Kernel Support for the Event-based Cooperation of Distributed Resource Managers

Christian Poellabauer and Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{chris, schwan}@cc.gatech.edu

Abstract

Multimedia and real-time applications require end-to-end QoS support based on the cooperative management of their distributed resources. This paper introduces a kernel-level facility, called Q(uality)-channel, which provides a flexible fabric with which Operating System developers can efficiently implement policies for managing the resources used by distributed applications. The inherent complexity of resource management in large-scale distributed applications is addressed by the event-based cooperation over asynchronous and anonymous Q-channels. Q-channel creation and operation (such as resource monitoring and adaptation) is hidden behind standard communication mechanisms, i.e., transparent to applications and thereby offering quality of service support with minimal application involvement. However, an application can influence the manner in which its operation is affected by Q-channels, such as permitting resource managers to customize individual application-level communications by dynamically installing Q-filters into data streams. Such Q-filters can also be parameterized, thereby permitting continuous manipulations of application-level communications based on requirement and performance information dynamically collected from event publishers and subscribers.

1. Introduction

Multi-party interactive multimedia (MIM), telepresence, and remote sensing are collaborative applications that cannot function without dynamic management of the underlying resources. To attain the Quality of Service (QoS) required by applications, distributed *resource managers* must dynamically allocate the resources required, monitor the QoS received, alter resource allocations when necessary, and even perform run-time adaptations of applications, middleware, and operating or communication systems [1, 18]. Further, since achieving and maintaining QoS for distributed applications is an end-to-end issue [12], resource managers must cooperate in these endeavors, so that QoS guarantees are applied to the entire flow of data, e.g., from a server to its clients.

Previous research has used middleware to 'bind' the multiple machines, applications, and resource managers that implement QoS provisioning, resource management, and performance differentiation for distributed applications and platforms, sometimes enhanced by operating system (OS) extensions on individual machines [7, 10]. However, there remain some open problems with such middleware-based approaches, including with our own previous work on the management of distributed radar sensors [18]. First, user-level approaches must rely on the voluntary participation of applications in QoS management, thereby enabling non-participants to threaten the guarantees made to participants. Second, the granularity at which resources can be managed and the fidelity of such resource management are not always sufficiently high to meet applications' performance needs. Reasons for this include (1) inappropriate delays of QoS management [19], often aggravated by the overheads of application-level QoS managers' interactions with the system-level mechanisms they must use to understand current resource usage and availability, and (2) inappropriate interfaces provided by operating systems that require managers to poll for changes in resource state or make unnecessary resource reservations, as also noted in [7, 15].

Q-channels: a system-level fabric for distributed resource management. Previous QoS approaches focused on end-to-end QoS management in simple client-server systems. However, large-scale multimedia applications such as teleconferencing or collaborative applications require the management of resources across a large number of clients and servers and with many individual streams that exhibit performance dependencies due to synchronization require-

ments. To address the complexity of QoS management in these applications, we have developed Q-channels, a kernel-level service that permits multiple, distributed, kernel- or user-level resource managers to cooperate in the provision of QoS management to applications:
(a) Q-channels provide a flexible, system-level 'fabric' on which distributed resource management policies may be implemented. Specifically, distributed resource managers perform tasks like QoS negotiation and adaptation via events over kernel-level event channels.
(b) QoS management can be transparent to applications, since they do not need knowledge of or direct access to the Q-channels associated with them.
(c) Applications can also explicitly interact with the quality management performed on their behalf, if desired, by using Q-attributes to share quality information with resource managers, and by using Q-filters to control the exchange of quality information across machines and to link their application-level communications with Q-channel-based resource management actions. In this paper we are concerned with the support for the communication between distributed resource managers in an off-the-shelf OS and not with any QoS adaptation algorithms in particular.

Managing QoS via Q-channels. To visualize the functionality of Q-channels, consider a video conferencing application, where each host exchanges a large number of video and audio streams. A resource management system has to ensure that all of these streams achieve their respective QoS targets. Therefore, user- or kernel-level resource managers have to cooperate with other resource managers of all participating hosts. Functions such as QoS specification, negotiation, and adaptation have to be performed in conjunction with all or a subset of the participating hosts. Finally, such cooperation has to reflect certain relationships across streams, e.g., to ensure lip synchronization for audio and video streams. In our approach, when a host joins a distributed multimedia application, the host's resource managers automatically create and subscribe to an associated Q-channel. The application does not require knowledge about this channel or the fact that application-level actions like message sending or receipt may lead to resource manager interactions, as per the resource management policies implemented (see [21] for experiments with sample policies). Alternatively, applications may explicitly interact with resource managers, e.g., to specify desired service qualities or to understand resource availability. Such interactions use *Q(uality)-attributes*, which are customizable lists of attributes that can be passed between applications and resource managers. Finally, applications can further refine the resource management actions taken on their behalf by specifying functions (residing in user- or kernel-space) (1) to alter their inter-machine data communications or (2) to dynamically change the functionality of the associated Q-channel-based resource management services. Useful functions include those that (i) filter user-level communications based on current end user needs [5], or that (ii) compute the utility [20] or payoff [8] of certain communications, or actions, or changes to their qualities from an application perspective (thereby providing to kernel-level resource managers application-level interpretations of 'quality' [21]), or that (iii) implement event filters that eliminate unnecessary transmissions of QoS management information.

