Implicit Quality Channels (IQC): Distributed Quality Management for Multi-Party Real-Time Applications

Christian Poellabauer and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

Multi-party, interactive multimedia (MIM) applications pose challenges for resource management due to their simultaneous use of multiple media and their heterogeneous, distributed, and potentially large numbers of participants. Two main difficulties are the support of (1) quality of service (QoS) and (2) scalable group or multi-peer communication, where QoS management is complicated by relationships between different media streams and by the sizes and dynamics of groups. That is, QoS management must capture not only the identities of group participants, but also their run-time behavior. This paper describes a novel operating system construct, termed Implicit Quality Channels (IQC)*, which extends standard Berkeley Sockets to support resource management of multi-peer applications like teleconferencing and CSCW. Specifically, when a participant joins an MIM, its inclusion in the MIM's quality management infrastructure is triggered transparently. Such inclusion is implemented in three steps. First, when creating a listening socket, the MIM associates appropriate quality management information with that socket. Second, when another participant connects to this socket, the operating system kernels involved* implicitly *create a kernel-level event channel that will carry this QoS as well as adaptation information. Third, the resource managers on the participating hosts transparently subscribe to this event channel, thereby dynamically creating appropriate groupings of collaborating resource managers and ensuring that all participants are managed properly. Management includes the monitoring of participant behavior, the submission and receipt of adaptation events, and the dealing with requests for re-negotiation of the initial QoS. In effect, any MIM participant establishing a connection to an MIM-provided socket implicitly accepts the MIM's QoS management.*

1 Introduction

Real-time and multimedia applications that require QoS guarantees rely on the support of the underlying operating system (OS) to manage the allocation of resources among competing applications. For instance, video display applications need sufficient network bandwidth to receive images with low latency, and they need sufficient memory and CPU cycles to process and display these images. *Resource managers* have the task to allocate the resources required by applications, to monitor the quality of service they receive, to alter resource allocations as necessary, and to affect run-time adaptations of applications, middleware, and operating or communication systems [1, 2, 3, 4, 5].

To achieve and maintain QoS for distributed applications is an end-to-end issue. For instance, for multimedia

applications, QoS guarantees have to apply to the entire flow of media from the server to the client. Previous research has used middleware to 'bind' the multiple machines, applications and resource managers that implement QoS provisioning, resource management, and performance differentiation [1, 4, 3] for distributed applications and platforms, sometimes enhanced by OS extensions on individual machines [6, 7, 8]. We argue that middleware-based approaches, including our own previous work [3, 9], cannot enforce that all programs interacting with QoS-managed applications participate in such management. We also maintain that the performance and scalability requirements of distributed real-time systems make it necessary to place at least some of the necessary QoS management mechanisms inside operating system kernels, i.e., in immediate proximity to the resources being managed and the system structures and services being accessed and utilized for resource management.

This paper describes a novel operating system mechanism to 'bind' applications to resource managers and machines, termed *Implicit Quality Channels (IQC)*. IQCs affect such bindings for arbitrary distributed applications by extending the functionality of standard Berkeley Sockets. Specifically, whenever a process connects to a socket extended with an implicit quality channel, its inclusion in the quality management infrastructure associated with that socket is triggered transparently. Such inclusion is implemented in three steps. First, when creating a quality-managed socket, the application associates appropriate quality management information with that socket. Second, when another process connects to this socket, the two operating system kernels involved implicitly create a new kernel-level event channel that will carry the QoS information associated with that socket, as well as monitoring and adaptation information. Third, the resource managers on the participating hosts transparently subscribe to these implicit quality channels for the exchange of quality events [10]. Thus, IQCs dynamically create appropriate groupings of collaborating resource managers and ensure that communicating distributed processes are managed properly. In effect, any process establishing a connection to an extended socket implicitly accepts the QoS management associated with that socket. Collaborative quality management actions implemented in this fashion may involve monitoring the behavior of communicating processes and machines, receiving and submitting adaptation events from/to processes, and dealing with their possible requests for re-negotiation of the initial QoS.

1.1 Multi-Party Interactive Multimedia

Multi-Party Interactive Multimedia (MIM) [11] or *multipeer* applications have become an important class of real-time multimedia applications [12]. Evidence of their importance is the recent emphasis on group communication, such as peer-to-multipeer and multipeer-to-multipeer [13, 14]. Sample MIM applications include teleconferencing, CSCW, and remote teaching, all of which support collaboration among a potentially large number of participants, by integrating the communication of discrete media (such as text or data) with continuous media (such as video and audio). As an example, *Grid* [15] computing has emerged as a new field of large-scale, high-performance distributed computing for science and engineering. The *Access Grid*[1] is a framework for group-to-group interaction to support distributed seminars, meetings and lectures on the Grid. It includes access to video and audio streams, text, presentation slides, and visualization environments.

Typical characteristics of MIM applications are (1) the potentially large number of participants, leading to

[1] http://www-fp.mcs.anl.gov/fl/accessgrid

scalability and performance issues, (2) the dynamic nature of these applications, i.e., participants joining and leaving sessions at run-time, and (3) their simultaneous use of multiple types of media, leading to problems regarding QoS provisioning, synchronization, and reliability. To address these problems, it is necessary to provide mechanisms for the efficient and scalable management of resources between participants of multi-party multimedia applications. The following criteria are essential for the efficient QoS support of such MIM applications [16, 17]:

- **End-to-end QoS.** The provision of QoS is an end-to-end issue [18], that is, QoS assurances are required for the entire path of a media stream, where several streams may compete for the same underlying resources. In addition, QoS management must take into account inter-stream relationships caused by resource sharing or explicit synchronization requirements. With IQCs, any number of hosts can participate in the QoS management associated with this quality channel, and any one host can manage multiple quality channels.

- **Performance and scalability.** The locations of both resources and resource managers are crucial to the efficient implementation of QoS management, as evident from the prevalence of distributed methods for resource management [9, 4], from investigations of different granularities for run-time adaptation [3], etc. With IQCs, no a priori restrictions exist for the relative locations of resources and resource managers. Quality management infrastructures may be adjusted to suit the needs of applications and the characteristics of platforms.

- **Separation vs. integration.** Quality-managed media streams and the control information required for quality management should be separated. Their separate transmission is indicated because media streams are typically isochronous in nature, whereas control flows (e.g., monitoring or adaptation events) are full duplex and asynchronous with media messages. Yet, at the same time, complex distributed applications require an integrated approach for establishing and managing their media and control flows, due to the direct relationships between these flows. IQCs are created and maintained separately from data channels. At the same time, by linking the creation and use of IQCs with the creation and use of sockets, one to one associations of quality events with data may be maintained, if necessary.

- **Transparency and QoS enforcement.** It is desirable to make QoS enforcement transparent to applications, because lack of transparency would force applications to have detailed knowledge about the OS services, resources, and platforms being used. As a result, such QoS-managed applications would be difficult to upgrade or port. In addition, especially for MIM applications where participants dynamically join and leave sessions, their media flows and therefore, their control flows change dynamically. If resource managers do not need to know about the identities and number of other resource managers, such dynamic behaviors are more easily handled. The principle that should be applied is one of *least knowledge*, that is, applications and resource managers should specify or state only the minimum knowledge necessary for their QoS management. This is our main motivation for the *implicit* association of IQCs with communication links.

