Experimentation with Event-Based Methods of Adaptive Quality of Service Management *

Richard West and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

Many complex distributed applications require quality of service (QoS) guarantees on the end-to-end transfer of information across a distributed system. A major problem faced by any system, or infrastructure, providing QoS guarantees to such applications is that resource requirements and availability may change at run-time. Consequently, adaptive resource (and, hence, service) management mechanisms are required to guarantee quality of service to these applications.

This paper describes different methods of adaptive quality of service management, implemented with the event-based mechanisms offered by the Dionisys quality of service infrastructure. Dionisys allows applications to influence: (1) `how` *service should be adapted to maintain required quality, (2)* `when` *such adaptations should occur, and (3)* `where` *these adaptations should be performed. In Dionisys, service managers execute application-specific functions to monitor and adapt service (and, hence, resource usage and allocation), in order to meet the quality of service requirements of adaptable applications. This approach allows service managers to provide service in a manner specific to the needs of individual applications. Moreover, applications can monitor and pin-point resource bottlenecks, adapt their requirements for heavily-demanded resources, or adapt to different requirements of alternative resources, in order to improve or maintain their overall quality of service. Likewise, service managers can cooperate with other service managers and, by using knowledge of application-specific resource requirements and adaptation capabilities, each service manager can make better decisions about resource allocation.*

Using a real-time client-server application, built on top of Dionisys, we compare alternative strategies for adapting and coordinating CPU and network services. In this fashion, we demonstrate the importance of flexibility in quality of service management.

---

*This work is supported in part by DARPA through the Honeywell Technology Center under contract numbers B09332478 and B09333218, and by the British Engineering and Physical Sciences Research Council with grant number 92600699.

1. Introduction

Complex distributed applications, including groupware[14], virtual environments (e.g., DIVE[5, 18]), distributed games[29], multimedia video and audio, and distributed interactive simulations (DIS)[24] require quality of service (QoS) guarantees on the end-to-end transfer of information across a distributed system. A major problem faced by any system, or infrastructure, providing quality of service guarantees to such applications is that resource requirements and availability may change at run-time. For example, an application may require more or less CPU cycles to identify a target object in a graphical image, depending on the content of that image. Likewise, resource availability may change due to a dynamically changing number of application tasks sharing a finite set of common resources. Consequently, adaptive resource (and, hence, service) management is required to guarantee and, possibly, maximize quality of service to complex applications. Furthermore, it is important that adaptive resource management mechanisms built within a system or infrastructure are aware of the specific needs of individual applications.

Many research groups have proposed different architectures and middleware[4, 13, 27, 20, 2, 26], to provide runtime quality of service guarantees on information exchanged between hosts in a distributed environment. However, the Dionisys QoS infrastructure, that we propose, supports application-specific extensions for quality of service management, by allowing applications to control: (1) *how* service should be adapted to maintain required levels of quality, (2) *when* such adaptations should occur, and (3) *where* these adaptations should be performed. Dionisys supports per-application *quality management*, which is particularly important for large-scale, complex applications, where service quality may depend on multiple service types, requiring that one service type be adapted to compensate for too much or too little service of another type. For example, consider an application that requires both CPU and network services to generate and transfer information to a client on a remote host. In this case, to compensate for insufficient bandwidth, the application can adapt by using more CPU cycles to encode (and compress) the information before it is transmitted. In contrast, if applications have no control over which services are adapted, the system would have to determine an acceptable quality for each of the n active applications, by providing appropriate levels of service in each of m possible service types. The difficult nature of runtime solutions for such a quality maximization problem is exacerbated by the fact that applications' service requirements as well as the system's resource availability vary dynamically.

The Dionisys approach to *quality management*, described in the previous paragraph, is to *specialize* the fashion in which adaptations are performed for each application. This is achieved by allowing each application to share, with specific service managers, the information necessary for runtime service adaptation. Additionally, the application may adapt its own behavior, in conjunction with the service adaptations being performed on its behalf. Such specialized, joint quality management uses an event-based mechanism offered by the Dionisys quality of service infrastructure. As part of this mechanism, applications specify: (1) *monitor* functions that are written to capture service quality at specified times and generate events if service adaptation is required, and (2) *handler* functions, that are executed in response to adaptation events raised by service managers. As a result, applications control which system service is adapted by specifying which service manager executes a monitor and which service manager executes its corresponding control handler. Toward this end, an `event channel' is established between a monitor in one service manager and its corresponding control handler, which is executed by the same, or by another service manager. In this fashion, service managers can provide service to applications in a manner specific to their individual needs, and applications can monitor and pin-point resource bottlenecks, adapt their requirements for heavily-demanded resources, or adapt to different requirements of alternative resources, in order to improve or maintain their quality of service. In summary:

- An event specifies the semantic information needed by a handler to determine *how* to adapt service in order to maintain or improve application-level quality. For this reason, events in the Dionisys infrastructure are termed *quality events*;
- Monitor functions execute at rates specified by applications and generate events *when* service adaptation is required;
- Applications specify *where* monitors and handlers should execute (i.e., within which service managers), in order to adapt service allocation and/or requirements.