Q-channels are intended to be used with a variety of communication mechanisms. Our current work uses a QoS-enhanced variant of standard Berkeley sockets (called *Q-sockets*), and an event service comparable to the CORBA Event Service. More specifically, the same event service that serves as the building block for Q-channels can be used by applications as a means to exchange multimedia data.

2. Architecture of Q-Channels

2.1. Kernel-level Quality Management

As stated earlier, a *Q-channel* is an OS service that provides communication paths for the exchange of control information between resource managers in a distributed system. Typically, such a channel is created or subscribed to when some user-level communication is established. The intent is to enable end-to-end management of QoS such that the transmission of application-level data is associated with, but also separated from the kernel-level exchange of control or adaptation information. Control information is represented by quality events [21] exchanged between remote and local resource managers.

Though Q-channels can be established and used from both user-level and kernel-level resource managers, this paper focuses on kernel-level quality management for several reasons. First, our past work has shown that fine-grain, kernel-level resource management can provide applications with benefits not derived from coarser-grain, user-level QoS management. Specifically, excessive delays or overheads experienced by dynamic adaptations due to slow performance monitoring or decision-making by resource managers can negatively affect or even negate the advantages of run-time adaptation for real-time applications [19]. Furthermore, OS kernels or network services often present interfaces to application-level resource managers that necessitate repeated kernel calls in order to determine the resources available for allocation to certain application tasks. In comparison, within OS kernels or in network services, it is straightforward to inspect and manipulate the data structures involved in resource allocation. Finally, OS kernels can enforce constraints on the delays experienced by applications when they are informed about changes in their allocations, or when they must be adapted to conform to new QoS requirements or resource availabilities, whereas application-level resource managers may be at the mercy of CPU schedulers.

Q-channels are intended to be used in conjunction with a variety of communication mechanisms, ranging from sockets to event services. As a result, the creation and operation of Q-channels is performed whenever the application uses such a communication mechanism and is typically hidden from the application. In this paper, we are associating Q-channels with a data event service which applications use for their distributed communications. Our motivation for using event channels in data communications is twofold. First, event and publish/subscribe services [6, 16] have become prevalent in distributed applications that range from virtual reality and avionics to support for mobile users [4, 9]. Second, recent research contributions have pointed out the applicability of event services for continuous media transfers [2, 11]. The event service used to demonstrate such communication for distributed applications, called KECho, is also used to implement Q-channels themselves. However, any other event service could be similarly designed to cooperate with Q-channels. KECho differs from previous approaches through its implementation as a kernel extension to the Linux operating system. As with middleware event services, KECho allows applications to subscribe to anonymous and asynchronous *event channels* as publishers or subscribers of events. However, KECho further allows kernel threads to participate in such event communication, which may be used to implement distributed operating systems, remote monitoring systems, or load balancing mechanisms [13]. Therefore, KECho is a convenient medium for implementing Q-channels and demonstrating their use. In ongoing work we are developing Q-sockets, which are communication sockets enhanced with the ability to specify quality attributes for communications and manage distributed communications via associated Q-channels.

A resource management model. The model supported in this paper may be described as follows. First, an application-level process establishes a communication path with some remote application process, using the KECho data event service (called a *data channel*). Second, a kernel-based QoS monitor (see Figure 1) periodically collects information about the resource allocations and availability on the machines used by the distributed application, or about the QoS they achieve. This information is distributed via *monitoring events* published on a Q-channel associated with the KECho-based data communication path. Third, a QoS controller receives and acts upon such monitoring events by (a) adjusting resource allocations locally and (b) issuing *quality events* to the Q-channel, which forces remote QoS controllers to reconsider and adapt their resource allocations.

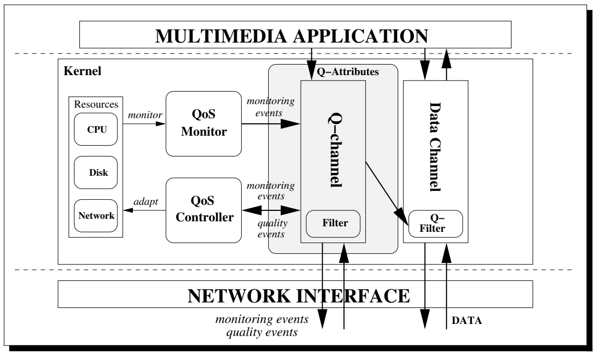

Figure 1. Q-channel architecture.

Applications can affect resource allocations by issuing Q-attributes, for example, to state their resource needs in a *QoS specification*. Finally, event channels can take advantage of filters, which can customize the event traffic to the sink's specific needs by dropping, aggregating, or preprocessing of events.

The goals of this architecture are: (1) to support OS developers in the design of resource management architectures for large-scale distributed applications (**reduced complexity**), (2) to enable fine-grain adaptation by supporting in-kernel resource management mechanisms (**improved performance**), (3) to hide Q-channel setup, operation, and tear-down from applications (**transparency and ease of use**), and (4) to permit sophisticated applications to customize and influence QoS management by offering attributes and filtering methods (**QoS-awareness**).