This paper assumes that MIM communications are inherently *group communications*, where media is *multicast* to multiple participants, the replay of different media streams may have to be *synchronized*, and different media streams may exhibit explicit relationships with respect to their abilities to *share* or even *trade* resources. QoS

management, then, should directly enable such group communications, without requiring applications to deliver identical data to recipients via multiple, individually managed peer-to-peer connections [19]. Moreover, rather than attempting to *isolate* changes in one connection from changes in others, as indicated for peer-to-peer communications, QoS management typically has to be integrated across multiple hosts and streams, enabling one group of streams to trade their resources with another, or to synchronize multiple streams. Finally, we assume that groups are highly dynamic, with members joining and leaving, and also adjusting their QoS requirements frequently.

1.2 IQCs: Implicit Quality Channels

Basic functionality. As stated earlier, an *Implicit Quality Channel (IQC)* is an operating system mechanism that transparently establishes communication paths for the exchange of control information between resource managers in a distributed system. The intent is to enable end-to-end management of QoS, while separating the transmission of media from the exchange of control or adaptation information. Each IQC carries *quality events* [10] between remote and local resource managers, and is established transparently at the time a listening socket is created by the provider of a service (i.e., a *server*). After an IQC has been created, a connection establishment from a remote socket (i.e., a *client*) to this listening socket causes the resource managers at the client to implicitly connect to the IQC associated with the listening socket. This way of configuring quality management in conjunction with socket establishment is called a *server-initiated group.*

A client also has the ability to *override* server-based group initiation, by forcing the server's resource managers to subscribe to an IQC provided by the client. This configuration is called a *client-initiated group.* It gives participants of a distributed media application the flexibility to form groups depending on their specific needs. In either case, after a group has been formed, other servers' and clients' resource managers can join and leave this group dynamically, and all resource managers subscribed to this group can exchange quality events with all other managers within the group.

Clients cannot ignore the IQCs associated with sockets. The intent is to ensure that clients participate in quality management, thus precluding the disruption of a middleware-based quality management infrastructure by non-managed clients. This raises interesting protection and security issues, which we will address in future work.

Our goals with the implementation of IQCs presented in this paper are:

(a) to demonstrate that IQCs and the quality events they carry offer levels of scalability and performance that permits them to serve as the primary means of communication used by the many resource managers and machines involved in distributed multi-party applications; thus, IQCs are efficient building blocks for implementing the varied quality management relationships necessary between the multiple parties involved in MIM applications' media transfers;

(b) to demonstrate that IQCs can link kernel- as well as user-level resource managers, thereby providing a uniform mechanism for building quality management infrastructures that span platforms' machines and their multiple levels of abstraction;

(c) to specifically show that run-time quality management via IQCs, including the run-time adaptation of appli-

cations, can be performed at a fine grain when resource managers can operate inside OS kernels; the intent is to avoid the delays or overheads that can negate the performance advantages sought by dynamic resource management and adaptation;

(d) to hide the creation of groups of resource managers from the application, by associating the implicit creation of quality channels with that of sockets, and thereby, avoiding unnecessary application involvement in quality management; and further,

(e) to allow applications to implicitly subscribe to service agreements or *IQC contracts* with little or no initial negotiation, so as to entirely remove quality management knowledge from certain applications.

There are additional motivations for performing quality management at the kernel level. First, kernel-level resource managers often have awkward user-level interfaces presented to resource managers (also argued in [8]), where multiple kernel calls may be necessary in order to determine the number of resources available for allocation to certain applications. In comparison, within OS kernels, it is straightforward to inspect and manipulate the data structures involved in resource allocation. Second, 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.

Using IQCs. IQCs bundle the resource managers of *related* hosts (or related media streams) into the groups required for coordinated quality management. The need to create and maintain such groups is common. Consider a simple video server, for example, which multicasts identical copies of data to a number of receivers. In this case, resource availability at one client's host can affect the achievable QoS at all other hosts. To differentiate the media streams received by multiple clients, quality management has to have knowledge about the server's and the clients' resources and capabilities. Furthermore, knowledge about multiple media streams has to be available when such streams use *resource sharing* mechanisms [20]. This is particularly important for large-scale MIM applications running on resource-limited infrastructures. The goal of resource sharing is to improve the utilization of these resources while ensuring the QoS guarantees for all connections. Another relationship between multiple media streams exists when they have to be *synchronized*, either to ensure proper replay or to achieve the desired quality of service. Such temporal relationships are typically defined at the time of stream generation, e.g., consider the synchronization between an audio and a video stream [21] or the overlaying of different video streams. Also, in MIM applications it can be necessary to raise the quality of one stream to the cost of all other streams in the same group, e.g., the stream to a chair of a teleconference might be considered of higher importance [22]. Finally, media streams can run through a processing chain at different hosts, where a drop in QoS for the stream between two hosts affects the QoS of the stream on the whole path from stream creation to stream replay [3]. For all such examples, IQCs allow applications to build groups of resource managers that reflect these relationships and allow them to exchange quality events only among the resource managers involved in the QoS management of these streams.

IQCs couple connection establishment with the negotiation of QoS guarantees [23]. Specifically, QoS is defined

by the *service provider* as the service it offers to its clients, and clients implicitly accept these QoS guarantees and constraints upon connection establishment. In contrast, in the Tenet approach to real-time networking [24], the client specifies its traffic model and QoS requirements, whereupon the network (if the admission control succeeds) establishes a channel guaranteeing the QoS. This contrast is due to the fact that connection establishment for MIM applications has to be fast: real-time clients often cannot wait for the end of QoS negotiation and connection establishment before they start to transmit or receive data [25, 26]. Furthermore, with short-lived connections like typical web server accesses, performance would be poor if QoS had to be negotiated for each connection. Another reason for the implicit subscription to an IQC contract is that distributed real-time systems often include applications that have not been developed to have their resources managed or to define a desired QoS. These best-effort applications will receive a QoS definition from the server, where the server tries to maintain this QoS as long as resource utilization allows. Finally, clients are freed from having to know about available resources, OS constructs, etc., at remote hosts if they simply accept the QoS proposed by the service provider.

While implicitly accepted QoS reduces the *call-setup delay* at connection establishment, after setup has completed, new members of the group are free to initiate re-negotiation of the QoS assigned with the service provider or with other members of the quality channel. A similar approach has been suggested in [14], where *profiles* are used to pre-define QoS for continuous media flows. A profile defines the QoS for a particular group of applications.

In the following, we describe the implementation of IQCs, followed by a more detailed description of their functionality and their APIs in Section 3. Section 4 continues with an evaluation and discussion of our implementation, and Section 5 concludes with a summary and an outlook on future work.

2 Implementation of Implicit Quality Channels

IQCs are currently implemented as kernel-loadable modules for Linux 2.4.0. They are comprised of two communication mechanisms and an extension to the standard Berkeley Sockets. *ECalls* is a lightweight interface between kernel services (such as resource managers) and user-level applications. It is used to exchange QoS attributes, information about resource allocations, and exceptions across the user/kernel boundary (also see Figure 1). Specifically, when creating a socket, an application uses ECalls to describe the desired QoS attributes associated with this new socket. Transparent to the application, an IQC is created using the *KECho* group communication tool. KECho is an operating system extension that allows for asynchronous and anonymous event communication within an operating system kernel and across different kernels. The initial subscribers to this KECho channel are the local resource managers that are involved in the QoS management of this particular socket. When a remote application requests a connection to this socket, it implicitly accepts the QoS attributes assigned to this socket and its resource managers are subscribed transparently to the KECho channel representing this IQC.

After two or more hosts have subscribed to an IQC, quality events can be exchanged to notify other resource managers of insufficient quality of service or of re-negotiation requests from applications.

The following two sections describe ECalls and KECho in more detail.