Contributions. In this paper, we compare and contrast several adaptation strategies implemented with quality events, thereby demonstrating the *flexibility* of the Dionisys quality of service infrastructure. Specifically, we show that for certain applications, `out-of-band' adaptations are inadequate, thereby necessitating the use of quality events to implement `in-band' adaptations. *In-band* adaptations occur as application-level data is being processed and/or transferred, along the logical path of resources leading to the destination. By contrast, typical *out-of-band* adaptations are not enacted until subsequent data is transferred along the same logical path. Finally, we show that by coordinating service adaptations between a number of service managers, applications can achieve high qualities of service in an efficient manner.

The next section describes the Dionisys approach to quality management in more detail. Then, Section 3 describes issues and tradeoffs in the implementation of different adaptation strategies. Section 4 presents an experimental evaluation of Dionisys, by using a video server application mentioned in Section 2.1 to compare and contrast different adaptation strategies. Related work is described in Section 5, while conclusions are described in Section 6.

2. The Dionisys Approach to Configurable Quality Management

The Dionisys QoS infrastructure (see Figure 1) comprises several key components to support per-application configurable quality of service management:

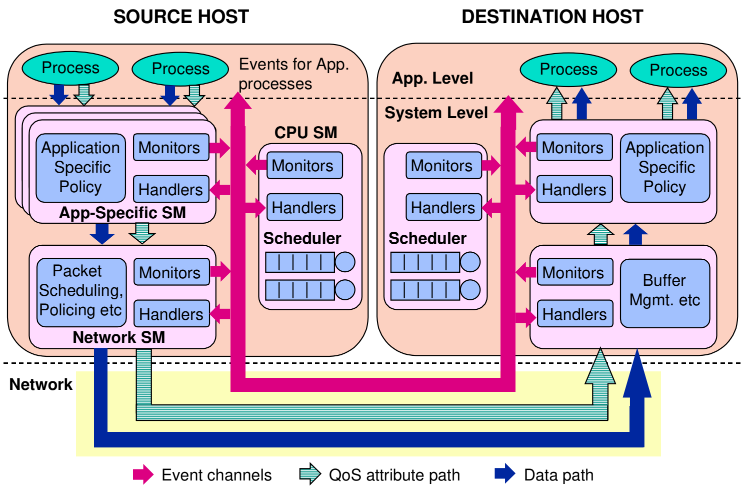

Figure 1. Dionisys QoS Infrastructure

- **Service Managers (SMs).** Service managers execute application-specific monitor and handler functions, as well as service policies that control the allocation of resources to competing application processes.
- **Monitors.** Monitors are application-specific functions that trigger potential service adaptations, if the actual quality of a given service type is unacceptable, or less than desired.
- **Handlers.** Handlers are application-specific functions that influence how service and, hence, the allocation of resources among competing applications is adapted.
- **Quality Events.** Events are generated by the execution of a monitor function when one or more service types are required to be adapted. The attributes associated with an event are used to decide the extent to which service should be adapted in the target service manager. For this reason, events and their attributes are collectively named `quality events'.
- **Event Channels.** Event channels allows quality events to be communicated between monitor and handler routines that execute in the context of different service managers. Note that events can also be delivered to application processes, that can adapt their own behavior.

Figure 1 depicts a number of application-specific service managers, complemented by two more `generic' service managers that control how CPU and network services are delivered to application processes. For example, for the video server described later in the paper, an application-specific service manager could control frame resolution, while the CPU and network service managers control the rate at which frames are generated and transmitted across a network, respectively. Quality events, transported via event channels between these service managers, cause frame resolution and rate to be adapted in keeping with the video server's quality of service requirements.

Dionisys Implementation. Each service manager used in Dionisys maintains two queues: one queue for application-specific monitor functions and another for application-specific handler functions. The monitors pass events to handlers when an application requires its service to be adapted. Events may also be shipped back to the applications themselves to be handled in the corresponding application's address space. As part of a service manager's job, it must execute its monitors and handlers at appropriate times: monitors are executed at application-specific times and handlers are executed in response to events generated by monitors. In general, a service manager runs when it must either execute its service policy, execute one or more monitors, execute one or more handlers in response to pending events, or process requests from applications to create new event channels.

