Cooperative Run-time Management of Adaptive Applications and Distributed Resources

Christian Poellabauer
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
chris@cc.gatech.edu

Hasan Abbasi
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
habbasi@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
schwan@cc.gatech.edu

ABSTRACT

This paper presents *Q-fabric*, which is a set of lightweight, kernel-level abstractions for cooperative, distributed resource management and system/application adaptation. The basis of Q-fabric is its kernel-level, anonymous, asynchronous event service. With this mechanism, (1) applications can monitor and manage the local and remote resources they are using, (2) system-level resource managers can customize their actions to meet the needs of individual applications, and (3) policies can be developed that combine application adaptation with distributed resource management. Results presented in this paper demonstrate the Q-fabric's ability to effectively adapt and manage the resources of a distributed multimedia application. In this application, media streams are adapted at application-level via data down-sampling, and their resources are managed at system-level (e.g., task scheduling) to cope with run-time variations in resource availability. The Q-fabric is implemented as kernel modules on standard Linux platforms.

Keywords

QoS management, adaptation, event service, OS services

1. INTRODUCTION

Background. Distributed multimedia applications require the dynamic management of underlying computer resources, including CPUs, networks, disks, and sensor/display devices. For instance, remote sensing applications need sufficient network bandwidth to receive images with the latencies they require, and they need sufficient memory and CPU cycles to process and display these images when needed by end users. In all such cases, the resource managers located at the hosts involved in a distributed multimedia application have to dynamically allocate the resources required, monitor the QoS received, alter resource allocations when necessary, and perform run-time adaptation of applications, middleware, and operating or communication systems [1, 10, 16]. Specifically, *local resource management* on a host ensures that resources are distributed across applications to help them achieve their desired quality of service. However, since achieving and maintaining QoS for distributed applications is an end-to-end issue [11], multiple local resource managers must cooperate – *global resource management* – so that QoS guarantees can be applied to the entire flow of data. Such end-to-end QoS management addresses the delivery and processing of data from the server to the client and the management of associated resources, including CPU, memory, disk, and network bandwidth, along the path of the application data flow (e.g., from a server to its clients). In addition and to meet the specific needs of individual applications, QoS-awareness of applications [9] has been shown important.

The topic of our research and the focus of this paper is the efficient integration of the multiple management techniques applied to distributed multimedia applications into one control path. Specifically, we aim to make the global management of such applications' distributed resources and the adaptation of their distributed components more effective. Our approach is cooperative, where (a) local resource management, which ensures the allocation of local resources according to an application's requirements, cooperates with (b) global resource management, which ensures that resources are allocated along the entire path of a distributed application's data communications, and both cooperate with (c) application adaptation, which enables an application to better deal with resource limitations and with unforeseen variations in resource availabilities.

Problem statement. *Cooperative run-time management* aims to combine the local and global management of distributed resources with the run-time adaptation of an application. Earlier research principally considered adaptations performed at application-level [2, 3]. More recent contributions include multi-resource solutions [15, 21], using libraries [17] and/or additional servers [6] for distributed adaptation and resource management [19]. Cooperation between application-level adaptations and system-level resource management is implemented as middleware [4] or as point solutions for specific resources like computer networks [5]). Other work, such as OMEGA [10] or QuO [16] introduce general QoS architectures to address the end-to-end management of QoS.

Our research provides efficient system-level and architectural abstractions that enable the cooperative management of kernel-level resources with the run-time adaptation of user-level applications. Abstractions are implemented as lightweight, kernel-level services, jointly termed *Q(uality)-fabric*, which are based on the notions of distributed event services and event handlers. The issues addressed by Q-fabric services concern the limitations experienced in current systems for cooperative resource management and application adaptation, which are due to (a) the specific interfaces defined between applications and resource managers and (b) the limited interactions permitted between distributed resource managers. Moreover, (c) cooperation is particularly difficult when resource management actions are performed at kernel-level. Finally, (d) in large-scale multimedia applications, the coordination of a large number of distributed resource managers for heterogeneous resources and the adaptation of all instances of an application can be overwhelming in complexity [17].

Solution approach. The **Q-fabric** architecture addresses issues in cooperation in multiple ways. First, it gives each application the ability to control the resource monitoring, management, and adaptations performed on its behalf. Second, these actions may be tuned continuously, by monitoring management effectiveness and altering management behavior through run-time modification of selected parameters. Third, all such actions are performed within a uniform framework that binds application and resource managers into a single *integrated* and distributed QoS management system.

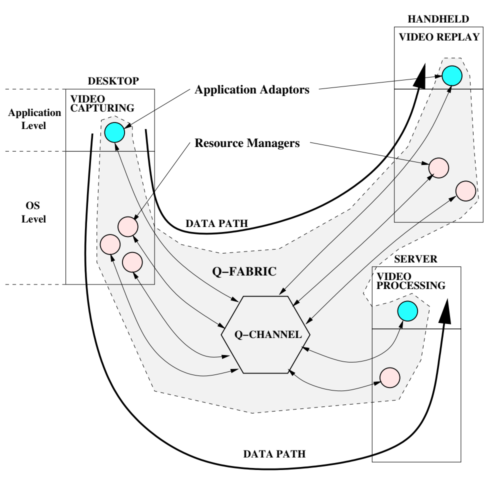

Figure 1: The Q-fabric architecture.