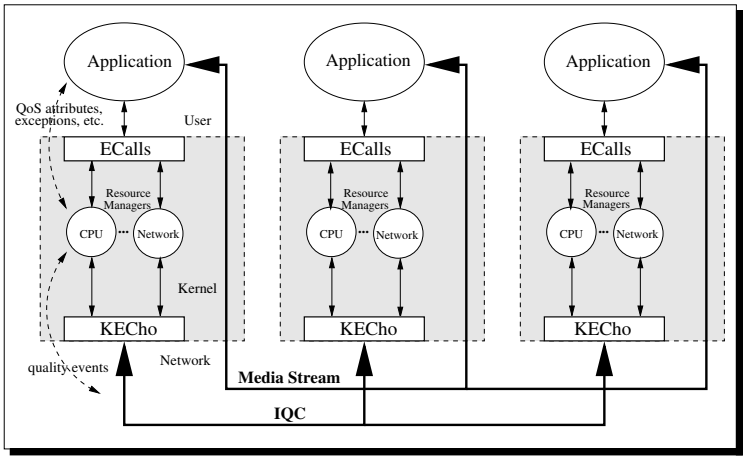

Figure 1: Example of three hosts involved in a media communication: applications and resource managers communicate via ECalls, while resource managers communicate with each other via KECho.

2.1 ECalls

ECalls is an event-based mechanism that links user-space applications with certain kernel-level services, allowing them to share information and events in an inexpensive and flexible way [27]. 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 and/or pending events.

ECalls 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. For instance, an event source can choose the method of event generation on a **per-event** basis. An event sink must register for a certain way of event handling, but can change this registration when desired. For the efficient transfer of data between kernel and user, ECalls allocates two separate memory buffers for each pair of application and kernel service. One buffer is used for data transfer from the application to the kernel, the other for data transfer in the opposite direction, thereby reducing the need for synchronization for single-threaded applications. The sizes and contents of these memory buffers are determined by the kernel service, which supplies this information to the application via a C header file. These memory buffers are pinned into memory to prevent paging and to allow kernel services to access them when the application is not running.

User-to-kernel events.

A *Generic System Call* can be used to produce an event directed at any existing or new kernel service. The memory buffer is used to transmit the system call attributes, including an identifier for the kernel service targeted by the system call. ECalls then uses this identifier to redirect the system call to the corresponding kernel service, which, in response, invokes a system call function. This removes the need to implement new system calls each time a new kernel service is added to the operating system. *Fast User-ECalls* and *Deferred User-ECalls* are both lightweight versions of traditional system calls, i.e., they reduce call overheads by minimizing the actions

performed when returning from the call function, which include signal handling, interrupt handling, and a possible scheduler invocation. Both calls are intended for simple short-running and non-blocking kernel calls, such as updating QoS parameters or polling status flags in the kernel. The overhead for the invocation of Fast-User ECalls is approximately 40% smaller than for regular system calls. Deferred User-ECalls reduce this overhead even more in high load situations through the *batching* of events, i.e., the handler invocation is deferred until the next time the scheduler runs. The advantages are that (1) the function is invoked when the application is already in the kernel, thus removing the need for a user/kernel boundary crossing, and (2) several Deferred User-ECalls between two invocations of the scheduler cause only a single invocation of the handler function, thereby further reducing the overhead. This is useful for situations where events can be aggregated or where only the last event is of use.

Kernel-to-user events.

From kernel to user, ECalls can be used to raise real-time signals to applications, or to invoke handler functions on behalf of applications, where these functions can reside either in user space (*User Handler Function*) or in kernel space (*Kernel Handler Function*). If these functions are non-blocking and short-running, they can even be invoked in interrupt context, otherwise a kernel handler function has to be run in the context of a kernel thread (*Kernel Handler Thread*), which is taken from a thread pool. This approach is similar to the functionality of optimistic message handlers [28]. Finally, ECalls is able to cooperate with the CPU scheduler, currently including both the standard UNIX scheduler and a novel hard real-time scheduler, termed DWCS [29], to influence scheduling decisions in conjunction with event communications (called *ECalls-based CPU Scheduling*). If ECalls has one or more events in the event queue, the event scheduler is invoked each time the CPU scheduler runs. After the CPU scheduler selects the next process, the event scheduler compares the scheduling attributes of this process with the attributes of the sink for the first event in the event queue [30].

2.2 KECho

The second kernel extension, termed KECho, is a port of the event delivery middleware ECho [31] to the Linux operating system kernel. It is a publish/subscribe group communication tool (Figure 2) comparable in functionality to event communications in CORBA[2], but with performance approximating that of socket-based communications.

A process can subscribe to a KECho event channel as a publisher and/or subscriber of events, without the need to know the identities and number of other subscribers. Although event channels are depicted as logically single entities in Figure 2, their implementation is fully distributed. Events on event channels are transmitted directly between peers and are exchanged asynchronously, yet all subscribers of a channel form a *closed* group, i.e., only group members can send/receive to/from this channel. KECho identifies each channel by a *channel ID*, which consists of the hostname and the IP port where the process that created the channel can be contacted.

[2] http://www.corba.org

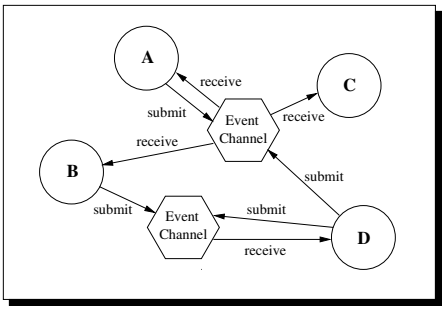

Figure 2: An event channel can have any number of sinks and sources, and any host can be member of multiple event channels.

3 Using Implicit Quality Channels for Quality Management

IQCs connect resource managers of related media streams, enabling them to exchange *quality events* to request adaptation of resource allocations. *Local quality events* are quality events which are being exchanged by resource managers on the same host. *Global quality events*, on the other hand, are events exchanged between resource managers on different hosts. In the following sections, we discuss the architecture and implementation of IQCs, the types of groups resource managers can form with IQCs, and the types of events these resource managers can exchange.

3.1 Issuing Quality Events

After a connection has been established, the resource managers that are members of a group monitor the achieved quality of service. Quality events are raised if a monitoring function returns that the QoS is unsatisfactory (i.e., deviates from the desired QoS) or if a re-negotiation of the quality contract is desired by the application. In most resource managers, monitoring is performed periodically. Similarly, the receipt of events from other resource managers is checked periodically. Specifically, resource managers periodically poll each of their event channels for new incoming quality events, where the frequency of polling affects the overhead and granularity of adaptation. Alternatively, using ECalls, a resource manager can be signaled as soon as a quality event arrives on one of its quality channels. Furthermore, ECalls' ability to *boost* the CPU priority of a process (via ECalls-based CPU Scheduling) can reduce event delivery latencies and alleviate the need for resource managers to poll their quality channels: handler functions can be invoked immediately at event arrival, thereby allowing fine-grained adaptations.

Resource managers can raise the following quality events (Figure 3):

- **Intra-host events** are directed to resource managers on the same host (including the sender of the event). For example, a network resource manager detecting that a buffer for a video server is empty may generate an event to request the CPU resource manager to increase the CPU allocation for this server, or to request from itself to decrease the bandwidth allocation for this stream [10].

