ControlWare: A Middleware Architecture for Feedback Control of Software Performance *

Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, John A. Stankovic
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
e-mail: *rz5b, chenyang, zaher, stankovic@cs.virginia.edu*

Abstract

Attainment of software performance assurances in open, largely unpredictable environments has recently become an important focus for real-time research. Unlike closed embedded systems, many contemporary distributed real-time applications operate in environments where offered load and available resources suffer considerable random fluctuations, thereby complicating the performance assurance problem. Feedback control theory has recently been identified as a promising analytic foundation for controlling performance of such unpredictable, poorly modeled software systems, the same way other engineering disciplines have used this theory for physical process control.

In this paper, we describe the design and implementation of ControlWare, a middleware QoS-control architecture based on control theory, motivated by the needs of performance-assured Internet services. It offers a new type of guarantees we call convergence guarantees *that lie between hard and probabilistic guarantees. The efficacy of the architecture in achieving its QoS goals under realistic load conditions is demonstrated in the context of web server and proxy QoS management.*

1 Introduction

Applications of real-time computing have gradually evolved from closed embedded systems to complex, distributed open platforms operating in unpredictable poorly modeled environments such as the Internet. Hard guarantees are impractical on such platforms since load and resource capacity are very difficult to predict. Yet, many modern applications require some form of performance assurances which may include guarantees on timeliness, bandwidth, data consistency, or jitter. Traditional approaches for providing these performance assurances, such as resource reservation [23] and *a priori* knowledge of worst case execution conditions [27], are no longer applicable.

To achieve predictable behavior in distributed, poorly modeled, uncertain environments of today's open performance-assured applications, several recent research efforts have suggested the use of control theory [16, 30, 21, 18]. This theory offers a new types of guarantees that lies between hard and average (e.g., probabilistic), which we call *convergence guarantees* [19]. A basic convergence guarantee states that, upon any perturbation, the performance variable of choice will converge to its desired value within a specified bounded time and that its deviation from that value is always bounded. Control theory employs difference equations as a fundamental modeling tool and has proven to be very successful in the engineering community in controlling a vast variety of nonlinear, stochastic, and time-varying physical systems. The success of control theory is, in large part, due to its robustness in the face of modeling errors and external disturbances, which reduces the need for accurate system models — a much welcome property when accurate models are difficult to construct. Recent results have shown that control theory can also be successfully applied to the control of software performance [19]. Intuitively, many software performance attributes are affected by accumulative factors such as queues building up on key system resources. Queues can be represented by difference equations relating flow to fill level, hence giving rise to difference-equation models analyzable in a control-theoretic framework.

The authors have applied control theory successfully in several example case studies involving computing applications. These case studies include performance isolation in web servers [5], web server delay control [18], proxy cache relative hit ratio control [21], network-layer active queue management for delay and loss differentiation [10], and microprocessor thermal management [29]. In this paper, we leverage the underlying insights to develop a middleware layer for QoS control that provides control-theoretic performance guarantees under uncertainty. The middleware is specifically targeted for Internet services. It allows the user

*The work reported in this paper was supported in part by the National Science Foundation under grants CCR-0093144 and CCR-0098269.

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 the middleware lies in isolating the application programmer from control-theoretic concerns while utilizing this theory to achieve the desired QoS guarantees.

The rest of the paper is organized as follows. Section 2 introduces the control theoretical approach in more detail and explains how QoS specifications are mapped into control loops. Section 3 presents the middleware architecture. Section 4 describes the resource management component. An evaluation of our QoS control functionality using the implemented middleware prototype in a web application scenario is presented in Section 5. Section 6 presents related work. The paper concludes with Section 7.

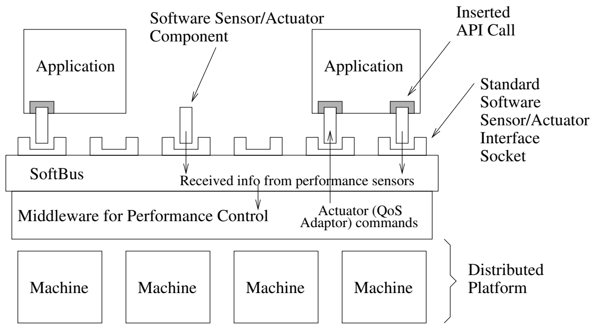

Figure 1. Overall Architecture

2 Middleware-Based QoS Control

A control-theoretic paradigm for QoS control offers performance guarantees on transient behavior such as bounded deviation, and convergence speed, which is a step between hard guarantees possible mostly in closed predictable environments, and soft statistical guarantees that describe only average behavior. The use of feedback-control theory to provide convergence assurances in open distributed QoS-sensitive applications, such as mail servers, web servers, and proxy caches, involves solving three main design challenges. First, from a control-theory perspective, a general methodology needs to be developed for converting QoS specifications of a computing system such as an Internet server, into feedback loops with known set points and feedback control parameters. This is achieved via multiple software tools and libraries that we describe in this section. Second, from a systems perspective, a convenient interface needs to be found between the service software and the middleware control loops that manage its performance. In our system, this interface is implemented by an entity called a *SoftBus*. The SoftBus, described in Section 3, is a distributed protocol that runs across multiple machines and address spaces forming a virtual application backbone into which servers, performance sensors, actuators, and controllers can be easily plugged-in. Third, appropriate software performance sensors and actuators must be designed. We elaborate on this challenge in Section 4. Figure 1 shows an overall picture of the middleware components.