Q-fabric (Figure 1) is a Linux kernel-level service with which multiple distributed resource managers and an application's adaptations can be dynamically integrated into a single QoS control path. Integration is dynamic in that applications and resource managers can choose to join or leave such cooperations at any time. This is achieved by using *Q-channels* [12] as the fabric's control infrastructure, which is a kernel-level event service that follows the publish/subscribe paradigm. With Q-channels, applications and resources are managed via the exchange of monitoring and steering events. Specifically, resource managers exchange events within a host and between hosts to monitor resource availability and to coordinate the allocation of resources to applications. Applications use the same event path for their control communications (i.e., for adaptation), as exemplified by a video server announcing the use of a different encoding scheme to its clients. Finally, applications and resource managers communicate with each other via events to exchange monitoring information or to request adaptations in resource allocations or in application behavior. In all such actions, the only interface between all communicating entities is that of an <event:action> pair, i.e., resource managers and applications submit well-defined events and react to the arrival of an event with a certain well-defined action. Note that a Q-channel as depicted in Figure 1 is not a centralized unit, but is distributed among all participating hosts, i.e., all hosts communicate directly with each other (e.g., via socket connections) and channel information (e.g., lists of participating publishers and subscribers) is replicated at each host.

Contributions. The main contributions of this paper are as follows. Q-fabric is a novel, kernel-level QoS management infrastructure that permits the integration of kernel-based global resource management with the user-level adaptation of applications. Adopting the event-based model of component interaction, Q-fabric provides the basic services needed for implementation of cooperative QoS management strategies for distributed multimedia applications. Benefits of using the Q-fabric are demonstrated for an on on-line video-conferencing facility, running on a platform of networked Linux PCs. Gains attained in this fashion can be substantial, as exemplified by a reduction in the jitter and therefore, end-to-end delay experienced by data sent from an active video source to a sink. In particular, we show in our experiments that cooperative management of resources and applications in a simple client-server setting is (a) able to maintain the desired video frame rates even with heavy network or CPU perturbation and (b) able to bound jitter to less than $0.05s$ (compared to $5s$ and more for situations where only the kernel resources or the applications are adapted). This reduction is due to the cooperative management of both the network and CPU resources on source and sink machines, coupled with the dynamic adaptation of application behavior.

2. QOS MANAGEMENT MODEL

A main contribution of the Q-fabric is that it allows applications to directly interact with other applications as well as with distributed resource managers, in order to cooperatively manage the quality of service experienced by end users. Further, the Q-fabric mechanism is implemented as a kernel-level service, which enables QoS management at higher levels of predictability and performance than possible with the user-level counterparts developed in our past work and elsewhere [7, 12, 14]. The remainder of this section describes the QoS management model used in our work.

Event-based communications. The Q-fabric's operation is based on the exchange of events, which trigger the execution of some action (e.g., monitoring, adaptation, etc.). Event exchange is anonymous, so that local and remote resource managers and applications can interact freely, without having to be aware of each others locations or of the number of applications or resource managers involved. In this fashion, Q-fabric addresses the highly heterogeneous and dynamic computing platforms on which future multimedia applications will be run. In any case, the *events* interchanged via the Q-fabric coupling distributed entities are variable-length data structures, defined by the QoS management system and exported to applications in C header files. New events can be defined at run-time – by applications and resource managers – using an event registry (not yet implemented). This minimizes the modifications necessary to applications and resource managers when new functionality is implemented. Events are represented by <**event:action**> pairs: a source submits an event, which is received by one or more event sinks. An event sink executes the corresponding action, that is, function, in response to the event.

Cooperative system- and application-level run-time management. Consider the scenario of one or more resource managers embedded in the Linux kernel (for resources such as CPUs, network connections, disks) and a number of applications sharing these resources. Cooperation implies not only that applications need a way to specify their resource needs, but also, that they can customize the resource management to their specific requirements, monitor their resource allocations, and control both the adaptation of such resource management and of their own implementations. The Q-fabric is implemented to provide appropriate 'connections' across all cooperating entities, including the resource managers and the applications on all participating hosts. Specifically, applications specify their QoS requirements via events, which are received by the underlying resource managers at all participating hosts. If admission control succeeds, resources are allocated to the application. During operation, resource managers again use events to monitor the achieved quality of service and adjust their resource allocations. At the same time, applications monitor the achieved quality of service and adapt themselves or request different resource allocations from the resource managers.

Cooperation via Q-channels. Event communication is performed via Q-channels, which connect resource managers and applications with the purpose of exchanging control information. A Q-channel is created whenever an application initiates an interaction (i.e., initiates communication) with another application, such as a client requesting a service from a server. Applications share control information to ensure optimal performance or to adapt to heterogeneous system environments. The video conferencing tool *vic*, for example, uses a control channel (based on RTCP) to achieve rate control. On the other hand, resource manager cooperation is required to achieve end to end QoS management. Our approach distinguishes itself from previous approaches by combining these two control paths into one: a kernel-based event channel connects kernel-based resource managers and user-level applications, allowing them to coordinate their resource management and adaptations.

Resource managers. A resource manager's task is to distribute one or more system resources among a number of competing applications according to some *resource management policy*. Figure 2 shows the structure of a resource manager. First, a **monitor** entity keeps track of changes in resource allocations and stores this information in a *statistics* data structure, and submits a *monitoring event* to the Q-channel. Second, an **adaptor** entity receives such monitoring events and uses these events together with the stored QoS requirements of applications and the statistics data structure to decide if (a) local resource allocations should be

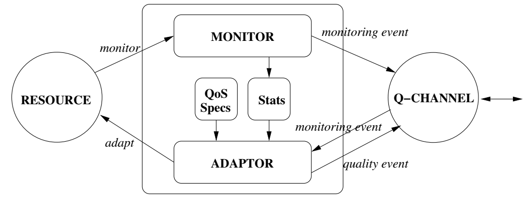

Figure 2: A resource manager.

adjusted and (b) remote resource allocations or applications require adaptation, that is, a *quality event* has to be issued. It has to be noted that this paper focuses on the implementation of a QoS management **mechanism** as opposed to a QoS **policy**, where a generic and flexible mechanism is able to support any desired QoS policy. A QoS policy is determined by the actual implementation of the monitor and adaptor entities described above and will be focus of our future work. For the purpose of evaluating our QoS management mechanism, we consider simple QoS policies in Section 4.