- **Forward events** are issued by the host that initiated the creation of an IQC, and are targeted at all other members of the group. Using these events, the initiator can request that channel members adapt their resource allocations or start re-negotiations of the assigned QoS.
- **Feedback events** are issued by group members and targeted at the initiator of an IQC. This is similar to the feedback approach discussed in previous work [21, 2], and it allows channel members to inform the channel initiator about deviations from the desired QoS or requests for changes in QoS.
- **Negotiation events** are issued by channel members and targeted at all other channel members except the channel initiator. The intent is to start a re-negotiation of the QoS for these clients. Intuitively, it might be faster to issue a feedback quality event (only the initiator has to reconsider resource allocations), but if unsuccessful, a client can trade resources or QoS with other clients as a second step. Also, negotiation events facilitate synchronization between clients, e.g., when the replay of a video stream has to be synchronized at multiple clients.

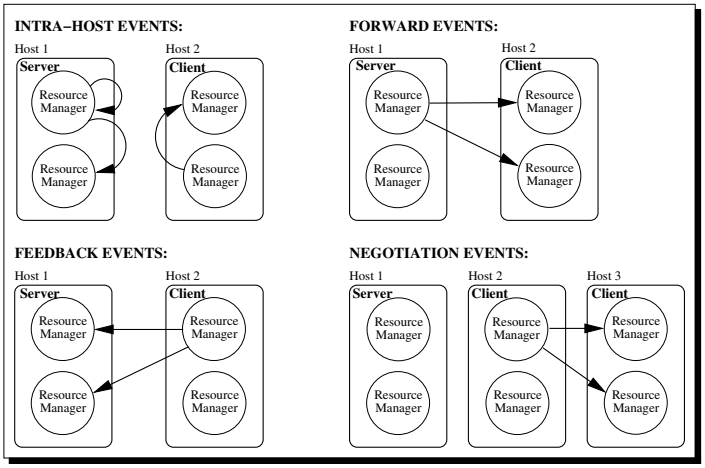

Figure 3: Event types exchanged on IQCs.

With KECho, filters can be used to prevent the sending or receiving of unnecessary events. Such filters can be placed at either the event sinks or the event sources; based on event types or even contents, they can decide whether events should be passed to the receiver or discarded.

3.2 Groups and Synchronization

A *group* is a number of resource managers connected via an IQC, whose membership expresses a certain relationship of the media streams associated with this channel. The goal of the group communication offered by IQCs is to reduce the complexity of distributed QoS management. Groups can be formed for different reasons, e.g., video-on-demand servers want to multicast their video streams to a number of clients. Groups can also express the need for the synchronization of streams, such as:

Inter-sink synchronization, where the replay of one or more media streams has to be synchronized at several receiver hosts. By adding these resource managers to the same group, they can directly inform each other with negotiation events of variations in achieved quality of service and trigger adaptations of resource allocations.
Intra-media synchronization, also called *rate control*, which aims at the proper playback of streams at the receiver. It addresses synchronization between the sender and receiver of a stream. Feedback and forward quality events are the types of events most useful for this kind of synchronization.
Inter-media synchronization, which addresses relationships between different media streams, e.g., for resource sharing or lip synchronization of video and audio. In the case of resource sharing, several streams share the same resources and have to coordinate the use of these resources to avoid interference. Intra-host events and negotiation events are the most useful events for this kind of relationship.

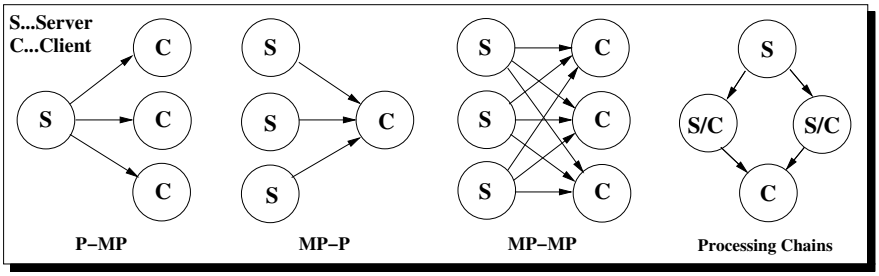

Figure 4: Four types of typical group formations are: Peer-to-Multipeer (P-MP), Multipeer-to-Peer (MP-P), Multipeer-to-Multipeer (MP-MP), and Processing Chains.

Four typical group formations are shown in Figure 4:

- **Peer-to-Multipeer (P-MP).** P-MP groups consist of one server multicasting the same media stream to one or more clients (i.e., *server-initiated group*). Although each of the synchronization scenarios described above can apply to this kind of group, this configuration particularly supports inter-sink synchronization of streams, e.g., receivers adapt the resource allocations to synchronize the replay of the stream.

- **Multipeer-to-Peer (MP-P).** MP-P groups consist of one or more servers transmitting different streams to one client (i.e., *client-initiated group*). This supports resource adaptation for scenarios where several different streams share resources or where the replays of these streams at the clients have to be synchronized (e.g., lip synchronization).

- **Multipeer-to-Multipeer (MP-MP).** MP-MP groups have any number of servers and clients and are useful if, for instance, streams share resources, or if the synchronization between different streams transmitted from different servers to different clients has to be supported.

- **Processing Chains.** Two streams may be indirectly related, if the client of a media stream is also the server of a media stream to a third party. To capture such a relationship, it is useful to bundle these hosts into a group. As an example, consider a stream that is transmitted from a sensor host to a processing host

and then forwarded to a displaying host. In this situation, the frame rates of the two media streams have to be synchronized to ensure proper processing and replay of the video.

3.3 QoS Specification

An application uses ECalls to specify the QoS attributes associated with a socket. The QoS attributes define the desired service quality of a certain media stream and these attributes are *translated* from the *application QoS* to the corresponding *system QoS* [32, 33].

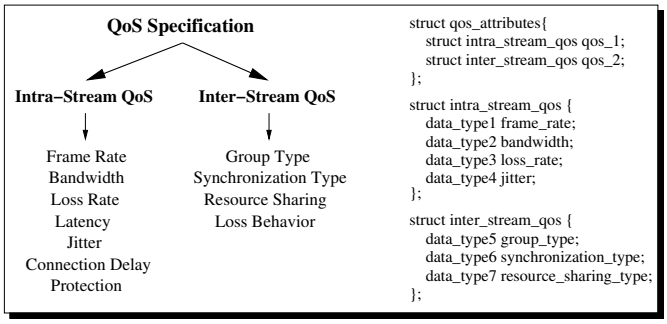

Figure 5: QoS specification

The memory segments offered by ECalls allow the lightweight exchange of information between a kernel service and an application. Here, these memory segments are used by applications to state the desired quality of service. The quality of service associated with a socket is very application-specific and therefore the structure defining the available QoS attributes has to offer different criteria for quality of service and should be easily extendible. We distinguish between two types of QoS guarantees: intra-stream QoS, which describes the quality of a single stream, and inter-stream QoS, which describes the relationship of two or more streams, including the *type* of group communication desired. Examples for both types of QoS attributes are shown in Figure 5.

3.4 Connection Establishment

The creation of a quality channel happens transparently to an application at the time a listening socket is created. Resource managers subscribe to an existing quality channel as soon as an application establishes a connection with this listening socket. However, depending on the desired group formation, there are differences in connection establishment that allow applications to influence the building of these groups. This section describes how these different groups are formed.