2.1 Service Development with ControlWare

ControlWare is a middleware incarnation of our control-theoretic paradigm for software QoS management. To illustrate its main features, we first overview the development methodology of ControlWare-based performance-assured software. An application designer using our middleware for QoS provisioning would typically undergo the process shown in Figure 2:

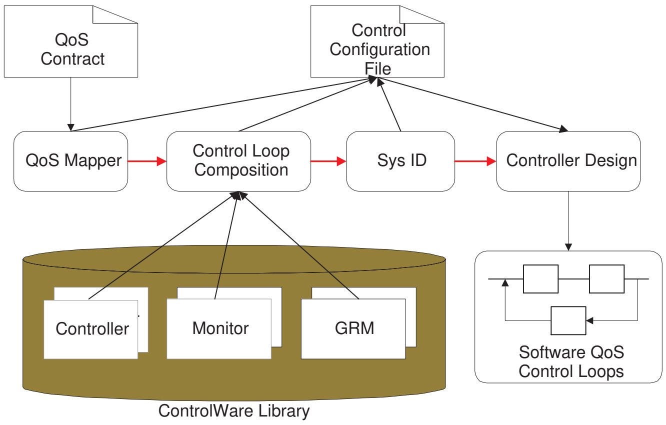

Figure 2. Development Methodology

- QoS specification: The required QoS guarantees should be specified for the system. ControlWare supports multiple types of QoS guarantees including absolute, relative, and optimization-based. The developer will describe the required guarantees with a simple Contract Description Language (CDL). A partial syntax of CDL is described in Appendix A. In CDL each guarantee is described by its type and desired QoS level. For example, a relative delay guarantee can be defined as follows:

```
GUARANTEE RELATIVE_DELAY {
GUARANTEE_TYPE = RELATIVE;
CLASS_0 = 1;
CLASS_1 = 2;
CLASS_2 = 3; };
```

The above guarantee states that the QoS (e.g., service delays of a Web server) of three service classes should follow a fixed ratio 1:2:3.

- QoS to control-loop mapping: A tool called the *QoS mapper* interprets the CDL description offline and maps the required QoS guarantees to a set of feedback control loops and their set points. For example, the above relative delay guarantee is mapped into multiple feedback control loops controlling the relative delay ratio of different classes. The QoS mapper specifies the feedback control loops using a *topology description language* and stores it in a configuration file.
- Control loop composition: The *loop composer* configures QoS monitors (also called sensors), actuators, and controllers in the manner described by the topology description language. The middleware includes a component library of sensors and actuators that can be used in the control loops. It can also accept user defined components.
- System identification: The middleware determines an approximate difference equation model for the controlled system. The difference equation model is a prerequisite for controller design in a control-theoretic approach to QoS guarantees. It allows the desired QoS guarantee to be achieved without laborious hand tuning and testing. ControlWare provides a system identification service that automatically derives difference equation models based on system performance traces using a least-squares estimator [7]. The success of system identification of software systems has been clearly demonstrated in several prior publications [13, 18, 1]. System identification can be performed off-line or on-line during normal operation and may be repeated periodically or after each software or hardware upgrade.
- Controller configuration and tuning: Based on the model derived by system identification, ControlWare's controller design service can automatically tune the controllers to guarantee stability and desired transient response to load variations [19]. The tuning process is based on established control theory methods such as gain margin or state-based design. The controller tuning process can be invoked off-line or on-line (the control scheme is called adaptive control when it is tuned on-line.) The resultant controller parameters are written into a configuration file.

The above process completes system configuration. The application linked with ControlWare will subsequently satisfy the QoS assurances specified in the first step of the configuration process. The process is somewhat similar to the way control engineers configure a distributed physical process control system. What is new here is that the controlled system is a software service, and the control goal is to provide convergence guarantees on QoS. With ControlWare software engineers can easily add performance assurances to their systems without the need for a control-engineer's background. Our middleware automates that part of the feedback loop configuration process.

2.2 QoS Mapping

The cornerstone of a control theoretic paradigm for QoS guarantees in software systems lies in our ability to convert common resource management and software performance assurance problems into feedback control problems. Our middleware 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 extendible in that a control engineer can transform a new guarantee type into a macro that describes the corresponding loop interconnection topology and store that macro in the middleware's library. Currently the library includes macros for absolute convergence guarantees, performance isolation, statistical multiplexing, prioritization, relative differentiated service guarantees, and optimization guarantees. As an example, we describe the implementation of the basic (absolute) convergence guarantee, and its use in formulating relative guarantees, prioritization, and optimization as feedback control problems.

2.2.1 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 $R_{desired} - R$ be bounded at all times, as shown in Figure 3.

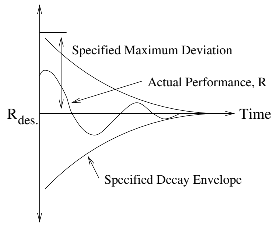

Figure 3. The Absolute Guarantee Specification

The problem requires that R be *measurable* (such as

CPU utilization, server throughput, or service queuing delay). We call a software probe that measures the value of the performance metric R, a sensor $S(R)$. We do not assume perfect sensors, but require that in the steady state (i.e., when the measured performance R does not change) the sensor output $S(R)$ should asymptotically approach R.

We further require that R be *controllable*. In other words, the application must have some adaptation mechanism, $A(R)$ that affects the value of R. We call this mechanism, actuator. For example, if R is CPU utilization, $A(R)$ can be an admission control mechanism. Our API requires that the adaptation mechanism have a single input "knob" with an abstract range from 0 to 100%. The actuator, $A(R)$, interprets the position of its input knob, denoted m, in some application-specific way (such as a probability of admitting the next client) and executes accordingly. We require that R be a monotonic function of m.

