

Kernel Support for Open QoS-Aware Computing *

Ronghua Zhang, Tarek F. Abdelzaher, and John A. Stankovic
Department of Computer Science, University of Virginia
Charlottesville, VA 22904
{rz5b,zaher,stankovic}@cs.virginia.edu

Abstract

Most research on QoS-aware computing considers systems where code is generally partitioned into separately schedulable tasks with associated timing constraints. In sharp contrast to such systems is a myriad of mainstream off-the-shelf applications and services such as Web servers, caches, mail servers, and content distribution proxies where QoS guarantees may be needed, yet the software follows a best-effort one-size-serves-all model. In this model, different traffic classes are not mapped to different schedulable entities (tasks), making it impossible to use real-time scheduling meaningfully to satisfy application QoS.

This paper presents a kernel-level solution to the problem of retrofitting such best-effort systems with QoS support without changing application code. The solution has been implemented in Linux. By downloading a few kernel patches and configuring the patched kernel appropriately, a system administrator can endow a best-effort service with QoS assurances transparently to the legacy server. An extensible library is provided in a separate QoS manager that allows implementing different types of QoS guarantees within the extended service. The performance of the resulting system is evaluated on the implemented Linux-based prototype. It is shown that QoS-sensitive behavior is successfully achieved.

1 Introduction

This paper addresses the problem of retrofitting QoS support into best effort server applications without changing application code. Many of the current popular server applications are designed without QoS in mind. They adopt a one-size-serves-all architecture in which, for example, a single common thread or thread-pool might dequeue service requests from a common FIFO operating system queue. In the rest of this paper, we call them *legacy* servers (or legacy application software in general). No support for

performance differentiation is provided. In particular, the architecture offers no mapping between individual entities schedulable by the operating system and different classes of content or clients with which QoS requirements are associated. Hence, real-time scheduling techniques cannot be meaningfully applied to provide QoS guarantees.

One solution to the above problem is to re-write legacy application software to be QoS-aware. For example, a separate task or thread pool may be associated with each traffic class such that different guarantees can be given to different classes by appropriately scheduling these pools. Changing all current server software to this QoS-aware architecture, however, is very costly, considering the volume of best-effort service software that exists today and the frequency with which new versions and releases are created by vendors whose primary market is best-effort applications. Hence, a practical strategy is to find a way to use existing legacy service code *as is*, yet at the same time allow QoS guarantees to be achieved. This offers an interesting new research topic. Namely, given the premise that we do not want to modify legacy service code, how to provide QoS guarantees in those services? The answer would have to lie in mechanisms external to server code and transparent to it.

We propose a kernel-level approach to QoS guarantees in legacy services. A kernel-level approach is justified for two reasons. First, some kernel modifications are unavoidable to provide QoS in an efficient manner. For example, in a web server, most of the clients' waiting time occurs in the kernel queue associated with the server's well known port. If that queue is not prioritized, the service will appear best-effort regardless of how the application software is structured. Second, compared with the large volume of existing legacy applications, the number of the operating system revisions is relatively small. Even in rapidly evolving operating systems, such as Linux, the network stack remains relatively stable. Thus, changing the operating system is more realistic and cost efficient than changing all evolving application code. An additional benefit is that changing one single operating system can give QoS capabilities to many applications running on top of it.

The difference between this paper and many prior papers

*The work reported in this paper was supported in part by NSF grants CCR-0093144, CCR-0208769, DARPA NEST grant F33615-01-C-1905, and MURI grant N00014-01-1-0576

on kernel-level QoS support lies in our explicit goal to eliminate the need for changes to legacy application software while achieving the desired QoS. The mechanisms that we provide are general and can be used with a variety of different legacy servers. While there may be special cases in the way servers interact with the kernel or clients that might require special handling, we believe our current architecture is sufficient for a large set of common servers. As an example, in this paper, we demonstrate the use and performance of our architecture with web, FTP, and mail servers.

The rest of this paper is organized as follows. Section 2 presents an architectural overview of our approach for QoS guarantees in legacy servers. Section 3 elaborates on a Linux-based implementation of this architecture. Section 4 evaluates the performance of the Linux-based prototype using various best-effort servers. Section 5 presents related work. The paper concludes with Section 6.

2 Architectural Overview

Below, we describe the main components of our architecture for QoS guarantees. The first component is the unmodified best-effort server. We describe our assumptions regarding the best-effort server in Section 2.1. The additional components we need to provide QoS guarantees are described in Section 2.2. In Section 2.3 we summarize the operational view of the methodology followed to endow best-effort services with QoS support.

2.1 Assumptions on the Best-effort Server

We consider a common best effort server in which one or more identical worker threads or processes serve incoming client requests. Connection requests to the server arrive on a common port (e.g., port 80 for web servers). These connection requests are dequeued from the server port, either directly by the worker threads themselves, or separately by a dedicated dequeue thread, which will pass the requests to a worker thread. We assume that the server is *multi-instance safe*. In other words, if we run multiple instances of the server, each instance should work well in presence of others. This assumption holds for many Internet server applications, such as web servers, mail servers, and FTP servers. There do exist some applications that violate this assumption. For example, the application may write some data to a file with a hard-coded absolute pathname. In that case, our scheme cannot be applied. Finally, we assume that the server code has been designed with no support for real-time guarantees. Thus, all the threads have the same priority and no code exists in the server to alter the thread priority, for example, based on the client identity.