2.2. KECho

KECho [13] is implemented as a set of kernel-loadable modules for Linux 2.4.17. Its main components are a publish/subscribe mechanism, connecting both user-level applications and kernel-level threads on remote hosts, and a lightweight API, connecting user-level applications and kernel services on the same host, termed ECalls. ECalls supports real-time and multimedia applications by (1) delivering events between applications and kernel services in a timely fashion, (2) enabling both to efficiently share relevant information, and (3) influencing process scheduling in response to the receipt of new events or pending events. It offers several methods of event generation and event handling, thereby giving event sources and sinks the flexibility to select a method that is appropriate for the specific needs of an application. More details about the ECalls interface can be found in [15].

KECho is a group communication tool comparable in functionality to event communications in the CORBA Event

Service [4], but with performance approximating that of socket-based communications. It is a kernel extension that allows for asynchronous and anonymous event communication within a kernel and across remote kernels. A process (or kernel thread) can subscribe to a KECho event channel without the need to know the identities and number of other subscribers. Event channels are implemented fully distributed and events are transmitted directly between peers.

2.3. Q-Attributes and Parameterizable Filters

Q-attributes are a way to pass QoS information between user-level applications and kernel-level resource managers. KECho provides functions that allow applications and resource managers to add, delete, or modify attributes. Filtering in continuous media streams allows the customization of such streams to the specific needs of clients. For example, high-performance end-systems can receive full video quality, while other end-systems receive a reduced quality version. Filter functions can manipulate streaming media in a variety of ways, including *frame dropping* (e.g., dropping B- and P-frames from an MPEG video stream), *compressing* (i.e., the choice of compressing standard influences the actual bit stream size), and *dithering* (i.e., reducing the color depth of images).

Both data channels and Q-channels are based on the KECho event service, which means that both can deploy filters. Filters for Q-channels are inserted by the developer of the resource management system and are typically limiting the event traffic between subscribers to reduce computation and communication overheads. As an example, the QoS monitor of a client-server interaction may wish to send a monitoring event to the QoS controller of the server only, therefore excluding other clients. However, the QoS controller of the server may wish to send quality events to all clients or a subset thereof. Further, such filters can reduce overheads by aggregating or dropping QoS events. For instance, depending on the 'speed' of the communication on the data channel, a filter could control how frequently monitoring events are being issued. On the other hand, filters for the data channel (Q-filters) are inserted by the application itself and are parameterizable via Q-channels. That is, a video server could send frames at the rate of its fastest client, but the filter ensures that other, 'slower' clients receive lower frame rates by suppressing the transmission of certain data events. A Q-filter is parameterizable in the sense that QoS events received on the Q-channel can influence parameters of the filter in the data channel. As an example, a filter on the data channel could change the color depth or resolution of images, where events on a Q-channel can decide the desired depth and resolution, depending on current resource availabilities.

3. Distributed QoS Management

3.1 Q-Channel Setup

When an application creates a new data channel, the underlying resource managers create a hidden Q-channel and subscribe to this new Q-channel. When an application opens an existing data channel, the underlying resource managers open and subscribe to the corresponding Q-channel. While a data channel connects instances of a distributed application, a Q-channel connects the underlying corresponding resource managers of the same nodes the instances of the application are running on.

In the approach investigated in this paper, both applications and resource managers use the KECho event service. The APIs for both scenarios, that is, user-level applications operating a data channel and resource managers operating a Q-channel, are identical. That has been achieved by exporting the same function names as (1) system calls for applications and (2) kernel symbols for resource managers. The following code shows an example of how applications and resource managers use KECho for event-based communication:

```
1: comm_mgr_id = CManager_create();
2: ctrl_id = ECho_CM_init(comm_mgr_id);
3: if (main_host == TRUE)
4:   chan_id = EChannel_create(control_id);
5: else
6:   chan_id = EChannel_open(ctrl_id, chan_name);
7: handle = ECsource_subscribe(chan_id);
8: (void) ECsink_subscribe(chan_id, hndlr_func, NULL);
```

Lines 1 and 2 initialize the *communication manager*, which is the part of KECho responsible for connection management. One application has a prominent role (the *channel manager*) in that it creates an event channel (line 4), which then can be opened by other applications (line 6). All applications joining or leaving an event channel have to do so by contacting the channel manager. A *registry server*, which runs on a host known to all applications, can be used as a directory service for all open event channels. After a channel has been created/opened, an application can register as a publisher – or source – (line 7) and as a subscriber – or sink – (line 8).

3.2. Operation of Q-Channels

The following code depicts the manner in which an application or kernel thread submits events and polls for incoming events:

```
while (1) {
  ...
  ECsubmit_event(handle, &event, event_length);
  CMpoll_network(comm_mgr_id);
  CMtask_sleep(time);
  ...
}
```

Events can be issued (line 3) or polled for (line 4) at any time. Using the ECalls mechanism, an application can also express the desire to be notified of events asynchronously, e.g., with signals, instead of using the *CMpoll_network* function call.

If an application wishes to use Q-filters, the subscription has to identify these filters by either (i) their function names or (ii) a unique character string for the kernel module that implements these filters. In other words, applications can provide the desired filter functions either within the application code, or as kernel extensions in a separate module. Though the first approach appears to be more attractive to application programmers, it has to be kept in mind that user-level function invocations from within the kernel (1) have slightly higher overhead than pure in-kernel functions and (2) can pose a protection threat to the OS or other applications. Our current work includes the dynamic generation of filter functions within the kernel, based on a subset of the 'C' programming language.