Peer-to-Multipeer Groups. In this default situation, an application (the server) associates certain QoS with the creation of a listening socket via ECalls, and KECho automatically generates an IQC associated with this socket. Initial subscribers to this channel are all resource managers on this host that are involved in the QoS management (e.g. CPU, network, or memory manager) as shown in the left image in Figure 6. Another application (the client) can now establish a connection to the server, where the client's resource managers are subscribed to

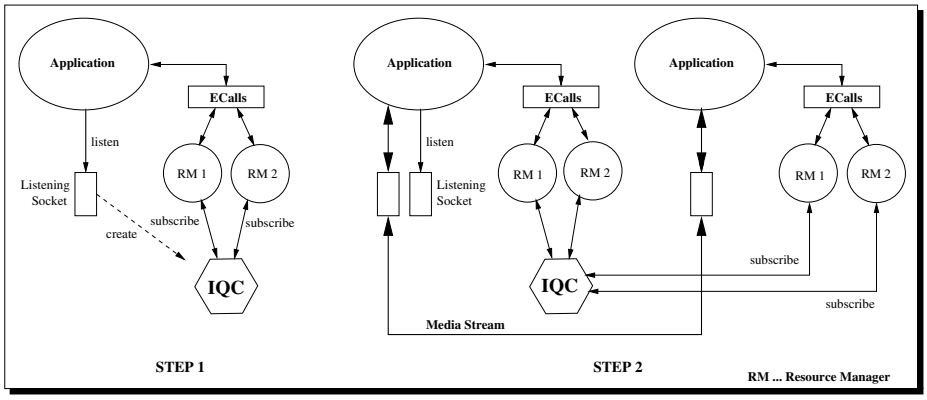

Figure 6: Example of server-initiated group communication.

the IQC associated with the listening socket of the server (right image in Figure 6). This happens transparently, i.e., the application does not participate in any initial QoS negotiation, but implicitly accepts the QoS guarantees offered by the server. However, it is possible for the client to initiate re-negotiations if the client's QoS needs deviate from the QoS offered by the server.

Multipeer-to-Peer Groups. If a client wishes to form its own group (e.g., when several instances of this client at different hosts wish to share resources), an application can indicate this via ECalls through the *group_type* attribute in the QoS attribute list. In this case, the socket creation triggers the creation of an IQC and the server is informed about the channel ID at connection establishment such that the server's resource managers can subscribe to the client's quality channel. However, the QoS associated with this connection is still determined by the server.

Multipeer-to-Multipeer Groups. In this case any number of servers and clients form a group. This kind of group is formed in one of two possible ways: (a) a server can join the group of another server and (b) a client establishing a connection to a server can force this server's resource managers to join a specific group identified by the client. In the latter case, the server's resource managers then contact the initiator of this group and join this group.

Processing Chains. A processing chain exists when the client of a media stream is the server of a second media stream, where these two streams are closely related, e.g., a video server transmits a video to a client that tracks the images for certain objects and sends a modified stream to a third party where the video image and tracking information are displayed. In this case, a drop in QoS for one stream affects all other streams in the chain. In this scenario, the resource managers of all the hosts in the chain subscribe to the first server's IQC.

Resource managers require knowledge about the channel ID to be able to subscribe to an IQC. In the default situation (server-initiated group), the channel ID is send to the clients by the server at connection establishment. The kernel extension at the client catches this information such that an application does not explicitly deal with these IDs. If a client wishes to form its own IQC (client-initiated group), it uses the *group_type* attribute to trigger the creation of a quality channel at the time the socket is created and the client's resource managers inform the server's resource managers about the ID of the client's quality channel. If the subscription to a third party's quality

channel is desired, the channel ID associated with the server's connection has to be obtained together with the server's IP address and port number from a directory service.

A client can only influence the type of group being used. The initial quality contract is always established by the server of a media stream. In typical teleconferencing situations, each participant is a server and a client of media streams at the same time, i.e., the quality of service for the video and audio stream originating from a host are determined by this host itself. However, the quality of an incoming stream is initially determined by the server of this stream. A server tries to allocate the maximum possible QoS to a new client, which can be reduced to the minimum allowable QoS if necessary (e.g., when the number of clients increases). After connection establishment has completed, a client is free to request a quality of service different from the initial QoS. Subsequently, the clients' resource managers can submit quality events to inform the server or other clients of unsatisfactory QoS or when an application requests a re-negotiation of the assigned QoS.

In the following section, we discuss and evaluate IQCs via their use with a teleconferencing tool.

4 Evaluation and Discussion

To evaluate the performance and functionality of IQCs, we have modified the *vic* video conferencing tool from the *Mash* streaming media toolkit[3]. The following microbenchmarks are performed on a 550MHz computer running Linux 2.4.0 and using our kernel extensions that implement IQCs.

4.1 ECalls

ECalls' responsibilities as interface between kernel services and user-level applications are threefold: (a) applications indicate desired QoS attributes for certain connections to the QoS management system, (b) the QoS management system and resource managers transfer information about achieved QoS, status of resource allocations, and exceptions or requests for adaptations to the applications, and (c) CPU scheduling priorities of event-receiving tasks may be temporarily increased to improve responsiveness of I/O-bound tasks. The latter includes the QoS and resource managers, i.e., resource managers receiving quality events via KECho are *boosted* by ECalls such that adaptation can be executed faster and more fine-grained.

Figure 7 shows the overheads of communication in both directions. In these measurements, all calls from user-level to kernel-level simply return the process ID of the calling process, and all calls from kernel-level to user-level access a certain address in memory and return its value. Compared to a standard system call, the generic system call offered by ECalls is slower by 1.3μs. However, a Fast-User ECall requires only 3.1μs, which is 1.8μs less than the latency of a system call. The library call to raise a Deferred User-ECall has a latency of 0.7μs. Deferred User-ECalls are executed at scheduler invocation, i.e., each time the scheduler runs the kernel has to poll for new calls, which adds an additional overhead of 1.6μs in the case there are no calls pending. If the kernel, however, finds one or more calls pending, the overhead is 3μs. In the opposite direction, real-time signals can be used,

[3] http://www.openmash.org

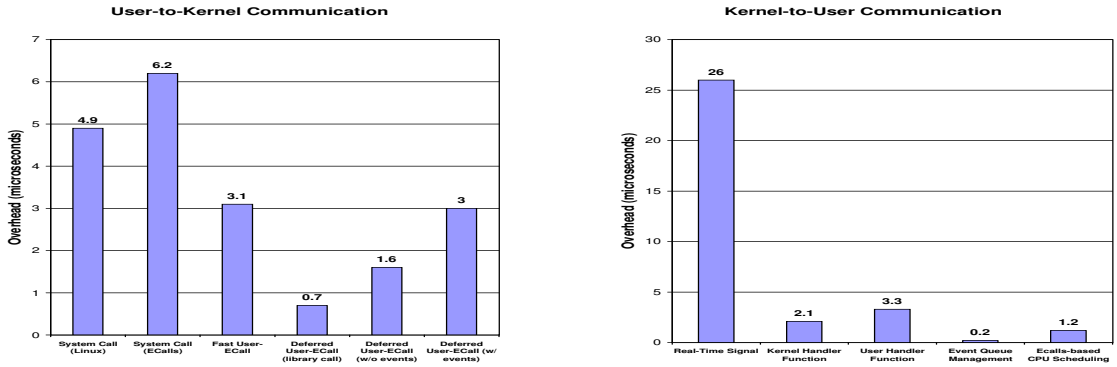

Figure 7: Overhead of User-to-Kernel and Kernel-to-User Communications in ECalls

which are known to be costly in UNIX systems (26μs). The invocation of a handler function in kernel space costs 2.1μs, compared to 3.3μs for the invocation of a user-level handler function. The kernel overhead for event queue management is 0.2μs per event, with 1.2μs additional scheduling overhead to revise the schedule of the standard UNIX scheduler if there are events in the event queue.