3. IMPLEMENTING QOS MANAGEMENT

Although the Q-fabric supports QoS management at both user-level and kernel-level, this paper focuses on kernel-level solutions because previous work, including our own work on the management of distributed radar sensors [6] has shown that there are problems with user-level approaches. For example, 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. This is caused by inappropriate delays of QoS management [18] (e.g., overheads of application-level QoS managers' interactions with the system-level mechanism they must use to monitor and steer resource allocations). Further, operating systems often provide inappropriate interfaces that require managers to poll for changes in resource state or make unnecessary resource reservations (as also noted in [8] and [13]).

3.1 Multimedia Applications

The multimedia applications addressed by Q-fabric are those that are subject to dynamic variations in underlying resources and in current user needs. Examples include networked mobile sensors cooperating to deliver remotely captured data to certain sinks in real-time, adhoc-networked PDAs operating in dynamically changing contexts, and embedded computers in mobile platforms like cars and airplanes that interact to exchange time-critical information. The application studied in this paper is a videoconferencing tool (*vic*) operating on both handheld and desktop machines, connected via wireless and wired communication links. The QoS issue arising for this application is the control of the end to end delay of image transfer and the control of the jitter of image arrival and display at the client. Figure 3 shows the experimental setup used in this paper, with two machines running vic, where the first one is used as a video source (server-vic) and the second as a video sink (client-vic). Both hosts use a CPU resource manager (managing a round-robin scheduler), and the vic server also uses a network resource manager (managing a DWCS [20] packet scheduler).

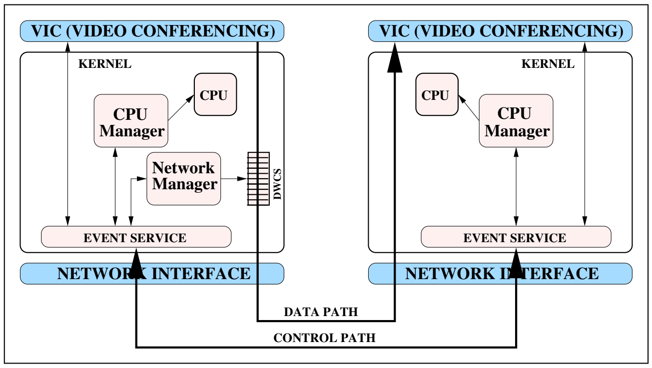

Figure 3: Experimental setup.

3.2 Communication Setup

A user-level application initiates a data transfer, typically by creating a socket or some other message transfer mechanism, which then triggers implicitly the creation of a Q-channel in the kernel. In this paper we use modified Berkeley socket calls to set up and operate a connection between distributed applications. A privileged application uses the *qos_conf()* system call to set up kernel resource managers. The *qsocket()* system call allows any application to establish a new socket connection, while the kernel – transparent to the application – creates or opens a new Q-channel and forces the kernel-based resource managers to subscribe to this new channel. Also, the application is being subscribed to this channel implicitly. Termination of the socket connection with *qclose()* causes the application to unsubscribe from the channel. The underlying resource managers also unsubscribe if there is no other application using the same channel on the same host. Once the Q-channel is set up, applications and resource managers can interact freely via the exchange of events.

3.3 Customization

Resource management and application adaptations are based on <event:action> pairs. Events are used to describe certain system states or significant occurrences. We distinguish between monitoring and quality events, where monitoring events are issued by monitor functions and carry system state such as current CPU load, network bandwidth, and average buffer lengths. This event is received and handled by the adaptor function that analyzes the received information, compares received QoS with desired QoS and possibly issues quality events. Quality events are intended for system adaptation: an adaptor function is invoked upon arrival of a quality event, which then decides *how* and *where* to adapt an application or to adjust resource allocations.

An important objective of developers of resource managers and applications will be to define <event:action> pairs that optimize the performance of an application. Currently we assume that both QoS developers and application developers are aware of the available <event:action> pairs, our future work will extend this to allow both resource managers and applications to define new events on the fly.

The definition of event types and the corresponding actions associated with these events is a powerful tool to customize the resource management to the specific needs of an application. Q-channels allow subscribers (resource managers and applications) to insert **filters** into an event stream to either customize the event stream to the specific needs of the subscriber (e.g., to ignore and drop certain unnecessary events) or to pre-process certain events (e.g., to aggregate or modify events, etc.) in order to influence the adaptation process.

Further, Q-channels allow applications to insert their own QoS monitors and adaptors, thereby implementing their own QoS management policies. For example, a video transfer application might require a different adaptation behavior to buffer overruns than an audio application. This approach allows applications to implement their own QoS management strategies into the kernel and therefore close to the resources being managed, resulting in low-overhead and fine-grain adaptations. The detailed discussion of the implementation of application-specific policies is out of the scope of this paper.