3.3. Run-time Quality Management

Resource management with Q-channels is based on the exchange of events. After a Q-channel has been created, resource managers are free to join and leave the channel and to participate in the exchange of quality events. Our approach introduces several desirable characteristics into a resource management system:

(a) Reduced complexity: The coordinated management of distributed resources can be complicated in large-scale applications, where (1) multiple data sources and sinks communicate, (2) data streams have relationships such as synchronization, (3) data streams reserve resources together (resource sharing), (4) resources or applications can migrate, or (5) the system is highly dynamic, that is, applications and their quality of service requirements can change at any time. Q-channels address this complexity by tying the creation of event channels for resource management with the creation of data channels. QoS 'engineers' can therefore rely on Q-channels to automatically interconnect the resource managers along the path of a data stream.

(b) Asynchronism: Event communication in KECho is asynchronous, i.e., event sources publish events on an event channel without waiting for a response. This loose coupling of event publishers and subscribers supports the dynamic behavior of the multimedia system (e.g., resource migration).

(c) Anonymity: Channel subscribers are unaware of the identities or even number of other subscribers. This facilitates the dynamic joining and leaving of subscribers or the dynamic migration of subscribers and resources.

(d) Performance and granularity of adaptation: Kernel-level resource management can provide applications with fine-grain QoS adaptations by avoiding excessive delays or overheads caused by costly system calls or signals. Further, within OS kernels or in network services it is straightforward to monitor and adapt data structures and kernel services. Because of these reasons, we (a) focus in this paper on the kernel-level implementation of resource management (although user-level implementations are equally possible), and (b) entirely implement the Q-channel fabric within the kernel.

(e) Transparency and ease of use: Q-channels reduce the need for application programmers to change their codes to permit QoS management or to learn how to use complex new APIs for such management. This is because the creation and management of Q-channels is completely hidden behind traditional communication mechanisms, like standard sockets or event channels. This also allows resource management systems to enforce the participation of applications in a QoS management without even their knowledge, if desired.

(f) QoS-awareness: Though transparency is an important goal in the design of Q-channels and desirable for legacy applications, newer applications may want to actively participate in the management of the system's resources. For this purpose, Q-channels implement mechanisms that allow applications to specify or retrieve QoS information with Q-attributes. Further, filters can be inserted dynamically into data paths in KECho, which can be parameterized dynamically via information from a Q-channel.

Resource management groups. Applications or resource managers may be grouped via Q-channels for many reasons, reflecting the fact that these applications and their data streams have to coordinate the use of resources to avoid interference or to explicitly trade off quality, as with MIM applications that raise the quality of one stream at the cost of all other streams in the same group. An explicit example is teleconferencing, where the stream to a chair of a teleconference might be considered of higher importance. Groups can express the need for stream synchronization, as with **inter-sink synchronization**, where the replay of one or more media streams has to be synchronized at several receiver hosts. By adding these hosts' resource managers to the same group, they can directly inform each other about variations in achieved QoS and trigger changes in resource allocations. Similarly, **intra-media synchronization**, also called **rate control**, aims at the proper playback of streams at the receiver. It addresses synchronization between the sender and receiver of a single stream. Finally, **inter-media synchronization** addresses relationships between different media streams, e.g., for resource sharing or lip synchronization of video and audio [17].

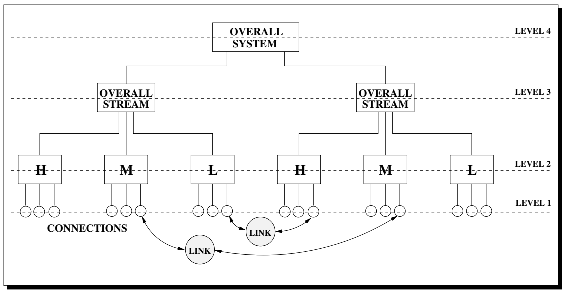

Figure 2. Hierarchical resource management.

Hierarchical resource management. Hierarchical resource management facilitates the control of the resource consumption of the whole system: A *QoS tree* can be generated where a parent in this tree manages the *group-QoS* of all its children without detailed knowledge about the resource needs of the individual children. On the other hand, the children can manage individual connections without knowledge of the resource needs of other children. Consider a tele-teaching application, where the source of a stream could be a teacher, and the sinks are the students. QoS specification can be done by a sink (e.g., a student desires a certain frame rate) or by the source (the teacher determines the frame rate the students will receive). In this example, we choose a combination of both, that is, the source offers data streams at three different quality levels: **high** (20 - 25 frames per second or fps), **medium** (15 - 20 fps), and **low** (10 - 15 fps). In such a case, cooperation and adaptation can occur at different levels, as shown in Figure 2. At level 1, a QoS manager tries to maintain the data rate at constant values by adjusting resource allocations like CPU and network bandwidth for each individual connection. At level 2, the QoS manager tries to adjust resource allocations within a role, e.g., to ensure that all individual connections within a role receive the same quality. At level 3, a QoS manager distributes the available resources among several streams and manages the per-stream quality of service. Finally, at level 4 all resources in the whole system are considered and distributed among streams. Further, **links** can be defined that signify relationships, e.g., an audio and a video stream can be linked to ensure that a drop in the quality of one stream affects equally the other stream.