The absolute convergence guarantee is translated into the control loop shown in Figure 4. The loop samples the measured performance, $S(R)$, compares it to the desired value $R_{desired}$, and uses the difference to induce changes in resource allocation via the actuator $A(R)$. A critical part of this loop is the controller. Control-theory offers techniques for designing the controller such that performance guarantees are obtained on the speed of convergence of R to $R_{desired}$, and on the maximum deviation $R_{desired} - R$ in the presence of transient disturbances [28].

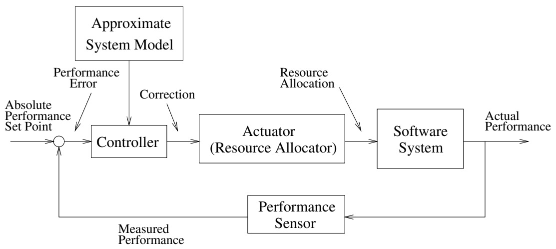

Figure 4. Absolute Guarantees

The absolute convergence guarantee loop is the elementary building block of our middleware from which all other assurances follow, as described below.

2.2.2 The Relative Guarantee Problem

In a relative differentiated services framework it is desired to keep the ratio between the performance levels (such as delay, throughput, etc) of two traffic classes fixed. This fixed value is a good candidate for the performance set point, R. In general, let there be n content (i.e., traffic) classes in the system. Let the measured performance metric of class i be H_i. The differentiation policy specifies that the performance metrics 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 a proportionality constant or weight of class i. We define the *relative* performance, R_i, of class i to be $R_i = H_i/(H_1 + H_2 + ... + H_n)$. It determines how the class is performing relative to other classes. The desired relative performance of class i should be $R_{i_{desired}} = C_i/(C_1 + C_2 + ... + C_n)$. We call the difference $R_{i_{desired}} - R_i$ the performance error e_i of class i. An appealing property of this model is that the aggregate performance error of the system is always zero, because $\sum_{1 \leq i \leq n} e_i = \sum_{1 \leq i \leq n} (R_{i_{desired}} - R_i) = \frac{\sum_{1 \leq i \leq n} C_i}{C_1 + C_2 + ... + C_n} - \frac{\sum_{1 \leq i \leq n} H_i}{H_1 + H_2 + ... + H_n} = 1 - 1 = 0$. Hence, for any linear function of error $f(e_i)$ the sum $\sum_{1 \leq i \leq n} f(e_i) = 0$. In particular, since the controller is a linear function of error, the sum of controller outputs (i.e., corrections in resource allocation) of all classes is zero. Hence, the feedback loops can operate independently with one loop per class, while the total amount of allocated resource remains constant. Individual loops provide a convergence guarantee on the *relative* performance of a particular class. The control loop configuration for some arbitrary class i is shown in figure 5.

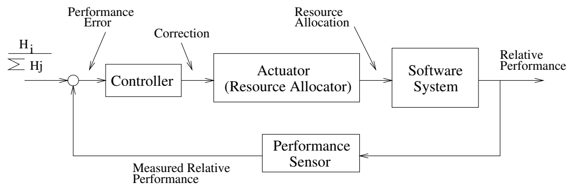

Figure 5. Relative Differentiated Service

2.2.3 The Prioritization Problem

The prioritization problem is defined as one where all service clients are partitioned into n classes, such that for every class i, it is desired that clients of that class do not suffer any contention over some shared resource r from any clients of classes $i + 1, ..., n$.

We implement these semantics by a composition of the elementary block described in Section 2.2.1. First, we make the entire server capacity available to the highest priority class using the basic convergence guarantee loop of Figure 4 with a set point equal to total server capacity. If that set point is not reached (because there is not enough demand), the unused capacity of each class is measured and treated as the set point for the resource allocation to the lower priority class. This requires a sensor array such that $S(R_i)$ measures the fraction of resource r consumed by clients of class i, as well as an actuator array where $A(R_i)$ controls the resource allocation of class i. The arrays are implemented by a set of per class performance counters and admission control limits.

The feedback loop architecture for prioritization is described for a two class server in Figure 6. One control loop is needed per class. Application performance converges to that of a strictly prioritized system. The approach may be used to implement logical priorities in middleware when the controlled server itself does not support priorities by design, such as the Apache [12] web server. An example and evaluation of this use is presented in [3].

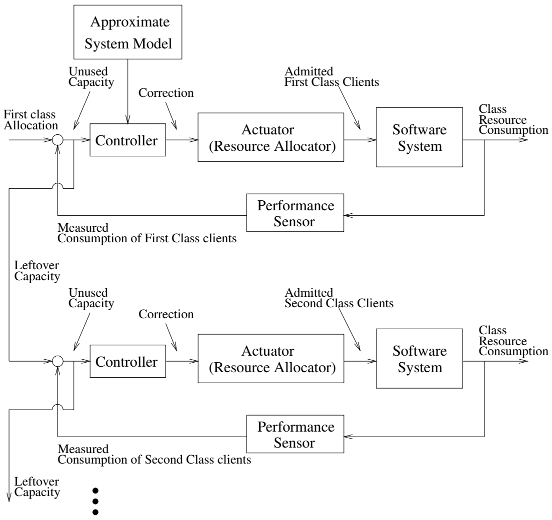

Figure 6. Prioritization

2.2.4 The Utility Optimization Problem

Another type of performance problems 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 approach is illustrated in Figure 7.