2.2 Delivering a QoS-aware Service

Figure 1 depicts what a typical QoS-aware service looks like, when it is implemented from scratch. As shown in

Figure 1, three important components are needed. First, a classifier is needed to separate the incoming requests into their appropriate classes. Second, one or more performance sensors are needed to measure and report the performance of each class. Third, one or more actuators are needed to adjust per class resource allocation such that the performance of each class remains at its respective desired level.

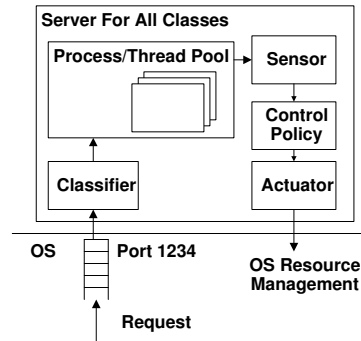


Figure 1. A QoS-aware Application

To convert a legacy application into a QoS-aware one, the classifier, sensors, and actuators must be added. In our approach, we add generic versions of these components as a general kernel-level mechanism. After classification, input request traffic of each class is queued separately on a different, per-class, kernel port (socket). To the legacy server application, these ports look like ordinary sockets. A separate instance of the best-effort server is instantiated to listen to each port. Hence, a separate server instance is assigned to each class of clients.

A separate user-space QoS manager will periodically query the sensor for the performance of each class and adjust resource allocation accordingly. The particular resource management policy used inside the QoS manager is outside the scope of this paper. In our experiments, we use a feedback control technique presented in previous literature [16, 2, 12, 15, 22, 9]. The structure of the resulting QoS-aware architecture is depicted in Figure 2. The components of this architecture are described next.

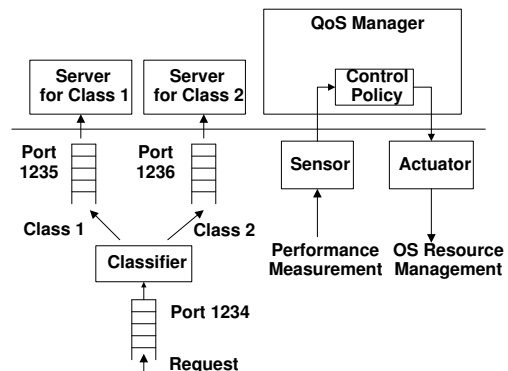


Figure 2. QoS with Legacy Servers

2.2.1 In-kernel Classifier

The in-kernel classifier is controlled by configurable rules. Each rule includes an incoming port and several target outgoing ports, one of which is designated as the default target. The incoming port, usually the well-known port for that service, is the port the clients contact. The outgoing ports are the ones the server instances are listening to. Conditions are associated with each outgoing port. When an incoming connection satisfies the condition associated with one particular outgoing port, the connection will be forwarded to that port. Since one connection may satisfy several conditions at the same time, the condition evaluation stops once a condition is satisfied. The evaluation of the condition is controlled by the order in which the conditions are added by the users. If none of the conditions are satisfied, the default target port is used.

An outgoing port in one rule can be the incoming port of another rule. Thus, it is possible to connect several rules into a forwarding chain, which is a very powerful feature. For instance, the first rule with port X as its incoming port may specify that when condition A is true, connections to port X will be forwarded to port Y. Another rule with port Y as its incoming port may further specify that when condition B is true, connections to port Y will be forwarded to port Z. Therefore, when condition A and B are both true, a connection to port X will finally be forwarded to port Z.

We have implemented a user-space tool, called *tcpchains*, to help the administrators manipulate the rules. To add a new rule to the system, the administrator first creates an empty rule specifying its type and its incoming port as well as the default outgoing port. Then, he/she can add the conditions one by one. In addition, a programmer's interface is created. A programmer can define and manipulate rules via an extension of the *setsockopt* system call.

Three types of classification and connection forwarding rules are supported; *address-based*, *content-based*, and *probabilistic*:

1. Address-based. The user specifies a set of IP addresses/subnet masks for each outgoing port. The target port selection is solely based on the source the connection comes from. For example, the following commands will forward all the FTP connections coming from subnet 128.143.0.0 and 128.141.0.0 to port 2021 and the rest to port 1021:

```
tcpchains -A a -p 21 -o 1021
tcpchains -a a -p 21 -i 128.143.0.0 -m 255.255.0.0 -r 2021
tcpchains -a a -p 21 -i 128.141.0.0 -m 255.255.0.0 -r 2021
```

The first command creates an empty address-based forwarding rule(-A a) with 21 as incoming port(-p 21) and 1021 as default outgoing port (-o 1021). The remaining two commands specify conditions: -i option specifies the IP address, -m option specifies the subnet mask

and -r option specifies the outgoing port when the condition is satisfied.

2. Content-based. This is the most powerful and complex classification scheme. A general filter is used to perform pattern matching on application-layer headers. Our current implementation only supports simple string matching, which is sufficient, for example, to determine which URL was named in an HTTP header. The administrator can specify certain strings for each outgoing port. The connection is forwarded only when one of the specified strings appears in the data packet. The following commands will forward all the requests for website www.foo.com to port 8081, all the requests for website www.bar.com to port 8082 and the rest will be forwarded to port 8080 by default:

```
tcpchains -A c -p 80 -o 8080
tcpchains -a c -p 80 -s www.foo.com -r 8081
tcpchains -a c -p 80 -s www.bar.com -r 8082
```

In the above commands, -s option is used to specify the string pattern.