4.2 KECho

KECho is the primary component that interconnects the resource managers involved in the QoS management of related media streams. These groups of resource managers exchange quality events over KECho event channels to request QoS adaptation.

Table 1: Overhead of KECho functions.

channel_create()	15ms
subscribe_as_source()	11μs
subscribe_as_sink()	8.8μs
poll_for_events()	22μs
submit_event()	10μs

Table 2 shows the overheads of event channel management in KECho. KECho is optimized for submitting and receiving events, as is apparent from our measurements. For instance, the unoptimized channel establishment protocol, used only once when an application opens a socket, has an overhead of 15ms. In comparison, subscription to a local event channel as a source costs 11μs and as a sink 8.8μs. Finally, polling a channel for incoming events requires 22μs, and the submission of an event to this channel requires 10μs. We conclude from these measurements that IQCs do not contribute significant additional overheads to the communications within MIM applications. Especially, the most frequent and most significant actions, which are the submission and receipt of events, contribute

overheads in the range of 10μs; they are typically overshadowed by the costs of data communications in the millisecond range. However, operations that include remote hosts such as the subscription to a remote event channel can significantly increase the overheads. In our future work, we will pursue additional optimizations of the IQCs channel establishment protocol to further reduce these costs.

4.3 Teleconferencing with Implicit Quality Channels

For the experiments described in this section, the following setup is used. A desktop computer (550MHz) is connected over a 10Mbit connection with a laptop (150MHz), and both are connected via a wireless link of 1Mbit to a second laptop (400MHz), as shown in Figure 8. All computers use the modified Linux 2.4.0 kernel and run the vic video conferencing tool.

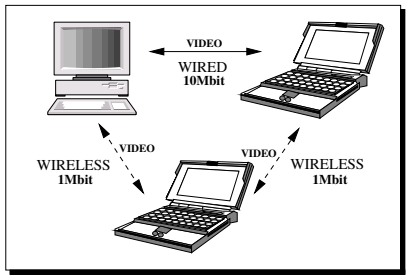

Figure 8: Experimental setup for the video conferencing tool vic.

To evaluate the implementation of IQCs, we implement a simple resource manager that is responsible for (a) acquiring the necessary QoS information from an application via ECalls, (b) admitting or rejecting new connections, (c) monitoring the achieved frame rates for all incoming video streams, (d) generating quality events if the achieved QoS is unsatisfactory, and (e) handling all incoming quality events and adapting the bandwidth allocations for all outgoing video streams. Toward this end, 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 200Kbit each, and we use the Token Bucket Filter (TBF) algorithm to transmit packets. This algorithm is modified such that the resource manager is able to influence the rate of token generation and therefore, influence the rate of packet transmission.

4.4 Connection Establishment and Admission Control

Real-time applications want to minimize the cost of connection establishment and QoS negotiation to be able to send and receive data immediately. In our approach, a client implicitly accepts the QoS determined by the server, i.e., the admission control at the server can know beforehand whether a new request for a connection can be serviced or not and can refuse connections without any further QoS negotiation. In other words, since the server determines the quality of service and therefore the resources associated with a socket, accept/deny decisions can be performed quickly. Consider the following simplified example, where we express the desired quality of a

connection with a service parameter S: a server associates a certain service quality S_{max} with the creation of a socket, which is the maximum allowable service for all connections made through this socket. Furthermore, the server determines a service quality S'_{min} and S'_{max}, which denote the minimum and maximum service quality for each client, respectively. At connection establishment, the QoS manager tries to give a new connection a service quality of S'_{max} if possible, but decreases it to S'_{min} if necessary. Each existing connection i has a quality S_i, which is typically between S'_{min} and S'_{max}, but can be lower than S'_{min} if a previous re-negotiation of the quality of service resulted in a smaller service guarantee for this client. The assigned services S_i for all existing connections i can be lowered to the minimum S'_{min} if the current value of S_i exceeds this minimum value. If Δ_i denotes the *excess service quality* for each connection i with

$$\Delta_i = S_i - S'_{min} \quad \forall S_i > S'_{min}$$

and

$$\Delta = \sum_i \Delta_i,$$

then a connection is **accepted** if

$$S_{max} - \sum_i S_i + \Delta \;\geq\; S'_{min}.$$

Figure 9 shows the two connection establishment protocols available with IQCs. In the first graph, the client

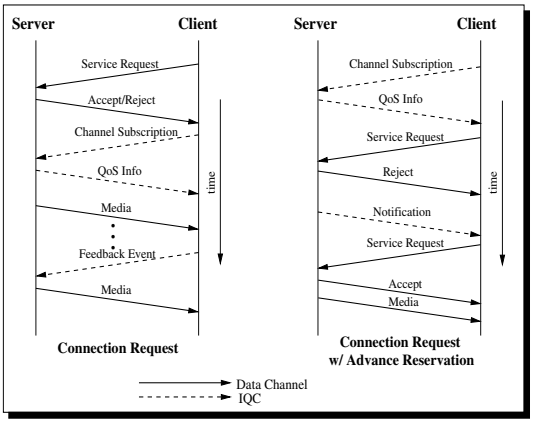

Figure 9: Connection establishment with IQC.

makes a request for a connection, and if accepted, the resource managers of the client subscribe to the IQC of the server (in the case of a server-initiated group). The server's resource managers inform the client's resource managers about the QoS associated with this connection. After that, media streaming can begin, where the resource managers at the client can send *feedback vents* to the server to trigger resource adaptation. As an alternative, a client can first trigger the subscription of its resource managers to the channel owned by the server (if the channel ID is known) and then request a connection. If the connection is established, media streaming can begin, however, if the connection is refused, the subscription is kept alive and the server notifies the client as soon as the request can be fulfilled (*advance reservation*).

4.5 Scenario 1: Peer-to-Peer

The first scenario is a simple client-server structure, i.e., each video server forms a group with a corresponding client. Quality events are exchanged within this group to ensure proper intra-stream synchronization.

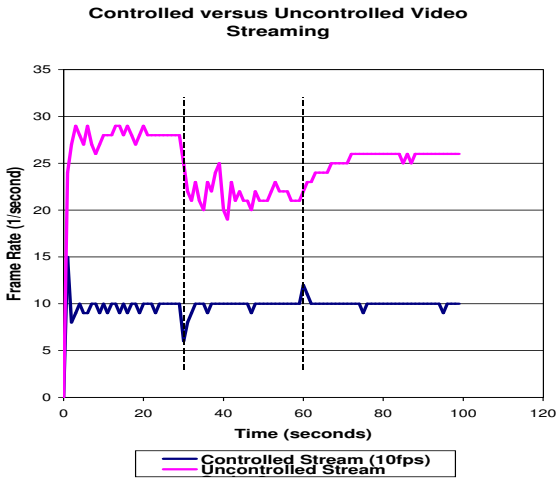

Figure 10: System and network load cause significant variations in frame rates for uncontrolled streaming, while the adaptation mechanism manages to keep the frame rate for controlled streams at a constant value.

Figure 10 compares the controlled and the uncontrolled streaming of a video. The top graph shows the frame rate of the uncontrolled stream at approximately 28fps, varying with system and network load. The controlled stream has a desired quality of service of 10fps, which is shown as the bottom graph. After 30 seconds, we start a *disturber process* (running in an endless **for** loop with the same real-time priority as the video conferencing tool) at the server, which causes the achieved frame rate to drop between 20 and 25fps, while the controlled stream only experiences a short *spike*, causing the frame rate to drop to 6fps for a short time. After 60 seconds, we remove the disturbance, causing another short spike (to 12fps) in frame rate for the controlled stream, whereas the uncontrolled stream increases to more than 25fps.