As shown above, ControlWare can express the most common guarantee types required in performance-assurance software by casting them appropriately as feedback control problems. Once the control loops are designed from the QoS specification, the middleware uses textbook techniques to estimate system models and determine appro-

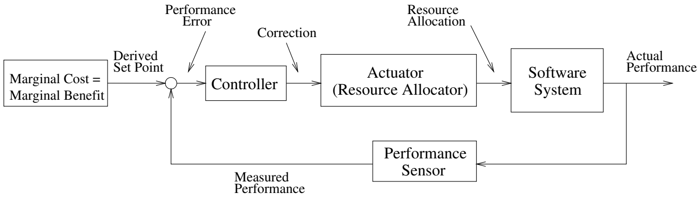

Figure 7. Utility Maximization

priate feedback controller parameters for guaranteed convergence of the control loops to the specified performance. In this paper, we do not discuss these textbook control techniques. A review of such techniques can be found in [28]. Instead, here, we shed light on implementing a general infrastructure for feedback loop composition focusing on the new challenges introduced when the controlled process is not a physical entity but a software service. These challenges are discussed next.

3 SoftBus: The Distributed Interface

To promote interoperability, the control engineering community standardized open layered interface architectures such as the Fieldbus [9] greatly simplifying the interconnection of sensors, actuators, and controllers in a digital control system. Hence, a crucial step towards developing an open middleware layer for software QoS control in software environments is to provide a similar generic API and communication backbone that is more appropriate when the controlled system is a distributed software entity. We call this backbone, a *SoftBus*.

ControlWare implements a SoftBus whose main purpose is to provide a common interface for efficient information exchange between software performance sensors, actuators and controllers across machines and address spaces. The sensors, actuators and controllers 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 8.

We support two types of software sensors and actuators: *passive* and *active*. A *passive* sensor or actuator is just a function call 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

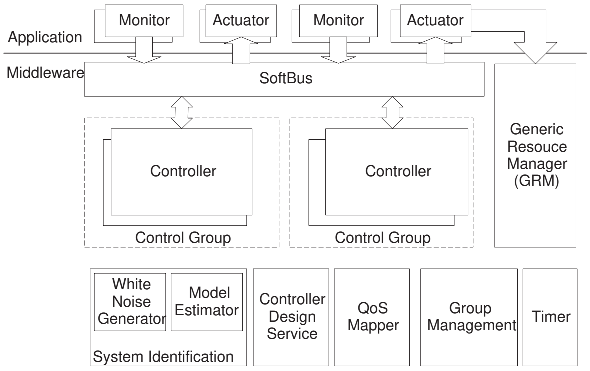

Figure 8. Interfacing to the SoftBus

call should be used.

3.1 Interface Modules

SoftBus provides a generic interface between sensors, actuators and controllers accommodating the nature of each component (passive versus active) and abstracting away the nature of communication (local versus remote). This is done by providing two types of interface modules. One for active components and one for passive. Each component is attached to the SoftBus via an appropriate interface module. These modules are defined in a library linked to each sensor or actuator component that is interfaced to the SoftBus. Communication with passive components occurs via direct function calls issued by the linked interface module and handled by the component. Communication with active components occurs when the component invokes the appropriate middleware call which accesses the module via shared memory. Internally, both types of modules may perform distributed communication transparently to the components, for example, if a controller on one machine sends data to an actuator on another. This is described in Section 3.4.

3.2 Registrar

Configurability is achieved through the registration and deregistration of control loop components. Registration API is exported by an entity, called the registrar. Internally, the registrar maintains a cache. For each local component, it records in the cache the component's type (sensor/actuator or controller), a callback function pointer if it is passive, or a shared memory slot index if it is active. For remote components, it will record their type and location. Originally, only local components have entries in the registrar's cache. When some component's information is needed but can't be found in the cache, the registrar contacts an external directory server and caches the received information. When caching information on remote components, the registrar also creates a daemon to wait for invalidation messages from the directory server. When it receives a message notifying it of the deregistration of some components from the directory server, the registrar will purge the corresponding entries from the cache accordingly.

3.3 Directory Server

The directory server maintains the location and properties of all control loop components. To maintain cache consistency, the directory server keeps track of all machines that cache its information and notifies them when data has changed. When all the components are on one machine, the directory server is no longer needed. In this case, SoftBus optimizes itself automatically by shutting down the unnecessary daemons, and inhibiting communication between the registrars and the directory server. In the present implementation, the number and identities of the machines which run SoftBus is stored in a static configuration file. It is reasonably straightforward to extend this architecture to allow new machines to subscribe to the SoftBus dynamically using a group membership service such as [25, 6, 2].

3.4 Data Agent

The data agent abstracts away remote communication between sensors, actuators, and controllers. When an interface module of some component has data to send to another the data agent first queries the registrar for information about the target component. If it is a remote one, the local agent forwards the request to the data agent on the destination machine. If the destination is local, data is passed to that component's interface module via shared memory.

Figure 9 depicts the above components and their interactions. The architecture allows easy and flexible configuration of control loops in which the controlled system is a software process.

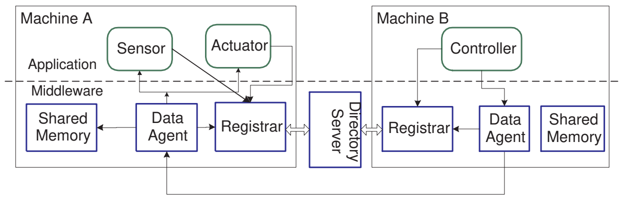

Figure 9. The Software Bus

4 Resource Management

An important challenge in applying a control-theoretic paradigm to software QoS control lies in finding appropriate sensors and actuators for software services. 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 that is reset periodically. 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., some queue length) or the operating system (such as CPU utilization). All one needs to do to implement a sensor, is insert an API call such as **componentWrite()** into the application code or call the OS and pass the returned value to the middleware.