4. Multipoint Feedback Adaptation

4.1 Experimental Setup

The example used in this paper is that of a feedback-based adaptation mechanism, i.e., a server streams data to a client. A QoS monitor watches the resource allocations or the achieved quality of service at the client and issues monitoring events to a Q-channel, which are received by the QoS controllers at the server. This simple scenario becomes a complex problem if a stream is received by a large number of clients, i.e., a large number of QoS monitors issue monitoring events back to the server-based QoS controller. A common problem of large-scale feedback-based systems has been described as *reply implosions* [22] in the literature, where a server solicits information from clients and all clients reply almost simultaneously to the server.

To evaluate the performance and functionality of Q-channels, we have modified the *vic* video conferencing tool such that it operates on top of the KECho event service instead of standard sockets. On the OS side, we implemented a resource manager consisting of a QoS monitor and a QoS controller. Whenever user *A* starts an instance of vic, a data channel is set up. Transparently to vic (and the user), the QoS monitor and the QoS controller subscribe to a newly created Q-channel. Once a second user *B* requests to receive a video stream from *A*, the resource managers of *B* subscribe to the Q-channel. If both *A* and *B* decide to submit and receive each others streams, their resource managers subscribe to two different Q-channels, one created by *A* and managing *A*'s data stream and one created by *B* and managing *B*'s data stream.

The task of the QoS monitor is to monitor the rate of incoming packets over the data channel. This is done by installing a filter into the data channel, which keeps track of the received images from each participants. The QoS monitor periodically submits a monitoring event. Another filter in the Q-channel makes sure that only the resource manager at the data source will receive such monitoring events. On the other hand, upon receipt of a monitoring event, the QoS controller at the data source reconsiders its own resource allocations for this stream or issues a quality event, which is submitted to either (a) one specific resource manager, e.g., the same that issued the monitoring event, or (b) all resource managers. The QoS controllers on the sinks then reconsider their resource allocations upon receipt of a quality event.

4.2. Event Submission and Handling

As argued in previous sections of this paper, kernel-level implementations of resource management can profit from limited calls across protection boundaries, e.g., with system calls. Kernel services and data structures are directly accessible to the resource management mechanism, permitting fine-grain adaptations. KECho achieves further performance gains by ensuring that new events arriving on a channel are dispatched to the corresponding threads with minimal delays. This is achieved by the cooperation of KECho with the CPU scheduler such that kernel threads with pending events are given preference over other (user-level) processes [14].

The following experiments have been performed on a cluster of 8 Quad-Pentium Pros with 200MHz each, 512 MB RAM, running Linux (version 2.4.17). Figure 3 compares the cost associated with event submission in the kernel-level event service, KECho, with the event submission of a similar user-level implementation, called ECho [3]. Both ECho and KECho use TCP, although work

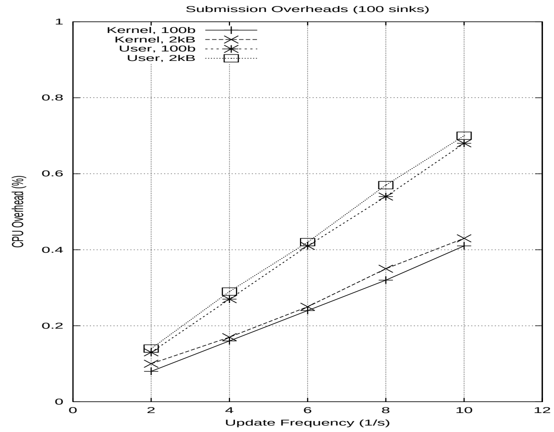

Figure 3. Event submission overheads.

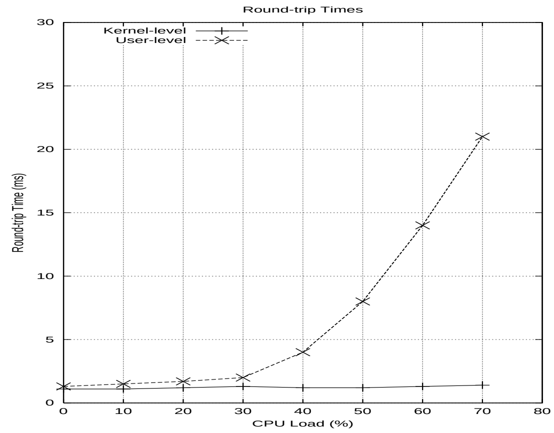

Figure 4. Round-trip delays.

is in progress to add UDP enhanced with a reliability layer to both. In these measurements a source transmits an event of size (i) 100 bytes and (ii) 2 kBytes to 100 sinks with a variable **update frequency** (number of transmissions of events per second) in the range of 2-10. The chosen data sizes reflect the sizes of monitoring and quality events used in our implementations, which are typically only a few hundred bytes large. With an update frequency of 10 events per second, 1000 TCP packets have to be sent out each second. In this situation, the CPU overhead of event submission in KECho is slightly above 0.4%, compared to 0.7% for ECho (excluding protocol processing). The graph shows further that this overhead increases minimally with larger data sizes. Figure 4 shows the measured **round-trip times**: here we measure the time beginning from the submission of an event at the server until receipt of a reply event from the client. The event size is 100 bytes and a **disturber process** is run such that it consumes CPU bandwidth from 0 to 70%. It can be seen that the kernel-level implementation shows constant round-trip times independent from the CPU load, whereas the user-level implementation suffers significant increases in round-trip-times above CPU loads of 30%. This is due to KECho's ability to coordinate its activity with the CPU processor such that handler functions are invoked as soon as possible after event arrival to increase the responsiveness of the resource management system, while maintaining the real-time requirements of all running tasks [14].