In the controlled version, we use adaptation of the packet scheduling attributes to achieve the desired frame rate of a given video stream. At connection establishment, the server's QoS manager informs the client's QoS manager about the offered frame rate of the server. A monitor at the client periodically checks the achieved frame rate and transmits quality events to a handler function in the server whenever the desired and the achieved frame rates differ. Table 2 shows the overheads involved with the execution of the monitor and handler functions. In this example, the monitor function simply looks up a socket structure which registers information about the frequency with which frames are read from the incoming socket as a measure for the frame rate.

Table 2: Monitor and Handler Overheads

QoS Monitoring Overhead without Event Generation	15μs
QoS Monitoring Overhead with Event Generation	120μs
Event Handler Activity without Pending Events	55μs
Event Handler Activity with Pending Events	270μs

If the achieved frame rate is satisfactory, no quality event has to be generated, resulting in an overhead of 15μs, which increases to 120μs in the case an event has to be generated. At the server, a handler function periodically checks for new events, causing an overhead of 55μs per handler invocation to poll the connections for pending events. However, if one or more quality events are found, the handler function needs 270μs for event handling. Note that these values have been measured for a simple scenario with only two resource managers monitoring and adapting the bandwidth for the video stream. In general, overheads depend strongly on the number of resource managers involved and the type and implementation of resource monitoring and adaptation.

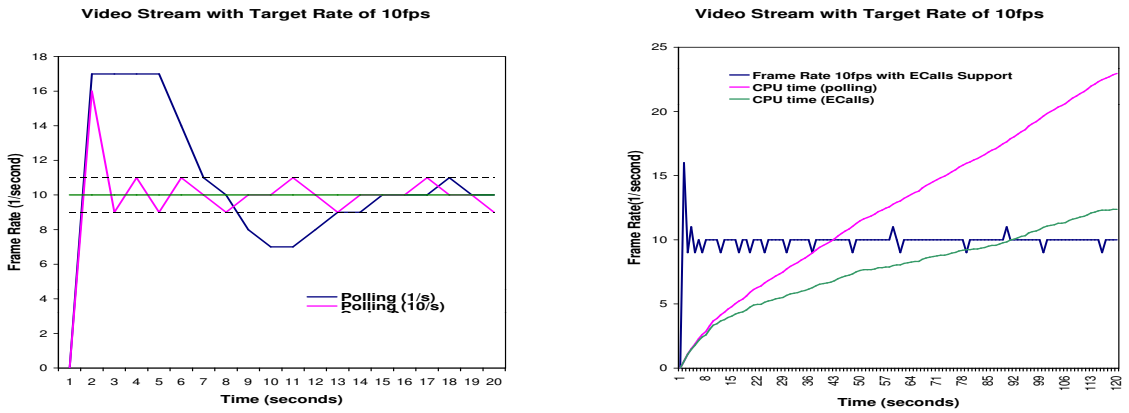

Figure 11: Evaluation of handler overhead, frequency of handler invocation and push/pull methods for event handling.

Furthermore, the frequency of monitor and handler invocations influences both the granularity of the adaptation achievable as well as the overhead of adaptation. Figure 11 shows the behavior of a video stream with a target frame rate of 10fps$\pm$1. In the first graph we invoke the handler function (a) once per second and (b) ten times per second to poll for possible quality events. In the case of a frequency of only 1/second, the video stream needs more than 12 seconds to reach its target frame rate. When we run the handler ten times as often, the stream needs approximately 3 seconds. The same result can be obtained if we do not poll at all, but instead use ECalls

to wake up the QoS manager as soon as a quality event arrives, as shown in the second graph. The second graph also shows the CPU consumption of the handler 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.

4.6 Scenario2: Multi-party Video Streaming

Chain groups are common in situations where several non-parallel streams are in a relationship, e.g., when a media stream is being processed on a separate host before it is transmitted to its final destination. Grouping of server, client, and all intermediate hosts participating in stream manipulation allows us to simultaneously inform all other resource managers directly about changes in the QoS of the whole stream. As an example, if the receiver of a media stream is not able to sustain a certain QoS and drops packets, it will inform the sender of this decrease in QoS, who will adapt its resource allocations and inform the sender of its incoming stream, etc. However, if we join all involved resource managers into the same group, we are able to let a QoS manager directly inform all other participating QoS managers of a drop in QoS such that they can react to this event immediately.

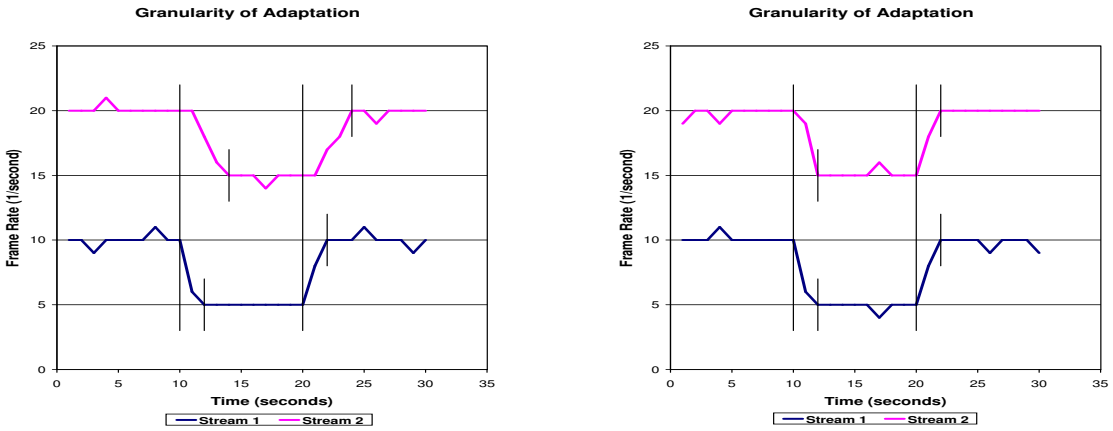

Figure 12: Event distribution influences the granularity achievable in QoS adaptation.

Figure 12 compares two different scenarios, where a server detects that it cannot sustain a certain frame rate and (a) informs the client, who in return informs the second server vs. (b) informs the client and the second server at the same time using implicit quality channels. The first graph shows the frame rates of the two video streams. The desired quality in this example is (a) to service stream 1 with 10fps and stream 2 with 20fps and (b) to keep the frame rate at a ratio of 1:2 between these two streams in case the actual frame rates drop. The server of stream 1 detects after 10 seconds that it cannot sustain its quality of service and drops the frame rate to 5fps, while at the same time informing the client about this new frame rate. The client informs the second server. However, it takes 4 seconds from the time when server 1 detects the drop in QoS until the second stream reaches it new frame

rate of 15fps. We experience the same delay when the frame rate is lifted to its original value at time 20.

The second graph shows the same behavior. However, in this case, server 1 informs server 2 directly about the new frame rate, thus decreasing the delay for the frame rate to drop to its new value to 2 seconds. The results shown in this experiment also apply to the group scenarios M-MP, MP-P, and MP-MP.

5 Conclusions and Future Work

This paper introduces a novel operating system mechanism, termed *Implicit Quality Channels (IQC)*, which supports the QoS and resource management in complex distributed applications. The main contribution of this work is the implementation of an extension to the standard Berkeley Sockets, which – transparent to an application – associates the resource management of media streams with the creation of a socket. Whenever a process connects to a socket extended with an IQC, its inclusion in the quality management infrastructure is triggered transparently and the process implicitly accepts the QoS management associated with that socket. Collaborative quality management actions implemented in this fashion may involve monitoring the behavior of communicating processes and machines, receiving and submitting adaptation events from/to processes, and dealing with their possible requests for re-negotiation of the initial QoS.