3. Probabilistic. A user or a user-space process can associate a probability with each outgoing port of a rule. Incoming connections are forwarded according to these probabilities. For example, the following commands will forward 30% of the HTTP connections to port 8081, and the rest to port 8080:

```
tcpchains -A d -p 80 -o 8080
tcpchains -a d -p 80 -r 8081 -c 3000
```

Forwarding probabilities can be changed dynamically in software. Probabilistic forwarding may be useful in cases where all clients belong to the same class, but resources are not sufficient to serve all of them. In this case, some dynamic fraction of them may be diverted to a different port where it can receive a lower quality of service. In the evaluation section, we elaborate on an example of this classification rule applied to utilization control in a web application.

2.2.2 In-kernel Sensors and Actuators

For each listening port, some resource usage data is maintained in the modified kernel, which includes the number of connections waiting to be accepted, the number of connection accepted so far, the total connection delay, and the number of bytes received and sent out by the connections accepted from this port. A user-space tool, called *netmeter*, can be used to report this data to the user. An application can query this data using the standard *getsockopt* system call. We call this mechanism, a *performance sensor*.

Kernel resource management schemes are integrated into our framework as actuators. In our current prototype, several socket level traffic control actuators are implemented.

They can limit the connection acceptance ratio of several listening ports, the connection acceptance rate of a single listening port, and the aggregate bandwidth usage of the connections accepted from a single port. Another user-space tool, called *skex*, is developed to control actuator settings by a system administrator. Alternatively, programmers can control these settings using the *setsockopt* system call.

2.2.3 QoS Manager

The QoS manager implements adaptive resource control policies that manipulate actuator “knobs” such that per-class QoS guarantees are achieved. While the QoS manager can in general be based on any of several resource management mechanisms proposed in previous literature, we opt for one that utilizes a feedback-based approach for performance guarantees. It uses a previously presented middleware package, called ControlWare [26], which realizes different QoS guarantees via feedback control loops and implements the necessary controllers.

2.3 A Methodology for QoS Provisioning with Legacy Code

Based upon the architectural components described above, in this section we summarize the methodology for using best effort servers to provide QoS guarantees. It consists of the following steps:

1. Appropriate sensors are chosen from the options provided by the architecture.
2. Several instances of the legacy server are started, one for each class of clients. Each server listens to a different port.
3. Appropriate classifying and forwarding rules are set up to correctly classify and forward incoming connections. While we provide parameterizable common types of classification schemes, if necessary the user can write a new kernel module to supply their own classifier. Note that this classifier only needs to be implemented once.
4. A separate QoS manager process is started. The sensors and actuators are connected to the QoS manager. The manager will collect performance data and interact with the operating system (actuators) to adjust resource usage until desired performance is achieved for each client class.

3 Implementation

To realize the architecture described in the previous section, we have implemented a general classification and connection forwarding mechanism in Linux kernel 2.2.15. We also

implemented socket level traffic control schemes to act as actuators for resource allocation. Their implementation details are presented in this section. The details of the QoS manager can be found in [26]. For the ease of explanation, we will use words connection and socket interchangeably.

3.1 Classification and Connection Forwarding

To better appreciate the details of our implemented classifier, let us first briefly review the original (unmodified) TCP connection implementation in Linux. In Linux, each socket has one queue dedicated to process TCP connection requests: *syn_wait_queue*. Each socket has a hook function, called *data_ready*. When a SYN packet arrives, the kernel creates an *openreq* structure to represent the embryonic connection, and appends it to the *syn_wait_queue*. When the 3-way handshake completes, a socket structure is created for the connection and the socket’s hook function *data_ready* is called. This function is also called when a new data packet arrives for this connection. The default implementation is to wake up any process blocked on this socket (either waiting for new connection or new data). The *accept()* system call removes the first finished connection from *syn_wait_queue*.

In our modified kernel, when a forwarding rule is created, the kernel creates an internal socket for the incoming port of the rule, and listens to it. That socket’s *data_ready* hook function is then replaced by the proper forwarding function according to the rule. When a connection is ready, (i.e., the TCP 3-way handshake is finished), our customized forwarding function gets called instead of the original one. In general, the forwarding function will test the condition associated with each outgoing port and the first port whose condition is satisfied becomes the forwarding target. The newly ready connection will then be moved from the incoming port to the target port, and finally the latter socket’s *data_ready* hook is called. This invocation will either trigger another forwarding rule (when the target port is the incoming port of another rule), or wake up a blocked process if there is any.

3.2 Actuation: Socket Level Traffic Control

Traffic control is based upon the concept of a *socket group*. A *socket group* is a set of sockets, whose aggregate traffic is subject to some actuation constraint. Usually, the administrator specifies the socket group by only listing several listening ports. All the connections for these listening ports belong to the same socket group. For example, if the listening ports 21 and 80 are in one socket group, then all the TELNET and Web connections for server instances that listen on those ports are in one socket group.

Three types of socket groups are supported: *accept ratio*, *accept rate* and *output*. The *accept ratio* group can control the relative connection accept rate of several listening ports.

