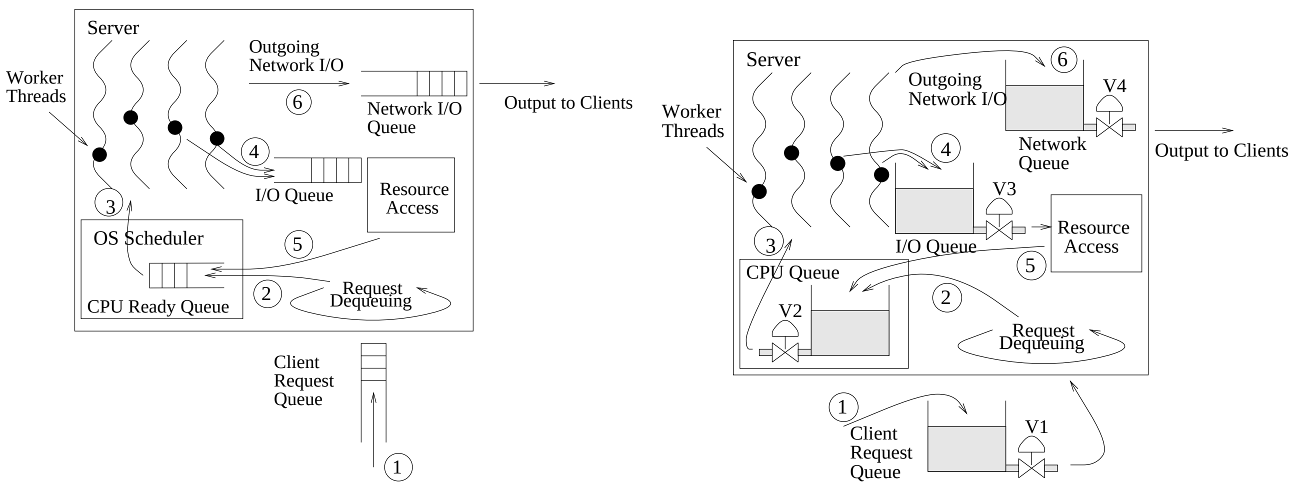
[26] K. Skadron, T. Abdelzaher, and M. Stan. Control-theoretic techniques and thermal rc modeling for accurate and localized dynamic thermal mangement. In *International Symposium on High Performance Computer Architecture*, Cambridge, MA, February 2002.

[27] J. A. Stankovic, T. He, T. F. Abdelzaher, M. Marley, G. Tao, S. H. Son, and C. Lu. Feedback control scheduling in distributed systems. In *IEEE Real-Time Systems Symposium*, London, UK, December 2001.

[28] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation*, pages 145–158, 1999.

[29] K. J. strm and T. Hgglund. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, 2nd edition edition, 1995.