The main application interface challenge, therefore, lies at the application/actuator boundary. To meet this challenge, our middleware includes a generic resource manager that serves as a multipurpose actuator. The manager exports a uniform API to the application and has a back-end interface to the machine's native resource allocation mechanisms. In this section, we present the design of our multipurpose actuator and the interface between it and the application. Note that, custom-made actuators that are not based on our generic resource manager can still be interfaced to SoftBus as described in Section 3, since the latter is oblivious to the type of actuator used.

Our generic resource manager (GRM) is designed for use with Internet servers such as web servers, DNS servers, mail servers, and proxy cache servers. It 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 10. In the above figure, the Classifier and Resource Allocator are provided by the application. The Resource Allocator executes allocation decisions. 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. The box labeled Actuator exports actuator API.

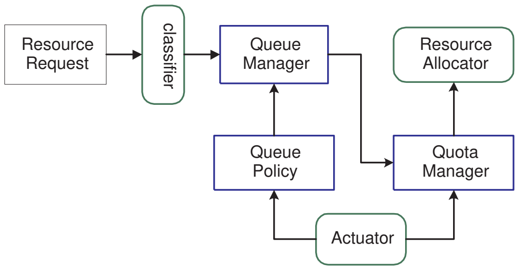

Figure 10. Structure of GRM

4.1 Customizing the GRM

To make this manager general and flexible, we try to expose as many tunable 'knobs' as possible so that the application can control the behavior of the manager as needed. These knobs are exposed to the outside world as *policies*. Currently, we support six categories of policy:

1. *Space Policy:* This policy controls the total space used by the managed resource queues and the space allocation among the queues. The total space can be unlimited (limited only by available memory) or limited. The application can also set a limit on some(all) queues and let the remaining queues share the remaining space.

2. *Overflow Policy:* This policy takes effect only when some queues are sharing limited space and the space gets used up. Two options are supported: *reject* and *replace*. When the policy is reject, current request will be simply rejected. If the policy is replace, the last request of the lowest priority queue that shares the limited space will be evicted from the queue (application will be notified via *rejectProc*) and current request will occupy its space.

3. *Enqueue Policy:* Apart from the queue for each class, the queue manager also maintains an ordered list of the requests in all the queues. This policy influences the order of the requests in the queue and the list. System default policy is FIFO, i.e., all the requests are appended the queue or the list. Application may provide its own sort procedure to determine the position of this request in the queue or the list.

4. *Dequeue Policy:* This policy influences the dequeuing of the request. Currently three choices are available: FIFO, priority and proportional. FIFO means dequeue the request according to its position in the ordered list. Priority means always process high-priority queue before processing low-priority queue. If proportional policy is chosen, application can specify the dequeue ratio among the classes. For example, by setting the ratio to be $2:1$, the queue for the class 0 will be dequeued twice as faster as the queue for class 1.

5. *Quota Policy:* This policy decides what the quota controls. If it's *resource*, the quota will actually control the resource usage. If it's *dispatch*, it will control the dequeue ratio (in this case, the dequeue policy must be proportional).

6. *Preempt Policy:* This policy will take effect only when the quota policy is *resource*. When one class's quota is used up and a new request arrives, there are two

choices: simply reject it and buffer it, or revoke some resource that was previously allocated to this class and satisfy current request(this is when *revokeProc* get called). Obviously, only space-multiplexed resource can make use of this option.

4.2 Interaction Between GRM and 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*. Among other parameters, the call includes an application-specific interpretation of a resource amount consumed by the request. It is expressed in the same units as resource quota. The GRM controls resource allocation by checking the input request stream against two constraints; (i) the queue length constraint, and (ii) the quota constraint. It is important to mention that quota is a purely logical concept. Unlike the case with a traditional resource reservation system, in our middleware the mapping of quota to physical resource consumption need not be known. This lack of knowledge is not a problem in a feedback-based system, since the system constantly monitors performance and adjusts logical quotas by the controllers accordingly until performance goals are met. The only restriction we need is that a change in logical quota be correlated with a change in the actual resource consumption pattern. The application is free to use any resource units it wants in the *insertRequest* call. For example, it may use 1 (for "one" request).

Upon receipt of a classified request, the GRM operates as follows. If the queue for the given class is empty and the class has quota, the request is satisfied immediately. A resource allocation call, *allocProc* is made to the resource allocator to admit the request. The logical quota is decremented by the amount specified in *insertRequest*. For example, in the web server performance control case presented in Section 5, the allocated entities are the server's worker processes. Hence, the quota is decremented by 1. In the proxy cache performance control case, the allocated entity was disk space. The quota is decreased by the file size.

If the request can't be satisfied immediately, it will be buffered in the appropriate queue. The buffering behavior depends on the policy of the queue manager. When some resource becomes available (e.g., in our web server example, the worker process is done with a request, or in our cache example a file is purged from the cache), the application calls *resouceAvailable* to notify GRM. The GRM first increments the quota usage by the amount passed previously in the appropriate *insertRequest* call, then tries to satisfy as many pending requests as possible.

Figure 11 summarizes this interaction. 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 in a trial-and-error fashion until performance constrains are met. The trial and error 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.