The *accept rate* group can limit the rate of connections accepted at one port. The *output* group can limit the total bandwidth usage by all the connections within that group.

Internally, each socket maintains a pointer to the socket group it belongs to. When a process tries to accept a new connection or send out some packet, the quota of the socket group is checked. If the quota is not depleted, the process can proceed. Otherwise, it is forced to sleep. Each socket group also has a timer to periodically replenish the quota. Once the quota is replenished, the processes blocked are awakened.

The above scheme works well for most of the TCP-based services except FTP. FTP is different from other TCP-based Internet services in that it uses two connections per session: one for control and the other for data transmission. To limit the bandwidth usage, both connections need to be controlled. However, the special passive mode in FTP complicates the problem.

In passive mode, a client first requests to enter the passive mode by command **PASV**. Receiving this command, the server starts to listen on some port N . In its response back to the client, the server specifies which port it listens to. The client will issue the connection request to that port. Since port N is decided at runtime, the traffic of the control connection must be monitored. Once the response for **PASV** is detected, we find out the port number N and force it to be in the same socket group.

4 Evaluation

In this section, we demonstrate the efficacy of our architecture in retrofitting QoS-support into best-effort services. The main goal of the experiments shown here is to provide examples of the versatility of our framework. These examples are not intended to be in-depth case studies of particular QoS-aware application scenarios, nor do they mean to provide full justification of the need for QoS guarantees in the applications considered. The main points being made here are that (i) reasonable QoS differentiation can be achieved without modifying application code, (ii) the approach applies equally well to a variety of best-effort servers, and (iii) its incurred overhead is generally acceptable.

We demonstrate the versatility of our approach both in terms of the best-effort services it can support and the QoS guarantees it can provide. Hence, we demonstrate attainment of delay guarantees in web servers and IMAP servers, performance isolation in web servers, and bandwidth guarantees in FTP servers. In each case, a common best effort server is used. The server is executed as-is on top of the modified kernel. The QoS manager (ControlWare) is configured to provide the appropriate type of guarantee. All the experiments are conducted on a testbed of 10 Linux machines connected by a 100Mbps LAN. Each machine has a 450MHz AMD K6-2 processor and 256MB RAM. The

server runs Linux kernel version 2.2.15 with our patch applied. The client machines all run the Linux kernel version 2.4.17. During the experiment, the data is collected every 10 seconds.

4.1 Delay Guarantees for Web Servers

In an earlier publication [14], the authors point out that connection delay is a major factor of user-perceived delay in web services. In this experiment, we show how to achieve absolute connection delay guarantees using an unmodified Apache web server.

In this experiment, a set of 8 client machines running `httperf` [18] (a web micro-benchmark from HP Labs) are used to overload the server. Collectively, they emulate a large community of clients by repeatedly sending HTTP requests. Of these, four machines emulate premium users. Figure 3 shows the average connection delay of all clients in the best-effort case when no classification mechanism is used and when all the users are serviced by a single Apache server. Observe that the same average delay of approximately 4.5 seconds is experienced by all clients.

Now assume it was desired that premium clients (who presumably are paying for the service) are to be guaranteed an average connection delay of only 2 seconds. No restrictions are specified on the delay of basic clients. To meet this guarantee, two Apache servers (named premium server and basic server) are started on the same machine at ports 8081 and 8080, respectively. The connections coming from premium users are forwarded to port 8081. Basic clients are forwarded to port 8080.

By separating premium and basic requests using an address-based classification rule, we allow the two server instances to run at different priority levels such that the premium server has higher priority than the basic one. Figure 4 plots the average connection delay of both premium and basic clients as a result of this prioritization. Although the connection delay of premium users has gone down, it is still around 3 seconds, which is above the target. This because the server as a whole is heavily loaded in this experiment.

An additional mechanism is called for to further lower the delay down to 2 seconds. Observe that while clients are now prioritized, a significant amount of priority inversion occurs due to blocking over other resources such as disk and network I/O. Also, the premium server does not always preempt the basic server even when the priority of the former is higher. A detailed discussion of Linux scheduling that explains this anomaly can be found in [5]. Hence, by throttling lower priority clients, we can reduce the competition on shared resources, thereby improving the delay of high priority clients.

Consequently, in the next part of the experiment, in addition to server prioritization, we invoke an actuator that regulates the accept rate of the basic server (port 8080). The

QoS manager is responsible for the rate regulation. A sensor attached to port 8081 reports to the QoS manager the average connection delay of premium clients periodically. An actuator changes the accept rate of port 8080 in a direction that attempts to stabilize this delay at its target value. The actual accept rate is calculated by a controller tuned using control theory. The design of such controllers has been described in prior publications [14, 17, 9], and is beyond the scope of this paper.

The resulting configuration of the server is depicted in Figure 5. The corresponding observed connection delay is plotted in Figure 6. As shown in Figure 6, the connection delay of premium users (left vertical axis) is now kept around 2 seconds. The figure also shows the number of basic clients admitted (right vertical axis). Since in our experiment, premium clients alone consume a large fraction of server capacity, the delay target is achieved at the expense of eliminating most of the best effort traffic. Note that the scheme can be generalized to more than two classes. In general, priorities must be assigned inversely proportional to the clients' relative deadlines. Traffic throttling is applied until the load is reduced to the point where all deadlines are met.