3.4 Global Resource Adaptation

Global resource management addresses the management and adaptation activities performed at the kernel-level, between distributed resource managers. These resource managers monitor resource allocations, achieved quality of service, or system data structures and issue monitoring events either (i) periodically or (ii) when certain thresholds have been exceeded. These events are issued without concern about the number or identity of the receivers. Other resource managers, on the same and on remote hosts, receive the monitoring event and analyze it to decide if and how to react. This could include the generation of a quality event to other resource managers if it is necessary to coordinate adaptation between several hosts and not only on just one. The goal of global resource management is to ensure that applications receive their desired QoS and that the system adapts to changes in the environment properly (e.g., arrival or departure of tasks, data streams, or resources). A typical resource management scenario is a feedback-based loop, where client-based resource handlers monitor buffer levels and return such information in monitoring events to the server-based resource manager, which, in return, adapts the CPU and network allocations and issues a quality event to the client-based resource managers to adjust their resource allocations as well.

3.5 Application-level Adaptation

Adaptation at the kernel-level is limited, because (a) only the application knows best how to adjust its operations optimally to changing situations, and (b) only the application can specify methods for graceful degradation such as reduction in color depths for video applications. Distributed applications can monitor their achieved quality and issue monitoring events if necessary. These monitoring events are analyzed and applications can modify their behavior accordingly and issue quality events to other applications if required.

3.6 Integrated Adaptation

Global resource management schemes typically implement some notion of fairness across multiple applications. At the same time, application-level management attempts to optimize each application's behavior given current resources. A combined approach can achieve both, indicating the need for integrated resource management and application adaptation. Our approach achieves integration by sharing the same control mechanism, a Q-channel. Events can be exchanged directly between all involved applications and resource managers, and resource managers and applications can jointly adjust to attain desired QoS and performance levels. In other words, applications and resource managers interact directly, without crossing unnecessary or costly interfaces (faster adaptations). Further, the integration lets applications directly monitor and control remote resource managers without having to rely on the QoS management on the same host to supply (possibly stale) QoS-related information. Finally, our future work will address the dynamic customization of QoS management by inserting run-time generated monitor and handler functions or by customizing the event traffic through event channel filters, all of which is facilitated by a close integration of application and resource management control paths.

3.7 Q-fabric Details and API

Q-fabric is based on Q-channels, a kernel-level implementation of an event service, similar in functionality to the CORBA Event Service. Q-fabric calls are identical for both user-level applications and kernel-level resource managers, which has been achieved by exporting the same kernel function names as system calls. These calls include:

- *EChannel_create* and *EChannel_open* are used to create and open a Q-channel.
- *ECsubmit_event* is used to submit monitoring or quality events to the Q-channel.
- *CMpoll_network* is used to poll for pending events.

Besides periodic polling for pending events, applications can also subscribe to the ECalls event notification mechanism, which is able to (a) asynchronously invoke event handler functions on behalf of applications and (b) modify the CPU scheduling priority of the application to minimize its event responsiveness (see [13] for details).

Finally, while Q-channels can be created, deleted, and used explicitly, to facilitate their use, applications can also use modified socket calls (called *QSockets*), where Q-channels are implicitly associated and used with these sockets:

- *qsocket*: additional attributes indicate the desired QoS for the application (QoS specification). Further, a qsocket call forces the kernel resource managers to create/open a Q-channel and to start resource management for this application.
- *qsend*: additional attributes indicate per-message QoS information (e.g., priority) or modify the previously indicated QoS specification.
- *qrecv*: in addition to message data, QoS-related information can be received (e.g., QoS statistics).

4. EXPERIMENTAL EVALUATION

All experiments are performed on a dual-Pentium II with 400MHz, 512MB RAM, 512KB cache and a dual-Pentium II with 300MHz, 256MB RAM, 512KB cache, both connected via a switched 100Mbps Fast Ethernet and running Redhat Linux 7.1 (kernel version 2.4.17).

Experiments analyze the behavior of Q-fabric in a specific application setting. The application used is that of a video conferencing tool, called *vic*, which is part of the Open-MASH toolkit (www.openmash.org). We apply a number of changes to vic such that it takes advantage of a Q-fabric-based resource management system (e.g., the replacement of regular socket calls with QSocket calls).

Resource 1: CPU. The CPU scheduler used in this example is the Linux real-time round-robin scheduler. The resource manager adjusts an application's priority class to react to its varying computational needs. The CPU resource manager has no monitoring component, only an adaptation component, that is, only quality events and no monitoring events are being issued. Applications are assigned a default priority class (e.g., 50 in the following experiments) and can be modified in the range of 1 and 99.

Resource 2: Network. The Dynamic Window-Constraint Scheduler or DWCS [20] is a real-time scheduler based on three attributes: a period T, a window-constraint or loss rate x/y, and a run time c, where DWCS guarantees an activity c time units of service within a period T. However, this guarantee is relaxed by the loss rate, which indicates that x service invocations in y consecutive periods (i.e., $y * T$ time units) can be missed. These attributes translate easily to streaming multimedia applications that require the generation and transmission of data (such as video or audio) with a certain rate. However, such applications can often tolerate infrequent losses or misses of data generation or transmission. If a packet is not scheduled within a period T, it is said to have *missed* its deadline. If the number of missed deadlines exceeds x in a window of y, the stream is said to have suffered a *violation*. The adjustable parameters of a DWCS stream are the period and the loss-rate. The following experiments use a default period of $50ms$ (to achieve a frame rate of 20fps) and a loss rate of $x/y = 1/10$.

Application adaptation. Vic can react to the receipt of quality events by adjusting a variety of parameters:

- **Encoding Method.** For our experiments we use H.261 and JPEG images. vic can dynamically switch between different methods, where the choice of encoding can significantly affect the resulting CPU and network overheads.
- **Image Quality.** In vic, the used image qualities are in the range of 1 (low) to 95 (high) for JPEG images and 1 (high) to 30 (low) for H.261.
- **Image Size.** Both JPEG and H.261 images can have the two following sizes: *small* (176*144) and *medium* (352*288). In addition, JPEG images can also have the image size *large* (640*480).