4.3. Multipoint Feedback Control

We consider a point-multipoint scenario, i.e., a source streams data to several sinks, using the 'roles' defined earlier in the paper (high: 20-25 fps, medium: 15-20 fps, low: 10-15 fps). A sink uses Q-attributes to select a quality level depending on its own capabilities, the resource management mechanism tries to supply all clients with the best possible quality within their roles. If this quality cannot be sustained, it is desirable to reduce the qualities of the 'low' role sinks first, then of the 'medium' role sinks, and finally of the 'high' role sinks.

In this particular example (1 source, many sinks), monitoring events are only issued by sink-based QoS monitors and directed to the source-based QoS controller. On the other hand, quality events are only issued by the source-based QoS controller to all sink-based QoS controllers. Q-channel filters make sure that sink-issued monitoring events are propagated to the source only, and that source-issued quality events are propagated to all sinks or to a subgroup of them.

The resource controlled for our application is network bandwidth. The vic video conferencing tool is executed as a real-time process in the SCHED_RR (round-robin) queue with priority 1. To limit and adapt communication bandwidth, we use Class-Based Queuing (CBQ) to define classes with a bandwidth of 200 Kbit each, and we use

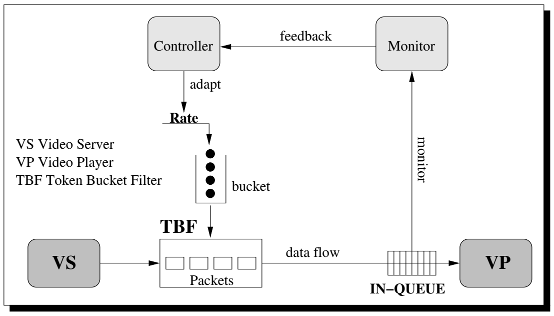

Figure 5. Feedback mechanism.

the Token Bucket Filter (TBF) algorithm to transmit packets. This algorithm has been modified such that the QoS controller is able to influence the rate of token generation and therefore, influence the rate of packet transmission.

Stream management. Figure 5 shows the mechanism for one stream: a video server streams packets through a token bucket filter to the video player, where a QoS monitor watches the packet arrival at the video player and feeds this information back to a QoS controller at the server. The controller is then able to adjust the rate at which tokens are added to the bucket, and therefore the rate at which packets are sent over the event channel. Figure 6 shows the achieved frame rates of 3 streams, one with role 'high', one with role 'medium', and the last one with role 'low'. We first start all 3 streams simultaneously, and the rate

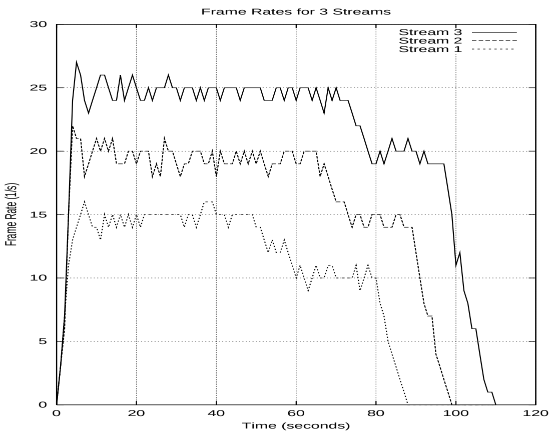

Figure 6. Frame rates for 3 streams.

control achieved with the token bucket filter ensures that all three streams achieve the highest frame rate in their desired ranges (i.e., roles). After 40 seconds we start a *disturber process*, which transfers files at increasing rates from the server to the clients, thereby causing the network to get saturated. After 50 seconds, the QoS controller is not able to further sustain the frame rates and it starts reducing the frame rate of the 'low' role stream, while sustaining the rates of the two other streams. After the 'low' role stream reaches its minimum (10 fps), the 'medium' role stream suffers a drop in achieved frame rate. Finally, after both the 'low' and 'medium' role streams have dropped to their minima, the 'high' role stream is reduced to its minimum value of 20 fps. After 80 seconds, the controller is not able to sustain all of the desired frame rates and starts reducing the rate of the 'low' role stream until it reaches 0. It continues this process until, after 110s, all three streams have stopped. This experiment shows further the implementation of a simple QoS policy that manages several media streams and their relationships to each other.