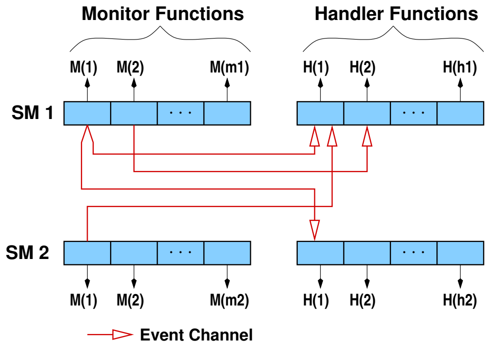

Figure 2. Example event channels involving two service managers, $SM1$ and $SM2$, in Dionisys.

Figure 2 shows an example of several event channels between two service managers, $SM1$ and $SM2$. Each service manager, SMi, maintains an array of pointers to monitor functions ($M(1)$ to $M(mi)$), and an array of pointers to handler functions ($H(1)$ to $H(hi)$). Each service manager executes at most one monitor function and one handler function per application process. An event channel binds one or more handler functions to a single monitor, which generates application-specific *quality events*. Note that there is at most one handler for an event channel in any one service manager, but there can be multiple handlers, spanning different service managers, connected to the same event channel. For example, Figure 2 shows two handlers connected via the same event channel to monitor, $M(1)$, in service manager, $SM1$.

In the current (Solaris-based) implementation of Dionisys, service managers on the same host all execute as kernel-level threads, within the same address space. Application processes access these service managers via library calls that utilize a shared memory application-programming interface (API). The Dionisys API allows applications to create, or delete, application-specific service managers and/or event channels, or to exchange data with existing service managers[1]. The rationale for this implementation is to emulate the way in which operating system services are accessed via system calls from application processes. Furthermore, applications can specialize the operation of Dionisys service managers, or create new service managers, in the same way that kernel-loadable

[1]The details of the Dionisys API are out of the scope of this paper.

modules can be dynamically-linked into an `extensible' operating system[2]. Stated concisely, when an application creates an application-specific service manager (which executes application-specific policies), the service manager functionality is compiled into a shared object and dynamically linked into the Dionisys system-level address space. For the user-level realization of Dionisys used in this paper, this address space is that of a daemon process running all Dionisys service managers. For the kernel-level realization of Dionisys now being developed by our group, this address space is that of the (Linux) operating system kernel. In this fashion, we can incorporate application-specific service managers into Dionisys, supporting configurable protocols and other functions applicable to, and provided by, each application. Finally, when running Dionisys across a distributed system, each host has its own Dionisys daemon process. A simple nameserver uniquely identifies each service manager executing within each and every daemon process. Sockets are then used to communicate application data, QoS attributes and events between these processes on different hosts.

Using Dionisys. Applications specify the functionality to monitor service quality at predetermined times. Typically, monitor functions are executed periodically, at a rate determined by the application[3]. Consequently, monitors influence *when* adaptations occur. Depending on the actual monitor function executed by a specific service manager, an event may be raised to alter the service provided by the same or another service manager. An event is typically raised when the actual (monitored) and required service levels differ by an amount that causes an application to jeopardize its QoS requirements. Alternatively, a monitor might generate an event if it is possible that *better* quality of service can be achieved by adapting one or more service types. Events are passed between or within service managers via `event channels'. *How* an event is handled actually depends on the handler function that is specified by the application, and executed by the target service manager. Furthermore, applications have the ability to control *where* monitor and handler functions are executed. That is, each application using the Dionisys QoS infrastructure has the ability to specify the service managers responsible for executing its monitor and handlers functions. In summary, the `system' establishes an `event channel' between a monitor in one service manager and its corresponding handler, which is executed by the same, or another service manager. Service managers dynamically-link with shared objects containing the monitoring and handling functions at runtime. Observe that the issues of *protection* associated with dynamically linking application-specific monitoring, handling and service manager functionality into Dionisys have been omitted. This work focuses more on the issues of providing a framework to support application-specific extensions for quality of service management.

2.1. An Adaptable Client-Server Application

Figure 3 shows an adaptable client-server application, using the Dionisys QoS infrastructure. Server processes send data to client processes on one or more remote hosts. The CPU service manager allocates CPU cycles to

[2]For example, Linux and Solaris allow objects to be dynamically-linked into the kernel using `insmod' and `modload', respectively.

[3]In practice, there are limitations on the maximum rate at which service can be monitored.