The primary goal of the following experiments is to maintain a frame rate of 20fps or the best possible frame rate if resource contention is too high. The secondary goal is to minimize jitter at the client.

4.1 Adaptation-level Adaptation

Policy. The client-vic feeds achieved frame rates back to the server-vic (once per second), which then adjusts the image quality to maintain the desired frame rate. The images are H.261 encoded and have a default quality of 15. The quality is modified in steps of 1.

Perturbation. A CPU-intensive task (endless for-loop) is run first at the server, then at the client.

Details. Application-level adaptation is a useful tool to adjust the application behavior to changing environments. We argue that application adaptation can significantly improve the overall quality of a media stream, however, the possibilities are limited without the support of a global resource management mechanism. Figure 4 shows the frame generation rate at the server side of a vic conference. The desired frame rate is 20fps, however, the actual generation rate can vary significantly at times, for example, to 11fps at 113s. After 146s, a CPU-intensive task starts at the server,

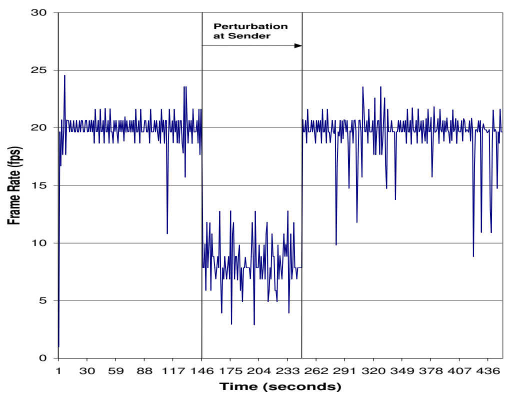

Figure 4: Frame generation rate (server) without resource management and adaptations.

causing the frame generation rate of the vic server to drop to a range of 4fps to 13fps. After 250s, this perturbation is stopped. Figure 5 repeats the same experiment, however this time, the client-vic uses the Q-fabric to inform the sender-vic about the achieved frame rate. After 146s, the server-side CPU perturbation is started and this time, the server manages to maintain the frame rate of 20fps by reducing the image quality. Note, however, that application adaptation is still not able to handle the strong deviations from the desired frame rates, which are caused by resource contention on both the server and the sink. Figure 6 shows

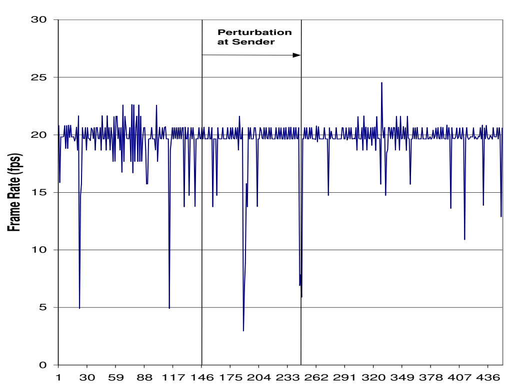

Figure 5: Frame generation rate (server) with application adaptation (image quality).

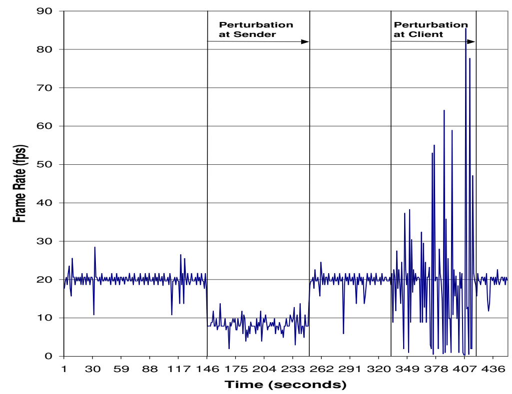

Figure 6: Frame replay rate (client) without resource management and adaptation.

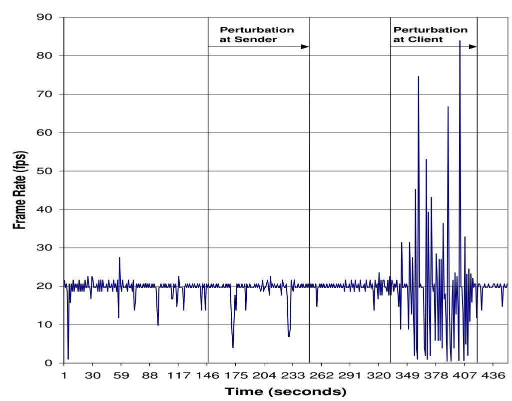

Figure 7: Frame replay rate (client) with server-side application adaptation.

the behavior of the client-vic when (a) the server and (b) the client experiences CPU contention. After 146s, notice the drop in frame replay rate, which is due to the server-side CPU contention described in the previous measurements. After 330s, a newly started client-side CPU-intensive perturbation task creates strong variations of frame replay rates between 0fps and 86fps. Figure 7 repeats this experiment with application-level adaptation. Though the adaptation manages to successfully handle the server-side contention, it fails to manage client-side contention properly even though image quality is reduced, resulting in smaller image sizes and less computational needs at the client-vic. This is also

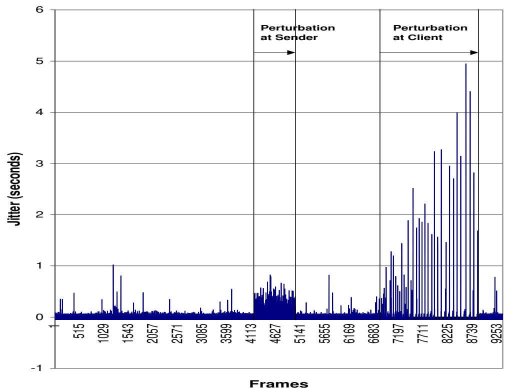

Figure 8: Jitter in frame replay (client) without adaptation.