Reply implosion. The problem of receiving a large number of requests almost simultaneously is referred to as reply implosion. Solutions to this problem include *probabilistic replies*, *statistical probing*, and *randomly delayed replies* [22]. The advantage of our event-based approach is that a push-based mechanism is used, which avoids this problem to the most part. This is because QoS monitors submit their monitoring events to a server-based QoS controller independently at certain intervals (e.g., every 500ms). However, it is still possible, particularly in large-scale applications, that many monitoring events are issued almost simultaneously. In Table 1 we analyze the **event distribution** for 1 second in the same setup as described in Section 4.2 (i.e., 1 server and 100 clients). The first column

Table 1. Event distribution.

	w/o adapt.		w/ adapt.	
Time	t=10	t=100	t=10	t=100
0–100ms	9	8	7	9
100–200ms	14	16	9	10
200–300ms	14	12	13	11
300–400ms	7	7	16	10
400–500ms	11	12	7	11
500–600ms	14	13	5	10
600–700ms	2	6	11	9
700–800ms	11	8	14	10
800–900ms	5	7	7	10
900ms–1s	13	11	11	10

shows the event distribution measured after 10 seconds of running the experiment and displays the number of received monitoring events at the server per 100ms. With 100 clients and a update frequency of 1 (i.e., each client sends exactly 1 monitoring event every second), the ideal distribution in our simplified scenario would show 10 events per 100ms. The event distribution is measured again 90 seconds later, showing a similar distribution with minor deviations due to timing and measurement errors. However, it can be seen that the distribution ranges from 2 to 16 events. In a second experiment we now implement an adaptation algorithm. The algorithm determines the number of clients N and the number of *expected events* per 100ms: $n = N/10$. For every time interval i of 100ms, the QoS controller counts the number of actually received events (r_i). For all time intervals i, the number of *excess events* is computed ($e_i = r_i - n$ if $r_i > n$), and then e_i randomly chosen clients from interval i receive a *delay request* with the next quality event. This delay request forces the client (i.e., the QoS monitor) to submit the next monitoring event 100ms later. Over time, this succeeds in distributing the issuance of monitoring events more evenly as can be seen in the last row of Table 1.

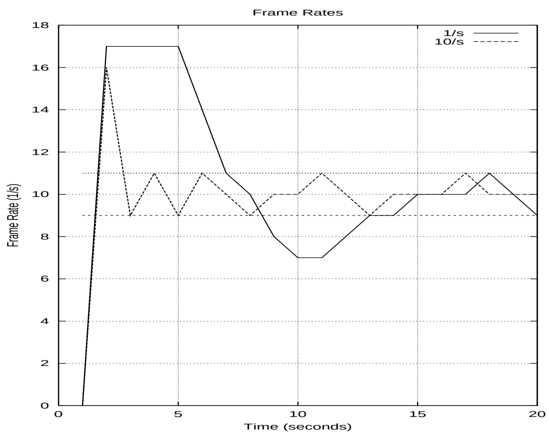

Figure 7. Handler invocation frequency affects the dynamics of the system.

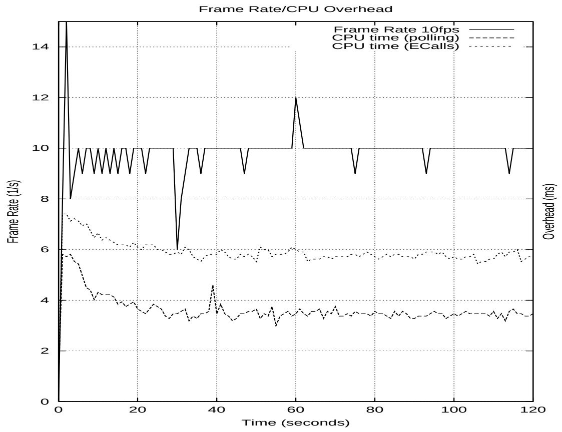

Figure 8. Event receipt with ECalls.

Polling versus ECalls. Furthermore, the frequency of QoS monitor and QoS controller invocations influences both the granularity of adaptations achievable as well as the overhead of adaptations. Figure 7 shows the behavior of a video stream with a target frame rate of 10 fps$\pm$1. In the first graph we invoke the QoS controller (a) once per second and (b) ten times per second to poll for possibly pending monitoring events. In the case of a frequency of only 1/s, the video stream needs more than 12s to reach its target frame rate. When we run the QoS controller ten times as often, the stream needs approximately 3s. However, KECho offers the ECalls interface, which makes polling unnecessary by invoking the QoS controller immediately at event arrival. Figure 8 shows a video stream operated at 10 fps, this time ECalls ensures the timely handling of incoming monitoring events. The graph also shows the CPU consumption of the QoS controller function when we (a) poll for events ten times per second and (b) use ECalls instead. Without ECalls, the CPU consumption is approximately twice as much as with the support of ECalls because the QoS controller is run *only* when there are monitoring events pending.

Q-Filters. Q-filters are application-specific event channel filters, parameterizable through the QoS management system built on top of Q-channels. Such filters can perform tasks such as down-sampling, color depth reduction, or even dropping of images to reduce network load. In other words, additional computation at the server is introduced to reduce the required network bandwidth between server and client or to reduce the required computation at the client (e.g., visualization of images with fewer colors or smaller size). However, a resource manager can activate or affect such filters without the active involvement of the application, further improving the granularity of adaptations. This is particularly useful for transient overload situations, where frequent application adaptations can be counterproductive, however, the resource manager can simply change the parameters of the Q-filter with less overhead. Consider a Q-filter that simply drops certain frames (e.g., B- and P-frames of an MPEG stream). If a resource manager considers it necessary to change the number of frames transmitted to ease the network load temporarily, it can simply change a parameter of the Q-filter, which takes effect immediately. However, if the resource manager decides instead to notify the application to let itself perform this adaptation, the additional delay can be significant, particularly when the system is highly loaded. As an example, we implemented a filter that drops frames if the network is overloaded. When the client-side QoS monitor detects a network overload, it issues a monitoring event directed to the server-side QoS controller. The controller then adjusts a Q-filter parameter that decides if and when a frame is being dropped. The overhead from the receipt of the monitoring event until the parameter is adjusted is in the range of 10μs. However, if we choose to send a signal to the application notifying it about the overload, the overhead from the receipt of the monitoring event until the application changes the rate of frame creation ranges from 50ms in a system with about 80% CPU load to several hundred milliseconds with CPU load $> 100\%$.