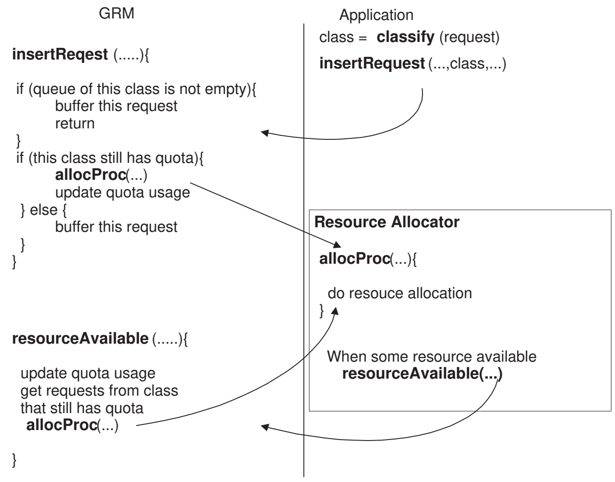

Figure 11. Resource allocation procedure

5 Evaluation

To test ControlWare, we instrument the Apache [12] server and Squid [26] server to interface to the middleware. We specify relative service differentiation as the QoS objective. On the proxy cache, we require differentiation in terms of hit ratio achieved to different content classes. On the server, we require differentiation in terms of service delay. While the semantics of the performance variable being differentiated are not interpreted by the middleware, our choice of variable is implicitly expressed in the choice of sensors. Hence, we instrument Squid to measure hit ratio and instrument Apache to measure service delay. These sensors are interfaced to SoftBus. The two servers chosen are quite different in terms of the resource types managed to achieve performance differentiation. In Squid, we manage cache size allocated to each content class. In Apache we manage the number of processes allocated to serve requests of each class. By applying our middleware to these servers, we demonstrate its versatility and ability to satisfy diverse performance guarantees. We describe our experiments in more detail next.

5.1 Providing Hit Ratio Differentiation in Squid

Figure 12 depicts the structure of the instrumented Squid. Cache space is shared by several classes and each class has a quota of the space. Generally, the space used by some class will directly affect its hit ratio. The sensor, actuator and controller provided by the ControlWare constitute the control loops. ControlWare creates one for each class. Each sensor $S(i)$ returns the relative hit ratio of class i, i.e., $\frac{HR_i}{\sum_{k=0}^{k=n} HR_k}$. Each actuator changes the value of $deltaSpace_i$ and passes it to the allocator. According to $deltaSpace[]$, the allocator implements the basic cache replacement policy which in Squid is a standard LRU. It also imposes a logical constraint on space used by each class that shares the LRU queue. The sensor and actuator are of the passive type. Periodically, ControlWare invokes the controller, which reads data from the sensor via SoftBus, calculates the resource change to be applied, and writes the result to the actuator via SoftBus.

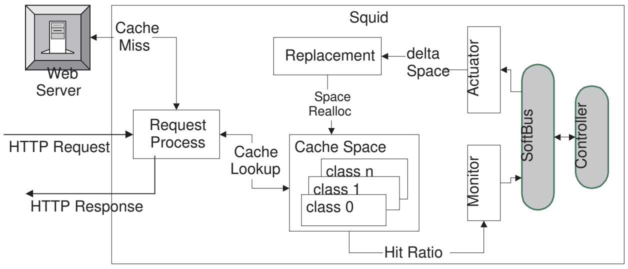

Figure 12. Structure of the modified Squid

The above proxy cache prototype was tested on a 100Mbps Ethernet LAN of nine Linux PCs. Each machine has a 450MHz AMD K6-2 processor and a 256MB RAM. Three client machines run Surge[8] to generate workload. Surge is a web benchmark known for its realistic reproduction of real web traffic patterns such as manifestation of a heavy-tailed request arrival and file-size distributions, a Zipf requested file popularity distribution, and proper temporal locality of accesses. Each client machine simulates 100 users. Three machines were used to run Apache. Each client machine generates requests for the content located at one of the Apache machines. In our experiment, there are 3 content classes. We specified that the hit ratio of the three classes satisfy the condition $H_0 : H_1 : H_2 = 3 : 2 : 1$. Controller tuning was performed automatically by the middleware. Figure 13 shows the observed hit ratio differentiation during the experiment.

As we can see from Figure 13, the squid server successfully provides the specified hit ratio differentiation, illustrating the success of the middleware in providing performance guarantees.

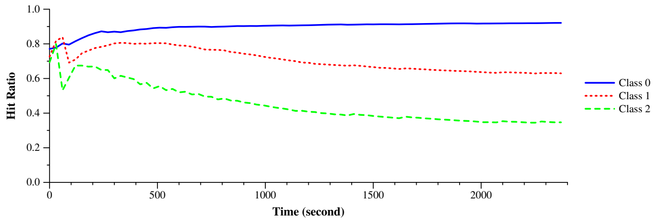

Figure 13. Hit Ratio of three classes

5.2 Providing Delay Differentiation in Apache

We interfaced the Apache web server to SoftBus as depicted in Figure 14. We implemented a request classifier, and a delay sensor. The generic resource manager described in Section 4 was used as the actuator. The GRM was interfaced to a resource allocator which passed accepted requests (socket descriptors) to background Apache processes when instructed by the GRM.