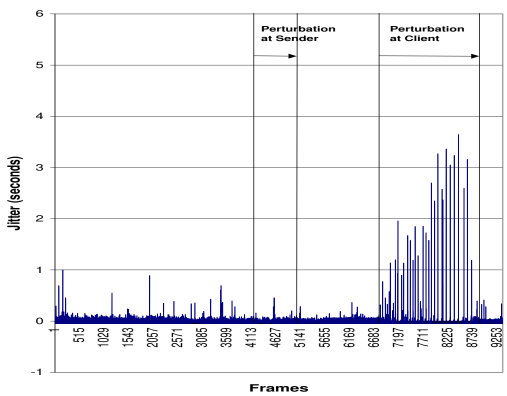

Figure 9: Jitter in frame replay (client) with adaptation.

underlined by the next two graphs, Figure 8 and 9, which show the jitter at the client. Without adaptation, the jitter for the server-side contention increases from an average of $0.05s$ to $0.5s$ for the server-side contention and to more than $1s$ for the client-side contention. Peak values for the jitter for the client-side contention are close to $5s$. With adaptation we are able to eliminate the increase in jitter for the server-side contention, however not for the client-side contention (Figure 9).

Results. Summarizing, we observe that in our example, application adaptation (e.g., changing the image quality) can significantly limit the negative effects of server-side CPU contention. However, application-level adaptation fails to efficiently support the application at client-side contention. Further we notice that frame rates can vary significantly even without relevant CPU or network contention, due to a variety of reasons such as network latencies, contention with OS daemons, sending of frames in bursts, etc. In the following measurements we further underline our argument that only management of all involved resources in conjunction with application adaptation leads to an acceptable application quality of service.

4.2 Distributed Resource Management

Policy. The network resource manager at the client-vic issues monitoring events containing the rate of received images at the client once per second. The CPU resource manager at the server modifies the CPU allocation to the server-vic if this rate differs from the desired frame rate. The default real-time priority is 50, which is being modified in steps of 1 (between 1 and 99).

Perturbation. A CPU-intensive task (endless for-loop) is run at the server. The CPU-intensive task has a real-time priority of 50.

Details. the problem with application-level adaptations is that it succeeds in reacting to changing environments in some cases only, and that competing applications have no way of preventing each other from stealing resources. Distributed or global resource reservation and management is required to distribute resources between applications. Figure 10 and Figure 11 display the client-side frame replay rate and the jitter for the same experiment, this time, however, with global resource management. The server CPU resource manager uses feedback from the client-based CPU resource manager to adjust the CPU allocations to the server-vic. In addition, the client CPU resource manager adjusts the CPU allocation for the client-vic if CPU contention exists.

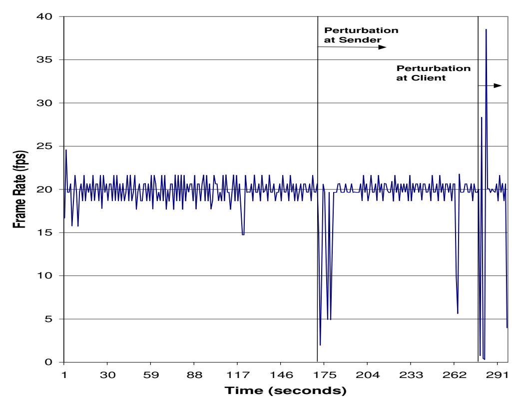

Figure 10: Frame replay rate at client with global resource management.

Results. Global resource management succeeds in adjusting resource allocations such that the desired frame rate of 20fps can be maintained. The jitter can also be maintained at its average value of $0.05s$. Note, however, the strong deviations in frame replay rate (which can be up to 100%) and jitter (which can reach several seconds) when contention starts. Although the global resource management succeeds in maintaining a desirable replay rate and jitter eventually, it takes about $10s$ to stabilize both replay rate and jitter in our measurements.

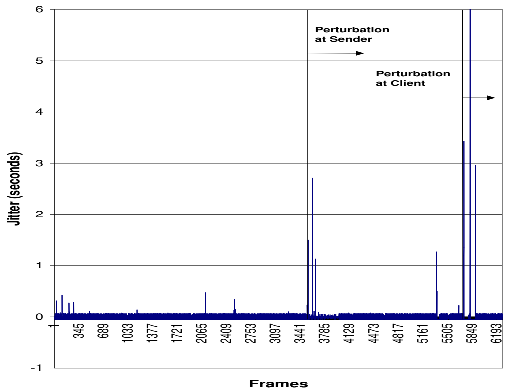

Figure 11: Jitter in frame replay at client with global resource management.

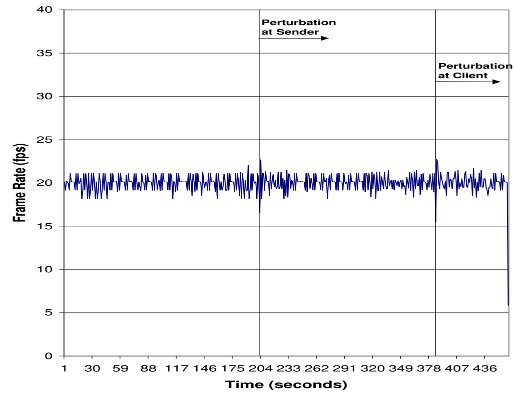

Figure 12: Frame replay rate (client) with global resource management and application adaptation.