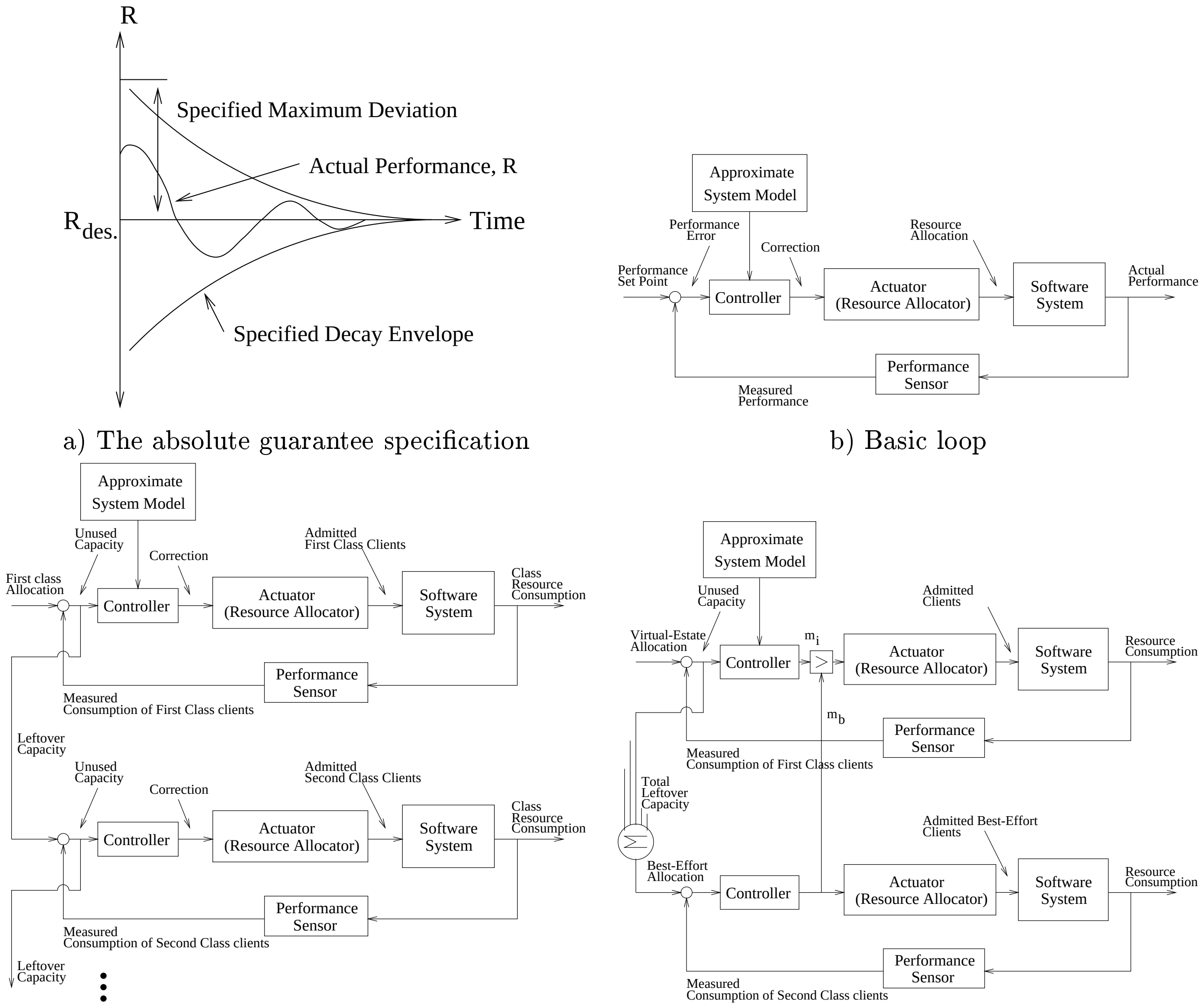
[30] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *Proceedings of the 2002 International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.