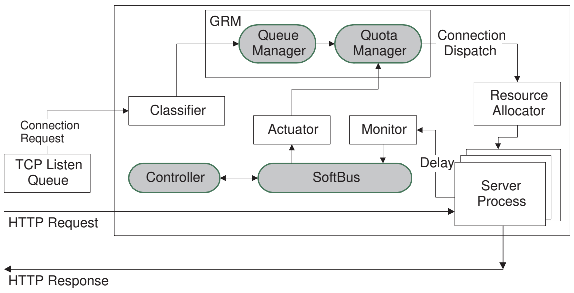

Figure 14. Structure of the modified Apache

Four client machines were used to run Surge and generate realistic server workload. As before, each client simulates 100 users. We divided the client machines into two classes with two machines per class. In the first half of the experiment, only one machine from class 0 generates requests. The second one is turned on after 870 seconds. We specified that the connection delay D_i of class 0 and class 1 should satisfy $D_0 : D_1 = 1 : 3$ at all times. Figure 15 shows the results of this experiment.

From the figure, we can see that before 870 seconds, the delay of class 1 is about 3 times of the delay of class 0 as specified. When the second machine of class 0 is turned on, the delay of class 0 is increased suddenly. The controller reacts by allocating more processes to class 0. At about 1000 seconds, the delay ratio converge to around 3 again.

The evaluation clearly demonstrates that the middleware is capable of providing the specified performance guarantees with only a modest instrumentation cost and no control-

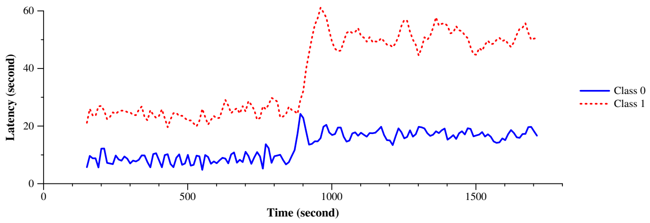

Figure 15. Relative Delay between two classes (controlware version)

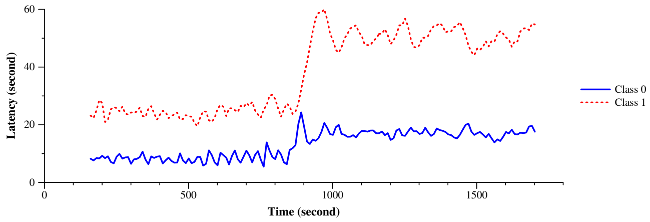

Figure 16. Relative Delay between two classes (hand-crafted version)

theoretical experience required from the software developer. The mideware is versatile in that it is not tailored for a specific software service or a specific performance metric. Not only does the middleware allow new services to be efficiently augmented with QoS provisioning, but also it makes it easy to retrofit delivery of QoS assurances into services that were not designed with this purpose in mind. This paper provides a proof of concept of the utility of ControlWare as an embodiment of a general control-theoretic paradigm for QoS guarantees in software systems. Relative guarantees were used as an example in our evaluation. The authors will report on a detailed evaluation of other types of guarantees and other Internet services in a subsequent publication.

6 Related Work

The control theoretical approach has been successfully applied to in several computer system projects. At the network layer, Hollot et. al. [16] applied control theory to analyze the RED congestion control algorithm on IP routers. Recently feedback control is developed for active queue management to provide loss and delay differentiation via active queue management [10].

In the area of CPU scheduling, Steere et. al. [30] developed a feedback based CPU scheduler that synchronizes the progress of consumers and supplier processes of buffers. In [20], feedback control real-time scheduling algorithms were developed to provide deadline miss ratio guarantees to real-time applications with unpredictable workloads.

Recently Internet server software has become a focus area of feedback control because the unpredictabilities of the workload. Examples of such QoS control includes delay and bandwidth control in Web servers [4, 18], hit ratio differentiation in web caches [21] and queue management in e-mail servers [24].

The above projects demonstrated that feedback control provides a powerful theoretical foundation to provide robust QoS guarantees in a wide range of software systems. However, their feedback control loops are implemented as individual cases from scratch. Significant efforts are needed to develop and tune the feedback control loop in every case. No middleware has been developed in the above projects to provide general support for composition and tuning of software feedback control loops.

The SoftBus architecture in ControlWare is related to distributed middleware such as CORBA [15] and DCOM [11]. Similar to the location-transparent method invocation in CORBA and DCOM, SoftBus allows the plug-and-play of control loop components (i.e., monitors, controllers, and actuators) independently of their locations. However, differently from CORBA and DCOM, SoftBus provides direct support for feedback control by supporting active and passive interaction mechanisms among monitors, controllers, and actuators, as well general resource managers for server systems. Furthermore, ControlWare provides system identification and controller tuning services that are not supported by other common middleware services.

The SWiFT project [14] at OGI and the Agilos project [17] at UIUC share some similar goals with ControlWare. SWiFT is a toolkit for constructing feedback control loops from libraries. It also supports the visualization and simulation of software control loops. The Agilos project constructs a middleware to support QoS control and adaptation. ControlWare is different from SWiFT and Agilos in its unique SoftBus architecture that enables flexible plug-and-play of any control components in a location independent fashion (e.g., components of a same control loop can be from different address spaces and even remote nodes. In comparison, Agilos has a fixed two level control structure and a fixed set of monitors (e.g., CPU and bandwidth monitors). More importantly, ControlWare is the first middleware that provides end-to-end support of the whole development process of creating QoS control in software systems. This process starts from defining QoS contracts, mapping contracts to feedback control loops, system identification, controller tuning, and implementation. For example, neither SWiFT or Agilos supports the mapping from QoS contracts to feedback control loops or system identification. They also do not provide the generic resource manager as in ControlWare.