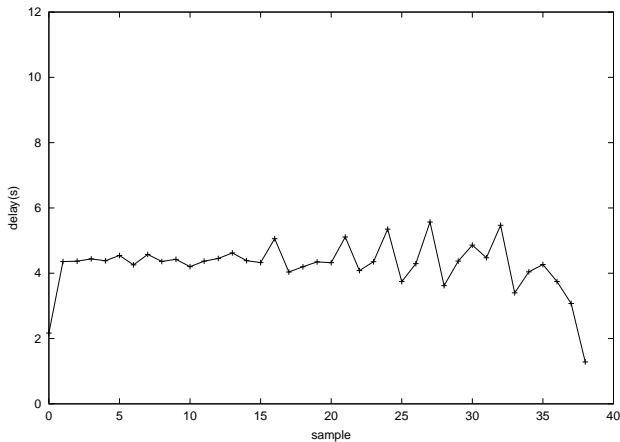


Figure 3. Connection delay with no classification

4.2 Delay Guarantees for IMAP Servers

To show the applicability of our scheme across multiple types of best effort servers, we repeat the previous experiment with an IMAP (mail) server, instead of the Apache web server. As before the objective is to achieve a specified average delay for the premium user class. IMAP (Internet Message Access Protocol)[8] is a protocol to access electronic messages on a mail server. In our experiment, we use Imapd [19], a best effort IMAP server developed at the University of Washington.

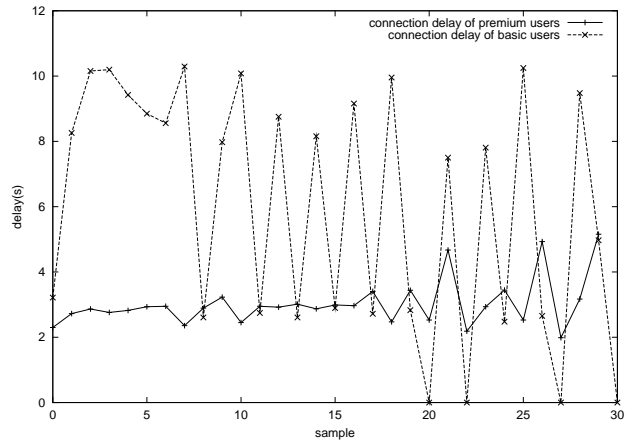


Figure 4. Connection delay of premium and basic users

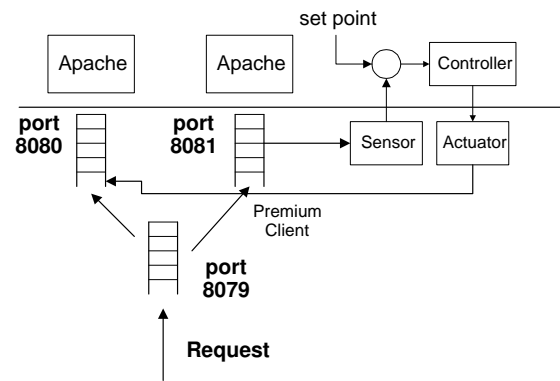


Figure 5. Server configuration to achieve connection delay guarantees

Four machines are used to emulate 60 users, with 15 users on each machine. Among them, two machines are classified as premium users. Each user repeatedly generates a session. Within each session, a connection to the IMAP server is first opened, then 5 email headers followed by their 3KB message bodies are fetched. After that, these messages are deleted and then undeleted. Finally, the connection is closed and another session begins. All communication is encrypted using SSL.

Figure 7 and Figure 8 show the delay of premium clients before and after invoking QoS support. In the absence of QoS support, all the users are treated equally. The connection delay is around 4 seconds as shown in Figure 7.

To provide a delay guarantee, xinetd is configured to start imapd at ports 1143 and 2143 (numbered server 1 and server 2)¹. Connections from premium users are forwarded to port 1143. The rest are forwarded to port 2143. Our target is

¹Imapd is not a stand-alone application. Instead, it depends on xinetd or inetd to start it.

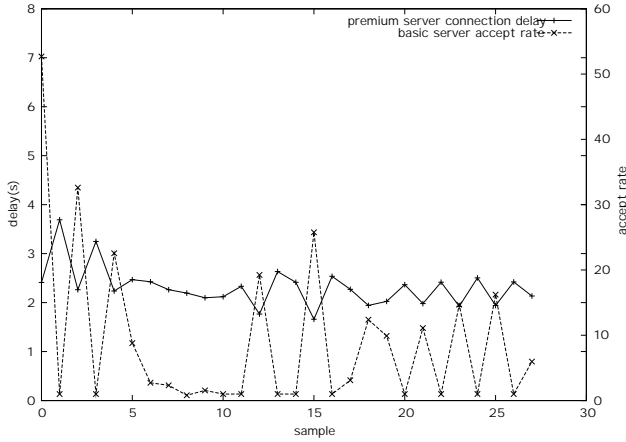


Figure 6. Web server: connection delay of premium users in the presence of rate control

to achieve 2 seconds connection delay for premium users. This is achieved by regulating the accept rate of port 2143 the way we did in the previous experiment. As shown in Figure 8, the connection delay of premium users is thereby maintained at around 2 seconds (left vertical axis). Also plotted is the accept rate of basic clients (right vertical axis). Notice that when the delay of the premium users is longer than 2 seconds, the accept rate of basic clients is reduced. At other times, more basic connections are allowed.