4.3 Integrated Approach

The following experiments display the successful interaction of multiple resources (CPU, network) and applications to achieve optimal QoS management.

1st Approach: client-feedback.
Policy. the client-vic issues monitoring events (once per second) to the resource managers and to the server-vic, containing the achieved frame rate. The server-vic adjusts the image quality of the H.261 encoded images (between 1 and 30) if the frame rate differs from the desired frame rate. Both the CPU resource managers at the server and the client adjust the CPU resource allocations as described in the previous experiment.
Perturbation. A CPU-intensive task (endless for-loop) is executed at the client.
Details. In the integrated approach of QoS management, applications and distributed resource managers cooperatively adjust their allocations and behaviors to react to changing system environments or load situations. Figure 12 and 13 show the frame replay rate and the jitter when we perform global resource management and application adaptation in conjunction.
Results. Figure 12 shows that the integrated approach succeeds in maintaining the desired frame rate of 20fps at all times. The jitter graph, however, indicates where the contentions start with a large deviation from the desired value for both server-side and client-side contention. In a feedback-based approach as used in our experiments, these short deviations are unavoidable because they are required to trigger adaptations.

2nd Approach: network-feedback.
Policy. The network resource manager uses information from the DWCS scheduler about the number of missed deadlines and issues this information as monitoring events once per second.
Perturbation. Network perturbation is simulated by limiting the network traffic between sender and client to 14kBps.
Details and Results. Figure 14 shows the frame replay

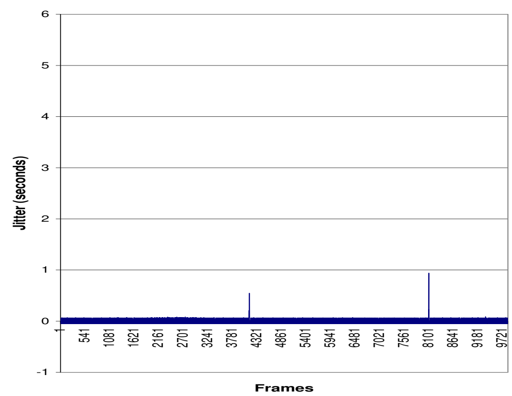

Figure 13: Jitter (client) with global resource management and application adaptation.

rate at the client. While the desired frame rate is 20fps, the achieved frame rate fluctuates around 15fps. In Figure 15 we display the same scenario, this time with integrated adaptation and resource management.

3rd Approach: client- and server-feedback.
Policy. Feedback about the generated frame rate (server) and the achieved frame rate (client) is used to adjust the image quality and size.
Perturbation. No perturbation is used, instead we use JPEG compression, which causes CPU overheads too high to maintain a frame rate of 20fps.
Details. In this final set of experiments, we use JPEG images instead of H.261 due to their increased CPU requirements. In fact, when generating a video stream using JPEG compression, vic is not able to generate the frames at the desired rate of 20fps (see Figure 16), even with the support of kernel resource management. The CPU resource man-

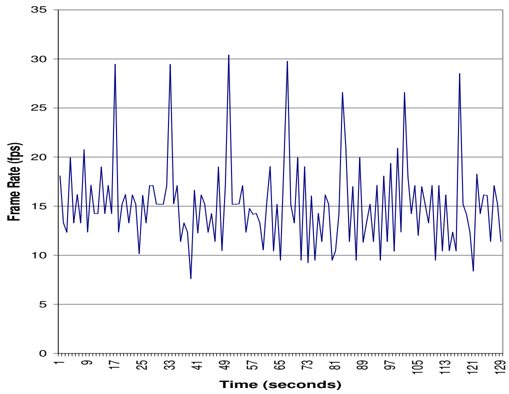

Figure 14: Frame replay rate (client) with kernel resource management.

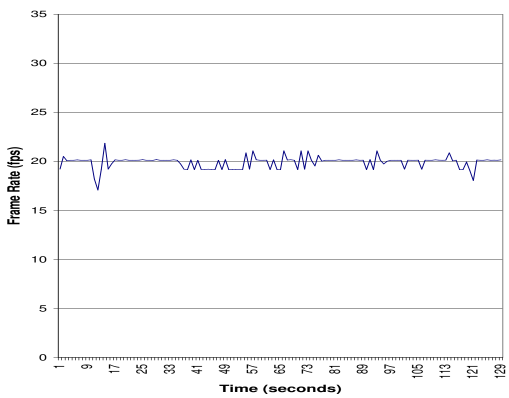

Figure 15: Frame rate (client) with kernel resource management and application adaptation.

ager tries to allocate more and more CPU bandwidth to the application, until it receives almost 100% of the CPU. In the second measurement shown in Figure 17, we use feedback from the client-vic (frame replay rate) and from the server-vic (frame generation rate) to adjust the CPU resource managers (particularly CPU) and to adapt the application (reduce quality and image size). This time we succeed in achieving the desired frame generation rate of 20fps. **Results.** This shows that when resource management hits its limits, application adaptation can ensure that the quality of the stream degrades gracefully while achieving the desired frame rate. Summarizing, we conclude that the global management of resources and adaptations of applications can benefit applications in their goal to achieve a desired quality of service, particularly when the application has to compete with others. In this paper, we introduced an event-based cooperation scheme between multiple distributed resources and distributed applications, allowing to cooperate

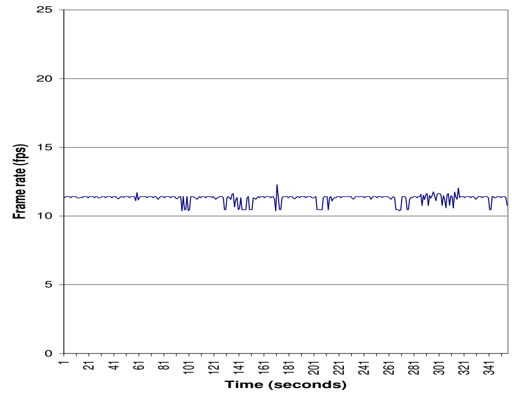

Figure 16: Frame generation rate (server) with kernel resource management but without application adaptation.

their management decisions by exchanging **events** directly between each other. Receipt of an event triggers the execution of application adaptation and resource re-allocations in an attempt to maintain or improve the received QoS.