In this work, we show that IQCs support QoS by achieving the following goals:

(a) they offer levels of scalability and performance that permits them to serve as the primary means of communication used by the many resource managers and machines involved,

(b) they link applications and resource managers, providing a uniform mechanism for building quality management, infrastructures that span platforms' machines, and their multiple levels of abstraction,

(c) they support run-time adaptation of applications, which can be performed at a fine grain when resource managers operate inside OS kernels,

(d) they hide the creation of groups of resource managers from the application, by associating the implicit creation of quality channels with that of sockets, thereby, avoiding unnecessary application involvement in quality management, and

(e) they allow applications to implicitly subscribe to service agreements with little or no initial negotiation.

Currently, KECho uses both TCP or UDP, however, work is in progress to extend the protocol stack with a reliable version of UDP (termed *RUDP*). Our group has been developing a CPU scheduler and a network scheduler, both based on the DWCS (Dynamic Window-Constrained Scheduling) algorithm [34], which will be used as basis for our future work on resource management. The performance overheads of resource and QoS management mechanisms can significantly influence the quality of adaptation, our future work will include additional optimizations of the IQCs channel establishment protocol to reduce costs of channel management and event transmission. Finally, clients connecting to IQC-based sockets are enforced to participate in a QoS management, which precludes the disruption of a middleware-based quality management infrastructure by non-managed clients. The protection and security issues raised by this will also be addressed in our future work.

Acknowledgments

The video conferencing tool used in this work has been originally developed by the Network Research Group at the Lawrence Berkeley National Laboratory in collaboration with the University of California at Berkeley and is part of the Mash media tool kit. KECho is a kernel-based event publish/subscribe communication mechanism whose code is based on the middleware event communication tool ECho, developed by Greg Eisenhauer. The term *quality events* has been coined by Richard West in his work on a mechanism for quality of service management [10].

References

[1] K. Nahrstedt, H. Chu, and S. Narayan, "QoS-aware Resource Management for Distributed Multimedia Applications," *Journal on High-Speed Networking, IOS Press*, vol. 7, no. 3,4, pp. 227–255, 1998.

[2] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A Feedback-Driven Proportion Allocator for Real-Rate Scheduling," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[3] 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 (RTAS)*, (Denver, USA), June 1998.

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

[5] 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.

[6] A. Campbell and G. Coulson, "A Quality of Service Architecture," *ACM Computer Communications Review*, 1994.

[7] 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.

[8] M. B. Jones, D. Rosu, and M.-C. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," in *Sixteenth ACM Symposium on Operating System Principles*, vol. 31, pp. 198–211, ACM, December 1997.

[9] 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.

[10] R. West and K. Schwan, "Quality Events: A Flexible Mechanism for Quality of Service Management," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, May 2001.

[11] M. Moran and R. Gusella, "System Support for Efficient Dynamically-Configurable Multi-Party Interactive Multimedia Applications," in *Proceedings of 3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1992.

[12] C. Szyperski and G. Ventre, "Efficient Group Communication with Guaranteed Quality of Service," in *IEEE Workshop on Future Trends in Distributed Computing Systems*, September 1993.

[13] L. Mathy and O. Bonaventure, "QoS Negotiation for Multicast Communications," *Lecture Notes in Computer Science*, vol. 882, 1994.

[14] F. Garcia, A. Mauthe, N. Yeadon, and D. Hutchison, "QoS Support for Video and Audio Multipeer Communications," in *Expert Contribution to SC6 WG4 and SC21 - ISO-Meeting, Beppu, Japan*, 1995.

[15] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid," *To appear in International Journal for Supercomputer Applications*, 2001.

[16] C. Aurrecoechea, A. Campbell, and L. Hauw, "A Survey of QoS Architectures," *Multimedia Systems*, vol. 6, no. 3, pp. 138–3151, 1998.

[17] C. Szyperski and G. Ventre, "A Characterization of Multi-Party Interactive Multimedia Applications," *High Performance Network Research Report*, January 1993.

[18] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, 1984.

[19] A. Mauthe, G. Coulson, D. Hutchison, and S. Namuye, "Group Support in Multimedia Communications Systems," *Lecture Notes in Computer Science*, vol. 1052, 1996.

[20] A. Gupta, W. Howe, M. Horan, and Q. Nguyen, "Resource Sharing for Multi-Party Real-Time Communication," in *Proceedings of INFOCOM '95*, April 1995.

[21] S. Ramanathan and P. Venkat Rangan, "Feedback Techniques for Intra-Media Continuity and Inter-Media Synchronization in Distributed Multimedia Systems," *The Computer Journal*, vol. 36, no. 1, pp. 19–31, 1993.

[22] H. Sakate, H. Yamaguchi, K. Yasumoto, T. Higashino, and K. Taniguchi, "Resource Management for Quality of Service Guarantees in Multi-party Multimedia Application," in *Proceedings ot the IEEE 1998 Int. Conf. on Network Protocols (ICNP'98)*, pp. 189–196, 1998.

[23] D. C. Verma, "Decoupling QOS Guarantees and Connection Establishment in Communication Networks," in *Proceedings of Workshop on Resource Allocation Problems in Multimedia Systems (in conjunction with RTSS)*, December 1996.

[24] D. Ferrari, A. Banerjea, and H. Zhang, "Network Support for Multimedia – A Discussion of the Tenet Approach," in *Computer Networks and ISDN Systems*, vol. 26, pp. 1267–1280, 1994.

[25] R. Bettati, D. Ferrari, A. Gupta, W. Heffner, W. Howe, M. Moran, Q. Nguyen, and R. Yavatkar, "Connection Establishment for Multi-Party Real-Time Communication," in *Proceedings of 5th International Workshop on Network and Operating Support for Digital Audio and Video*, April 1995.

[26] D. Ferrari and D. C. Verma, "Scheme for Real-Time Channel Establishment in Wide-Area Networks," *IEEE Journal on Selected Areas in Communications*, vol. SAC-8, 3, pp. 368–379, 1990.

[27] 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*, June 2001.

[28] D. A. Wallach, W. C. Hsieh, K. L. Johnson, M. F. Kaashoek, and W. E. Weihl, "Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 217–225, July 1995.

[29] R. West and K. Schwan, "Dynamic Window-Constrained Scheduling for Multimedia Applications," in *6th International Conference on Multimedia Computing and Systems, ICMCS'99*, IEEE, June 1999.

[30] C. Poellabauer, K. Schwan, and R. West, "Coordinated CPU and Event Scheduling for Distributed Multimedia Applications," Tech. Rep. GIT-CC-01-05, College of Computing, Georgia Institute of Technology, 2001.

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

[32] K. Nahrstedt and J. Smith, "A Service Kernel for Multimedia Endstations," *Multimedia: Advanced Teleservices and High-Speed Communication Architectures*, pp. 8–22, 1994.

[33] R. Kravets, K. Calvert, and K. Schwan, "Payoff Adaptation of Communication for Distributed Interactive Applications," in *The Journal for High Speed Networking: Special Issue on Multimedia Networking*, Winter 1999.

[34] R. West and C. Poellabauer, "Analysis of a Window-Constrained Scheduler for Real-Time and Best-Effort Packet Streams," in *Proceedings of the 21st IEEE Real-Time Systems Symposium*, (Orlando), IEEE, November 2000.