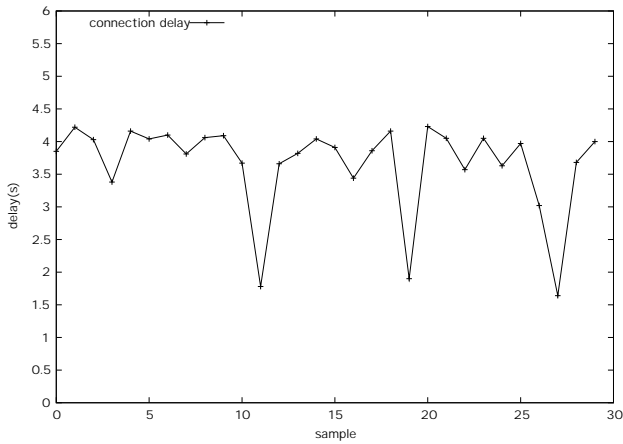


Figure 7. Average connection delay of mail clients

4.3 Performance Isolation in Web Servers

Having illustrated the feasibility of delay control in our architecture, we move on to a different type of guarantees, namely guarantees on resource consumption. Such guarantees may be needed, for example, when multiple services share resources and it is desired to bound their respective re-

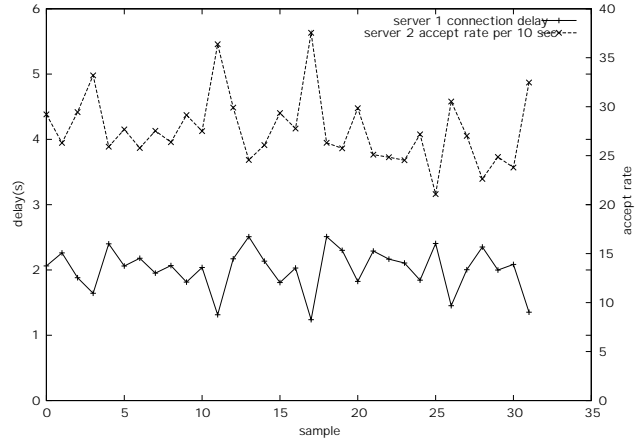


Figure 8. IMAP server: Connection delay of premium users in the presence of rate control

source quotas. In this section, we demonstrate how to limit the CPU utilization consumed by a web server. Let the CPU share of the server be 30%. A two-level control scheme is adopted to achieve the desired resource allocation: namely, content adaptation and admission control. To achieve the illusion of a web server with a content adaptation capability while using a regular Apache web server with no such adaptation support, we pre-process the hosted web content a priori to generate two different versions. The first is the original content, which we call full content. The second is an abbreviated version (e.g., with backgrounds and cosmetic icons removed, and images compressed to lower quality). We call it degraded content. Two Apache servers are started at ports 8080 and 8081, respectively. The server at port 8080 serves full content, while the one at port 8081 serves degraded content. When the server's CPU consumption exceeds its allotted resource share, the QoS manager first tries to reduce resource usage by forwarding more connections to the degraded content server. When all the requests are forwarded to it and the CPU utilization is still above the limit, admission control is activated, which reduces the accept rate on the degraded content server. When the CPU utilization is below the allotted limit, the controllers are activated in the reverse order: first the accept rate of the degraded content server is increased until all connections are accepted. Then, more connections are forwarded to the full content server until the desired CPU utilization is achieved.

Figure 9 shows the server configuration. As before, httpperf is used to generate client requests. The request rate is doubled after 200 seconds, and is reduced in half after 500 seconds. Figure 10 plots the CPU utilization, percentage of degraded content and the connection accept rate observed during this experiment. In the beginning, content adaptation alone brings the CPU utilization down to the set point of 30%. When the request rate is doubled at 200 seconds, CPU

utilization rises abruptly. The fraction of requests forwarded to the degraded content server increases sharply, then admission control is invoked. Observe that when server load increases, more requests are admitted which might seem counter-intuitive. The reason is that at higher load, more requests are served degraded content, which significantly reduces per-request resource consumption allowing more clients to be served. After an initial perturbation, the CPU utilization returns back to its set point of 30%. At 500 seconds, the request rate is halved. Consequently, CPU utilization drops. Eventually, all requests are admitted and enough are forwarded to the full content server. The CPU utilization is back to 30% again.

Combined with address-based or content-based forwarding, the above mechanism can be easily extended to limit the resource consumption by one particular class of users or one particular web site in case of web hosting.