5. CONCLUSIONS

This paper introduced the 'Q-fabric' approach to QoS management, which is used to integrate application adaptation and resource management closely via a kernel-based event service. Q-channels are event channels that are shared between resource managers and applications to exchange resource management information and requests, both asynchronously and anonymously. The advantages of having applications and resource managers share the same control path are several. First, distributed applications can interact freely to monitor and adapt their behavior according to the desired and achieved QoS. Second, applications are able to directly address resource managers, locally and globally. Previous approaches either prohibited the access to remote resource managers or only allowed this by crossing several interfaces, typically involving actions by underlying resource managers. Third, resource managers can interact with each other to support applications in achieving their desired QoS and to ensure fair distribution of resources. Fourth, resource managers can address applications directly by issuing monitoring and quality events to them, therefore requesting application adaptations. Finally, QoS management policies can be developed that combine application adaptation with distributed resource management.

6. FUTURE WORK

Though customization functions can be inserted dynamically, they are assumed to be supplied to the Q-channel by the application or a kernel-loadable module. Future implementations of the Q-fabric will be able to dynamically generate such functions. Using such functions, applications will be able to monitor resource management actions, associate these actions with certain application behavior, and detect

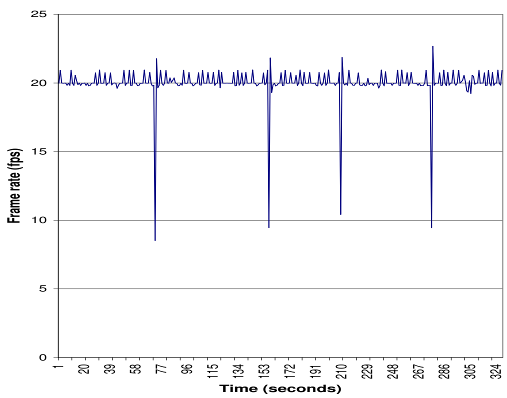

Figure 17: Frame generation rate (server) with both kernel resource management and application adaptation.

'hot spots' in application code (i.e., code executions that trigger resource adaptations). Possible actions include the removal of hot spots or warning resource managers about them to enable preventative actions. Alternatively or in addition, resource managers can monitor application management and react immediately to such actions by also adjusting resource allocations. Further, they can monitor application adaptations and log the effects of these adaptations on resource usage. This can be used to predict the need for changes in resource allocations when such application adaptations occur again in the future.

7. REFERENCES

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

[2] T. E. Bihari and K. Schwan. Dynamic Adaptation of Real-Time Software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[3] C. Diot. Adaptive Applications and Qos Guarantees (Invited Paper). *Multimedia and Networking*, 1995.

[4] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proc. of the 8th Intl. Workshop on Quality of Service, Pittsburgh, PA*, 2000.

[5] Q. He and K. Schwan. IQ-RUDP: Coordinating Application Adaptation with Network Transport. In *Proc. of High Performance Distributed Computing*, 2002.

[6] J. Huand, R. Jha, W. Heimerdinger, M. Muhammad, S. Lauzac, B. Kannikeswaran, K. Schwan, W. Zhao, and R. Bettati. RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications. In *Workshop on Middleware for Distributed Real-Time Systems*, 1997.

[7] 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.

[8] 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.

[9] L. Li, F. Lizheng, L. Guixing, and W. Ping. A Hierarchical Architecture for Distributed Network Management. In *International Conference on Internet Computing*, pages 420–423, 2001.

[10] K. Nahrstedt, H. Chu, and S. Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal on High-Speed Networking, IOS Press*, 7(3,4):227–255, 1998.

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

[12] C. Poellabauer and K. Schwan. Kernel Support for the Event-based Cooperation of Distributed Resource Managers. In *Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, September 2002.

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

[14] 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.

[15] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. Practical Solutions for QoS-based Resource Allocation Problems. In *Proceedings of the IEEE Real-Time System Symposium*, December 1998.

[16] C. Rodrigues, J. P. Loyall, and R. E. Schantz. Quality Objects (QuO): Adaptive Management and Control Middleware for End-to-End QoS. In *OMG's First Workshop on Real-Time and Embedded Distributed Object Computing*, 2000.

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

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

[19] D. Waddington and D. Hutchison. A General Model for QoS Adaptation. In *Proc. of the 6th Intl. Workshop on Quality of Service (IWQoS '98), Napa, California, USA*, May.

[20] R. West, K. Schwan, and C. Poellabauer. Scalable Scheduling Support for Loss and Delay Constrained Media Streams. In *Proc. of the 5th Real-Time Technology and Applications Symposium*, Vancouver, Canada, 1999.

[21] D. Xu, D. Wichadakul, and K. Nahrstedt. Multimedia Service Configuration and Reservation in Heterogeneous Environments. In *Intl. Conference on Distributed Computing Systems*, 2000.