a) The computing model

b) a control-oriented representation

Figure 1: Server architecture



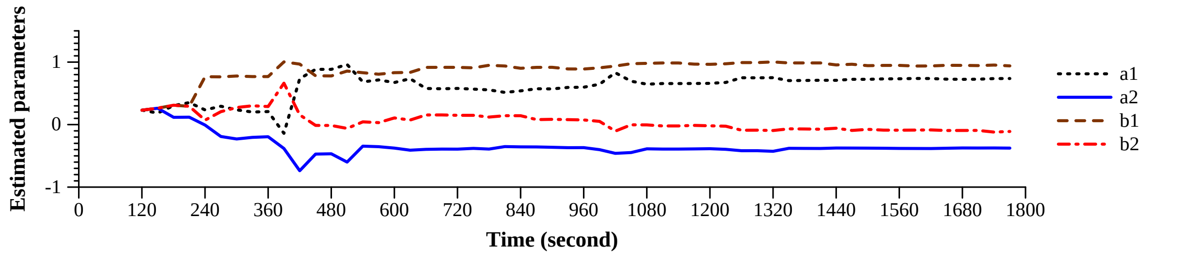
a) The absolute guarantee specification

b) Basic loop

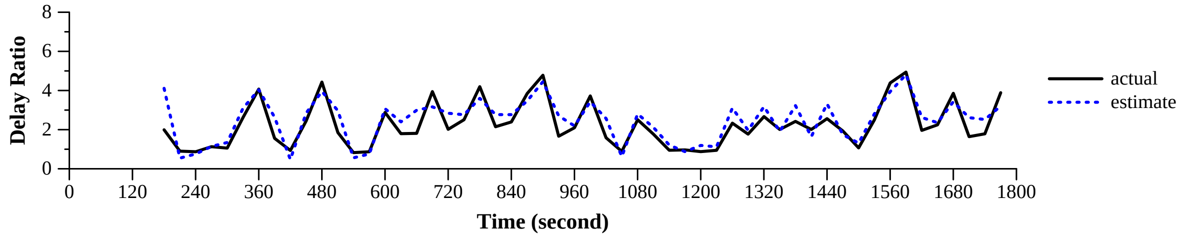
c) Prioritization

d) Excess capacity management

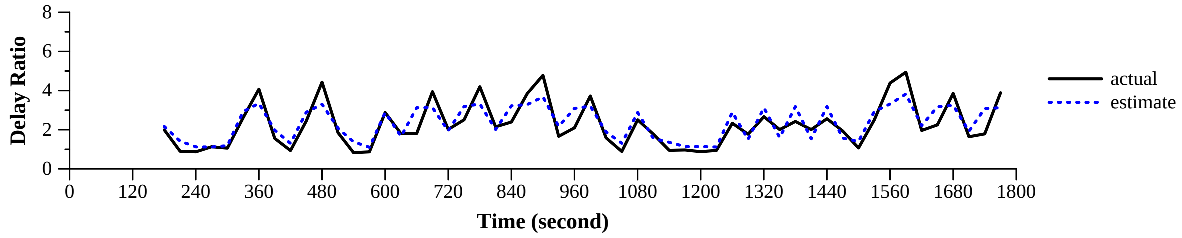
Figure 2: Control loop templates



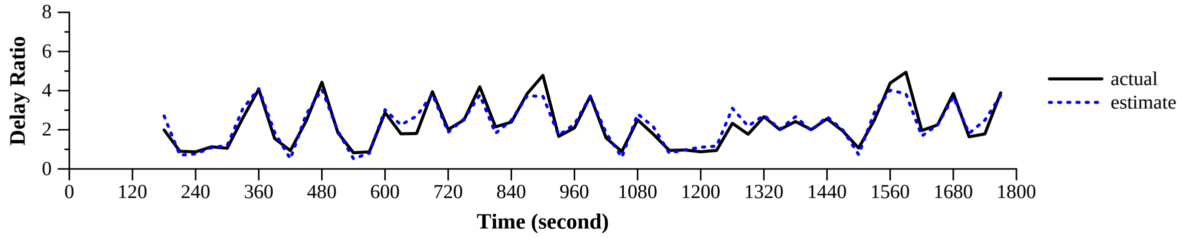
(a) Estimated model parameters (second order model)



(b) Modeling error (second-order model)

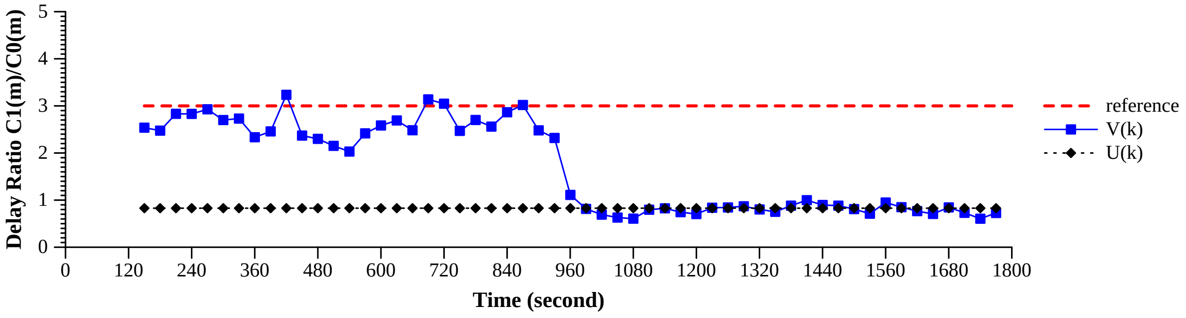


(c) Modeling error (first-order model)

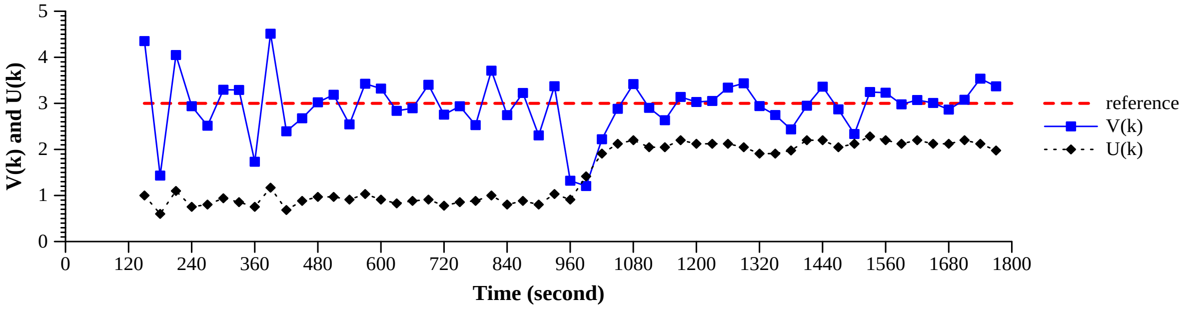


(d) Modeling error (third-order model)