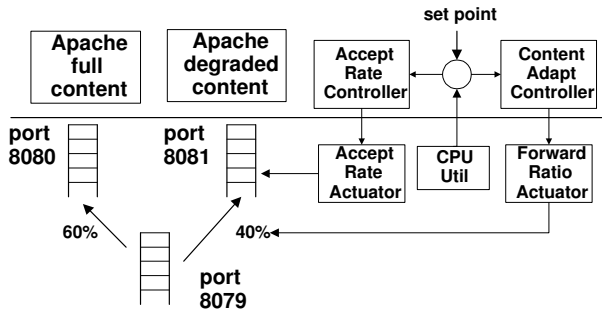


Figure 9. Server configuration for CPU utilization control

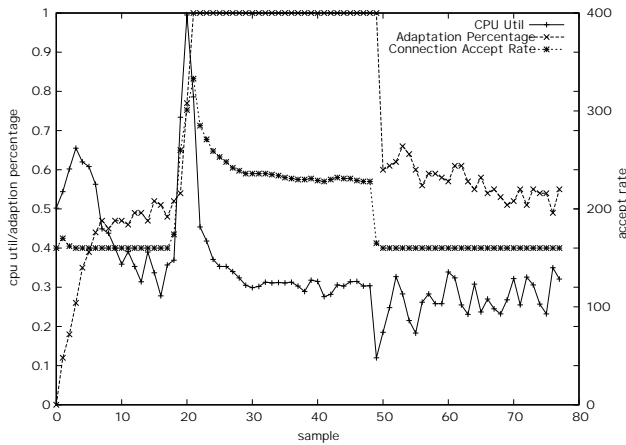


Figure 10. Performance of CPU utilization control

4.4 Performance isolation in FTP servers

Our second experiment on resource share control is to support total bandwidth allocation for different classes of

number of users	total bandwidth usage	
	restricted users	normal users
4	98KB/s	9500KB/s
3	96KB/s	9500KB/s
2	97KB/s	9100KB/s
1	96KB/s	7400KB/s

Table 1. Total bandwidth usage

clients accessing an FTP server. We use wu-ftp [10], which is a very popular FTP server. It supports some level of service differentiation by limiting the bandwidth usage per single client. However, it cannot limit the total bandwidth consumption of an entire client class. Our kernel support easily allows us to add this feature into wu-ftp. To demonstrate this feature, two FTP servers are started at port 2021 and 3021, respectively. Clients are classified into two classes via their IP address: restricted users and normal users. Clients contact the server at port 21 as usual. Connections from restricted users are forwarded to port 2021. All others are forwarded to port 3021. A bandwidth limit of 100KB/s is imposed on port 2021. Thus, the aggregate bandwidth usage by the restricted users cannot exceed 100 KB/s.

In the experiment, a different number of restricted users and normal users request some large file at the same time. Table 1 summarizes the aggregate bandwidth usage by each class. It shows that the restricted user class cannot consume more than 100KB/s bandwidth even if many users are downloading simultaneously, thus demonstrating the efficacy of our policy. In contrast, the total bandwidth usage by normal users increases as more users are downloading at the same time until the entire Ethernet bandwidth is totally consumed. The experiment illustrates the ability of our architecture to achieve bandwidth control.

4.5 Overhead

Since for each new connection, the kernel has to evaluate the conditions of the classification and forwarding rules to decide the forwarding target, the overhead of the classification and forwarding is directly related to the number of conditions. To show this relationship, we start one Apache server at port 8080 and use 4 clients running httpperf to send out requests to port 8079. An address-based or a content-based rule is created with 8079 as the incoming port and 8080 as the default target port. Some conditions are attached to another outgoing port 8081, but these conditions are intentionally designed so that none of them can be satisfied. Consequently, the kernel has to walk through all these conditions and at last forward the connection to port 8080. In the experiment, we change the number of conditions and observe the number of connections the server can process.

Each experiment is repeated 5 times and the average value is used for comparison. The results are summarized

Number of conditions	Connection Processed (loss)	
	Address-based	Content-based
0	54916	54916
1	54916 (0%)	53602 (2.4%)
2	54257 (1.2%)	53570 (2.5%)
4	54278 (1.2%)	52761 (4.0%)
8	54298 (1.2%)	49387 (10%)

Table 2. classification overhead

in Table 2. The numbers in parentheses represent the server capacity loss in terms of the number of served connections compared to the case when no conditions are present. An obvious result is that address-based classification has smaller overhead than content-based classification, which is reasonable, since address-based classification only involves IP address matching. Moreover, there is no obvious overhead increase when more conditions are added, since the address matching is very fast. As for the content-based classification, the overhead steps up as the number of conditions increases. When 8 conditions are added the system performance degrades by 10%. We believe that 8 conditions for a content-based forwarding rule is very uncommon and that the typical number would be less than 4, in which case the server capacity loss is acceptable.

5 Related Work

Many research efforts have focused on admission control and service differentiation in Web servers [4, 13, 1, 7, 11, 14]. Current approaches typically require modifications to the application. For example, [4] and [14] add a new connection manager process to Apache, [13] adds a bandwidth manager process, and [11] requires that the web server be augmented with admission control capabilities. The work in [1] is unique in that content adaptation is achieved without modification of the web server. However, QoS-support is implemented entirely in user-space. As the authors of [1] themselves admit, a user-space approach is less efficient than a kernel implementation. For example, all requests rejected under overload are forwarded to the user-space admission controller before they can be discarded. Hence, a significant amount of kernel processing may be wasted on eventually dropped connections.

Resource containers [3] have been proposed to achieve accurate accounting and control of server resource consumption. They extend the *sockaddr* name space with a ‘filter’, which specifies a set of foreign addresses. These filters are used to assign requests from a particular client, or a set of clients, to a socket with a matching filter. Our address-based forwarding is similar, but is explicitly designed such that application code need not be changed.