7 Conclusions

In this paper, we described a new middleware architecture for QoS guarantees in distributed environments such as the Internet. The architecture implements a new paradigm for QoS control, which is especially suitable for system operating in highly uncertain environments or when accurate system load and resource models are not available. Our preliminary evaluation of the ability of this middleware to provide advertised guarantees in the context of selected different applications illustrates the promise of this approach. While prior efforts have been made to apply control theory to QoS control, ours is the first comprehensive middelware service that incorporates these principles under a clear well defined set of APIs which substantially reduces the development effort of performance assured applications and Internet services.

Future work of the authors will focus on understanding the limitations of a control-theoretic approach and deriving new guarantee semantics. A possible disadvantage of using feedback only as a means to correct performance is the need for a performance error to occur first before a feedback controller can respond. In the future, we shall focus on mechanisms that combine prediction with feedback to improve convergence to specifications in a highly dynamic unpredictable system. We shall also extend the middleware to allow fully dynamic online re-configuration during normal system operation, and investigate other types of performance guarantees that might be achievable in a feedback control context. For example, it may be interesting to cast adaptive guarantees on service availability, security, and fault-tolerance as feedback control problems.

Acknowledgments

The authors would like to thank Sang Son, Gang Tao, and Ying Lu for providing useful comments.

References

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

[2] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RT-CAST: Lightweight multicast for real-time process groups. In *IEEE Real-Time Technology and Applications Symposium*, Boston, Massachusetts, June 1996.

[3] 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.

[4] T. F. Abdelzaher and N. T. Bhatti. Web content adaptation to improve server overload behavior. *WWW8 / Computer Networks*, 31(11-16):1563–1577, 1999.

[5] 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*, 2001. Accepted.

[6] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[7] K. J. Astrom and B. Wittenmark. *Adaptive Control*, chapter 2. Addison Wesley, 2nd edition, 1995.

[8] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

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

[10] N. Christin, J. Liebeherr, and T. Abdelzaher. A quantitative assured forwarding service. Technical Report CS Technical Report 2001-21, University of Virginia, 2001.

[11] M. Corporation. Distributed component object model protocol - dcom/1.0, 1998.

[12] A. S. Foundation. http://www.apache.org.

[13] N. Gandhi, S. Parekh, J. Hellerstein, and D. Tilbury. Feedback control of a lotus notes server: Modeling and control design. In *American Control Conference*, 2001.

[14] A. Goel, D. Steere, C. Pu, and J. Walpole. Swift: A feedback control and dynamic reconfiguration toolkit, 1999.

[15] O. GROUP. The common object request broker: Architecture and specification, 1995.

[16] C. Hollot, V. Misra, D. Towsley, and W. Gong. A control theoretic analysis of red, 2000.

[17] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations, 1999.

[18] 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.

[19] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *IEEE Real-Time Systems Symposium*, Orlando, FL, December 2000.

[20] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems Journal*, Special Issue on Control-Theoretical Approaches to Real-Time Computing, March-April, 2002.

[21] Y. Lu, A. Saxena, and T. F. Abdelzaher. Differentiated caching services; a control-theoretical approach. In *International Conference on Distributed Computing System*, Phoenix, Arizona, April 2001.

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

[23] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[24] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. S. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In

IFIP/IEEE International Symposium on Integrated Network Management, 2001.

[25] L. Rodrigues, P. Veríssimo, and J. Rufino. A low-level processor group membership protocol for LANs. In *Proc. Int. Conf. on Distributed Computer Systems*, pages 541–550, 1993.

[26] S. W. P. Server. http://www.squid-cache.org.

[27] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[28] F. G. Shinskey. *Process control systems: application, design, and tuning*. McGraw-Hil, New York, 4th edition edition, 1996.

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

[30] 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.

Appendix A: Contract Description Language

The Contract Description Language (CDL) is used to describe desired convergence guarantees. Its main syntax is as follows.

```
GUARANTEE NAME {
  GUARANTEE_TYPE = type;
  TOTAL_CAPACITY = capacity;
  CLASS_0 = QoS_0;
  CLASS_1 = QoS_1;
  CLASS_2 = QoS_2;
  ......
  CLASS_num = QoS_num;
}
```

GUARANTEE_TYPE: The GUARANTEE_TYPEs currently supported by ControlWare include ABSOLUTE, RELATIVE, and STATISTICAL_MULTIPLEXING. Different guarantee types need different interpretations of the QoS values and are mapped to different feedback control loops as described in Section 2.2. Although reservation and utility optimization are not listed as a guarantee type, they are equivalent to absolute guarantees because they are all mapped to single feedback control loop per class.

TOTAL_CAPACITY: total capacity is only useful when GUARANTEE_TYPE = STATISTICAL_MULTIPLEXING. The set point of the best effort server is the total capacity minus the capacity allocated to all guaranteed service classes.

CLASS_i: Each service class represents a category of requests with a guarantee depending on the application requirements. For example, a service class can be all the HTTP requests from premium clients, or the requests related to the check-out procedure in an e-commerce server. The assignment CLASS_i = QoS_i specifies the guaranteed QoS for class i. Note that the guaranteed QoS have different meanings for different guarantee types. For ABSOLUTE and STATISTICAL_MULTIPLEXING guarantees, QoS_i represents the absolute value for desired QoS, while RELATIVE guarantees are only interested only the relative value (ratio) between the QoS_i's of different classes.

Using CDL, system developers and users can specify a wide range of QoS guarantees in a precise way, and then run The QoS mapper to map the CDL script into the feedback control loops whose topology is stored in a configuration file.