Figure 3: System identification results of an Apache server

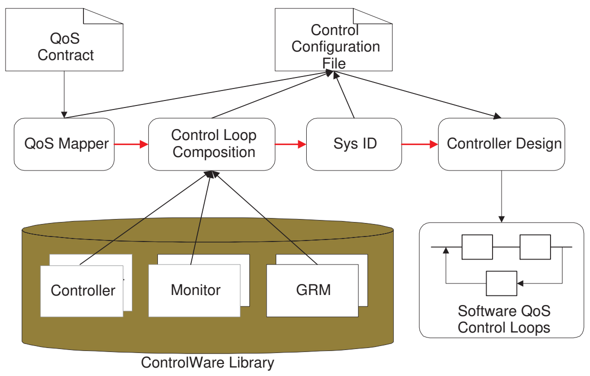


(a) Open-loop server

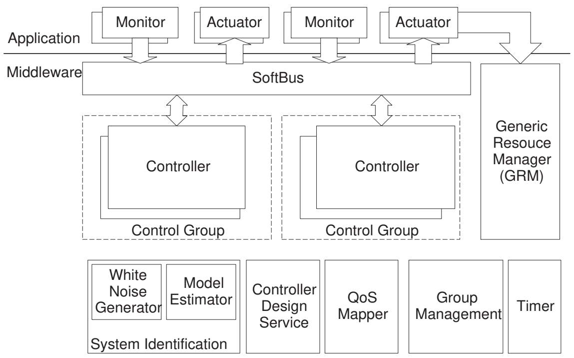


(b) Closed-loop server

Figure 4: Compare the performance of the Open-loop and Closed-loop servers: connection delay ratio $V(k) = C_1(k)/C_0(k)$; process budget ratio $U(k) = B_0(k)/B_1(k)$

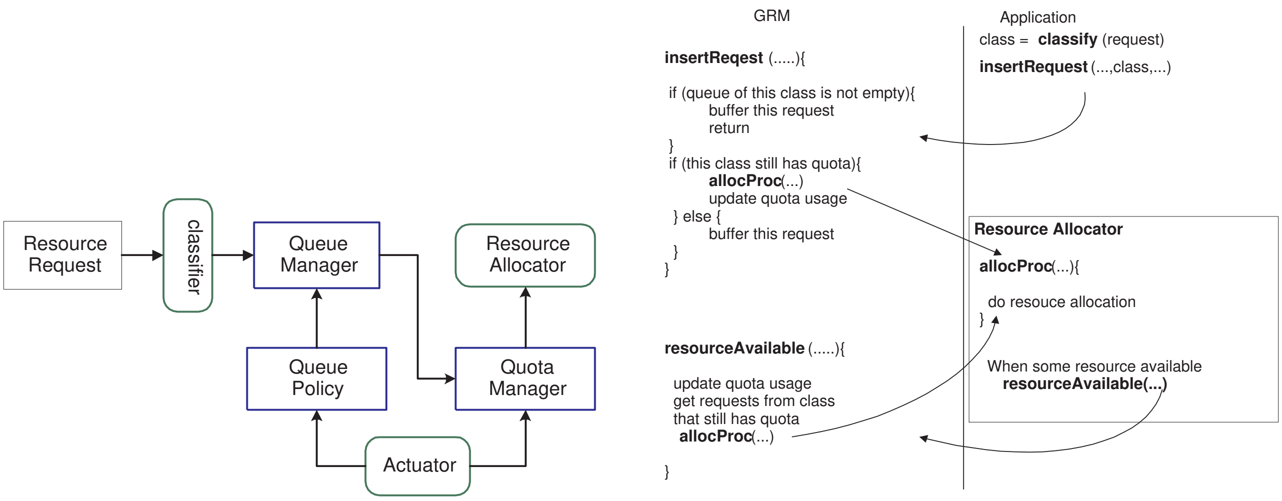


a) Development methodology



b) Interfacing to the SoftBus

Figure 5: ControlWare



a) Structure of GRM b) Resource allocation procedure

Figure 6: Generic resource manager