Linux/RK[21, 20] provides another abstraction for resource allocation, namely, a capacity reserve. Using capacity reserves, applications can make explicit reservations

for multiple resources, and attach timeliness requirements to them. In a somewhat different approach, [6] introduces *Eclipse/BSD*, which integrates proportional share schedulers into a mainstream operating system. A special file system is implemented through which applications can specify their resource reservation. In [24], virtual services are introduced that allow resource partitioning and management. These approaches are complementary to our in-kernel classifier and user-level QoS manager in that they provide sophisticated actuators for resource allocation. For example, porting our implementation to Linux/RK will allow the QoS manager to control the capacity reserves associated with individual server processes and partition the CPU among them much more precisely.

The authors of [25] present three kernel mechanisms to support service differentiation and admission control: TCP SYN policing, prioritized listen queue and URL-based connection control. TCP SYN policing is similar to our accept rate control, except that it is implemented at the TCP layer while ours is implemented at the socket layer. URL-based connection control can be implemented in our framework as content-based forwarding in combination with other network resource control. Our approach has the advantage of being more general and can be integrated with any resource control scheme.

In [23] an in-kernel classification approach similar to ours is described. It associates several accept queues with a listen socket and classifies the connections into one of them. Requests in these queues are scheduled using a work-conserving weighted fair queuing (WFQAQ) scheduler. In contrast, we choose to forward the connection to another port once the connection is classified. How the connection is forwarded can be controlled and one connection can be forwarded multiple times, which is very powerful and flexible. Moreover, our classification mechanism is more general. For example, it can be applied to all TCP-based servers instead of just Web servers.

6 Conclusions

In this paper, we described an in-kernel architecture for traffic classification and resource management into which multiple instances of best effort servers can be plugged-in such that an abstraction of a single QoS-aware service is created. A user-space QoS manager implements feedback control mechanisms described in earlier literature, that maintain the performance of individual traffic classes at their respective set points. An experimental evaluation of this architecture shows that it is successful in providing several types of QoS guarantees in the context of several server applications.

The authors believe that their architecture solves an important problem in the deployment of QoS-aware applications; namely the prohibitive cost associated with retrofitting QoS awareness into the large volume of main-

stream legacy best effort server code. With that barrier removed, deployment of QoS-aware architectures and services can proceed at much faster rate. The approach, while successful in achieving its purpose, has some limitations. Most importantly, it works well only when the number of client classes is relatively small. This is expected to be the case in most server installations for economical and societal reasons. First, users typically like simple service plans where billing is easy to understand. Hence, a smaller number of service options is more appealing. Second, billing gets more complex as the number of classes increases. Thus, most services opt for a smaller number of service grades. The related software and documentation can be found at <http://www.cs.virginia.edu/~rz5b/software/software.htm>

Acknowledgments

The authors would like to thank Ying Lu, Chengdu Huang and the reviewers for their valuable suggestions.

References

- [1] T. F. Abdelzaher and N. T. Bhatti. Web content adaptation to improve server overload behavior. *WWW8 / Computer Networks*, 31(11-16):1563–1577, 1999.
- [2] 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*, 13(1):80–96, 2002.
- [3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, 1999.
- [4] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5), September 1999.
- [5] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.
- [6] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting quality of service into a time-sharing operating system. In *Proc. of USENIX Annual Technical Conference*, 1999.
- [7] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical Report HPL-98-119, June 1998.
- [8] M. Crispin. Rfc 2060: Internet message access protocol - version 4rev1.
- [9] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Network Operations and Management*, 2002.
- [10] W.-F. D. Group. <http://www.wu-ftp.org>.
- [11] V. Kanodia and E. Knightly. Multi-class latency-bounded web services. In *8th International Workshop on Quality of Service*, Pittsburgh, PA, June 2000.
- [12] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE J. Select. Areas Commun.*, Special Issue on Service Enabling Platform, September 1999.
- [13] K. Li and S. Jamin. A measurement-based admission-controlled web server. In *INFOCOM (2)*, pages 651–659, 2000.
- [14] 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.
- [15] 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.
- [16] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for qos guarantees and its application to differentiated caching services. In *10th International Workshop on Quality of Service*, Miami Beach, FL, May 2002.
- [17] Y. Lu, A. Saxena, and T. F. Abdelzaher. Differentiated caching services, a control-theoretical approach. In *21st International Conference on Distributed Computing Systems*, pages 615–624, 2001.
- [18] D. Mosberger and T. Jin. httpperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59–67. ACM, June 1998.
- [19] U. of Washington. <http://www.washington.edu/imap/>.
- [20] S. Oikawa. Linux/rk: A portable resource kernel in linux. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec 1998.
- [21] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *IEEE Real Time Technology and Applications Symposium*, pages 111–120, 1999.
- [22] 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.
- [23] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy. An observation-based approach towards self-managing web servers. In *10th International Workshop on Quality of Service*, Miami Beach, May 2002.
- [24] J. Reumann, A. Mehra, K. G. Shin, and D. Kandlur. Virtual services: A new abstraction for server consolidation. In *Proc. of USENIX Annual Technical Conference*, pages 117–130, 2000.
- [25] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proc. of USENIX Annual Technical Conference*, pages 189–202, 2001.
- [26] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *22nd International Conference on Distributed Computing System*, Vienna, Austria, July 2002.