5. Conclusions and Future Work

To address the complexity of QoS management in large-scale distributed multimedia applications, we have developed Q-channels, a kernel-level fabric that permits multiple, distributed, kernel- or user-level resource managers to cooperate. The goals of Q-channels are: (a) Reduced complexity in the development of new resource management architectures by offering an anonymous and asynchronous event service. (b) High performance and fine-grain adaptations by supporting lightweight in-kernel resource management. (c) Transparency to applications, i.e., setup, operation, and tear-down are hidden from the application. (d) Application-specific QoS customization for sophisticated applications by using Q-attributes and Q-filters.

Our future work includes the replacement of TCP with a reliable version of UDP, the support of deadlines and priorities associated with events, and the dynamic in-kernel generation of Q-filters.

References

[1] T. F. Abdelzaher and K. G. Shin. QoS Provisioning with qContracts in Web and Multimedia Servers. In *IEEE Real-Time Systems Symposium*, pages 44–53, 1999.

[2] D. Chambers, G. Lyons, and J. Duggan. Stream Enhancements for the CORBA Event Service. In *Proc. of the 9th ACM Multimedia Conference*, October 2001.

[3] G. Eisenhauer, F. Bustamante, and K. Schwan. Event Services for High Performance Computing. In *Proceedings of High Performance Distributed Computing (HPDC)*, 2000.

[4] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-time CORBA Object Event Service. In *Proceedings of the OOPSLA '97*, October 1997.

[5] C. Isert and K. Schwan. ACDS: Adapting Computational Data Streams for High Performance. In *Intl. Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[6] K. Jeffay. The Real-Time Producer/Consumer Paradigm: A Paradigm for the Construction of Efficient, Predictable Real-Time Systems. In *Selected Areas in Cryptography*, pages 796–804, 1993.

[7] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU Reservations: Efficient Predictable Scheduling of Independent Activities. In *Proceedings of the* 16^{th} *ACM Symposium on Operating System Principles*, pages 198–211, October 1997.

[8] R. Kravets, K. Calvert, and K. Schwan. Payoff-Based Communication Adaptation based on Network Service Availability. In *IEEE Multimedia Systems*, 1998.

[9] C. Ma and J. Bacon. COBEA: A CORBA-Based Event Architecture. In *Proc. of the 4th USENIX Conference on Object-Oriented Technologies, Santa Fe, NM*, April 1998.

[10] C. W. 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*, 1994.

[11] S. Mungee, N. Surendran, and D. C. Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. In *Proc. of the 32nd Annual Hawaii Intl. Conference on System Sciences*, 1998.

[12] K. Nahrstedt and J. Smith. The QoS Broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[13] C. Poellabauer, K. Schwan, G. Eisenhauer, and J. Kong. KECho - Event Communication for Distributed Kernel Services. In *Proc. of the Intl. Conference on Architecture of Computing Systems (ARCS'02)*, April 2002.

[14] C. Poellabauer, K. Schwan, and R. West. Coordinated CPU and Event Scheduling for Distributed Multimedia Applications. In *Proc. of the 9th ACM Multimedia Conference, Ottawa, Ontario, Canada*, October 2001.

[15] C. Poellabauer, K. Schwan, and R. West. Lightweight Kernel/User Communication for Real-Time and Multimedia Applications. In *Proceedings of 11th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, June 2001.

[16] R. Rajkumar, M. Gagliardi, and L. Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proc. of the 1st IEEE Real-Time Technology and Applications Symposium*, May 1995.

[17] S. Ramanathan and P. Venkat Rangan. Feedback Techniques for Intra-Media Continuity and Inter-Media Synchronization in Distributed Multimedia Systems. *The Computer Journal*, 36(1):19–31, 1993.

[18] D. Rosu, K. Schwan, and S. Yalamanchili. FARA - A Framework for Adaptive Resource Allocation in Complex Real-Time Systems. In *Proc. of the 4th IEEE Real-Time Technology and Applications Symposium*, June 1998.

[19] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. In *Proc. of the 18th IEEE Real-Time Systems Symposium (RTSS)*, December 1997.

[20] S. Shenker. Fundamental Desing Issues for the Future Internet. *IEEE Journal on Selected Areas in Communications*, 13(7), 1995.

[21] R. West and K. Schwan. Quality Events: A Flexible Mechanism for Quality of Service Management. In *Proc. of the 7th IEEE Real-Time Technology and Applications Symposium*, May 2001.

[22] A. Youssef, H. Abdel-Wahab, and K. Maly. A Scalable and Robust Feedback Mechanism for Adaptive Multimedia Multicast Systems. In *Proc. of 8th Intl. Conference on High Performance Networking*, 1998.