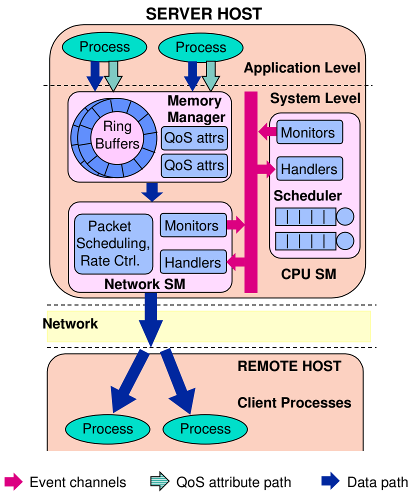

Figure 3. Example adaptable client-server application, with event channels between CPU and network service managers on the server host. Server processes send data to client processes on a remote host.

application processes based on a scheduling policy that is configured when the service manager is initialized. The network manager maintains a buffer for queueing packets of data ready for transmission across a network. Shown in Figure 3 is a memory manager, responsible for allocating space to application data, and QoS attributes associated with that data. In Dionisys, the memory manager is used merely to manage shared memory, so that application processes and service manager threads can communicate and exchange information.

A Video-Server Example. Consider the case where Figure 3 represents a video-server. Server processes could be video streaming processes, that generate sequences of video frames. These video frames are split into packets and placed in ring buffers in shared memory, with one ring buffer per video stream. Associated with each stream are QoS attributes, that specify parameters such as throughput and frame loss-rate. If a ring buffer is backlogged with too many packets, the network service manager can be configured to drop queued packets or discard incoming video frames. Furthermore, the network manager multiplexes packets over the outgoing network link, using one of several packet scheduling policies, such as best-effort/FCFS, static priority (SP), earliest-deadline first (EDF), start-time fair-queueing (SFQ)[21] and dynamic window-constrained (DWCS)[28] scheduling. Once the video frames have been transmitted and received at the destination, client processes decode and play back these frames.

We now discuss several different adaptation strategies that can be implemented in Dionisys, to control the allocation of resources in applications like the one just illustrated. In our cooperation with Honeywell Corporation, we have also developed similar adaptation strategies and experimented with them on an Automatic Target Recognition (ATR) application. Consequently, the adaptation strategies described in the remainder of this paper also apply to other applications, and not just the one described in this section.

3. Adaptation Configurations

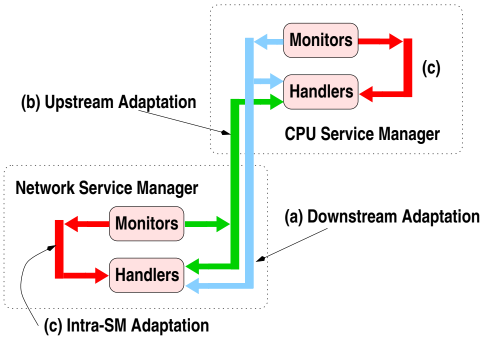

Figure 4. Adaptation scenarios: (a) downstream adaptation, (b) upstream adaptation, and (c) intra-SM adaptation. Observe that in both downstream and upstream adaptation, events are also delivered to handlers within the same service manager from where they originated.

Figure 4 shows the three adaptation configurations considered in this paper. For simplicity we focus on adaptations between CPU and network service managers, but what follows applies to any combination of service managers:

- **Downstream Adaptation** – In this configuration, events are delivered from monitors in the CPU service manager to handlers within both the CPU and network service managers. Intuitively, events from the CPU service manager to the network service manager follow the forward-going path of application data, as it is generated and then transmitted. Observe that, in order to generate data, application processes must acquire CPU cycles from the CPU service manager. Only after data has been generated, can it be allocated network bandwidth for transmission by the network service manager. Consequently, it is possible for downstream events to be delivered in synchrony and, hence, `in-band' with data flowing through the logical path of resources.

- **Upstream Adaptation** – In this configuration, events are delivered from monitors in the network service manager to handlers within both the network and CPU service managers. The term `upstream adaptation' is used to identify the situation where events are delivered in a direction *opposing* the logical flow of application data. This is the case for events from the network service manager to the CPU service manager. Upstream adaptation requires events to be delivered `out-of-band' with the flow of data.

- **Intra-SM Adaptation** – In this configuration, events are delivered from monitors to handlers *within* the corresponding service manager. This configuration does not allow events to be exchanged between service managers, so there is no explicit *coordination* of multiple service types.

Observe that `upstream adaptation' is not to be confused with `feedback control', which is a term commonly used in classical closed-loop control theory. Astute readers may notice that passing events between monitors and handlers is similar to a closed-loop feedback control mechanism, in which the handlers are like control *actuators*. Therefore, all three adaptation configurations can be thought of as using feedback control loops, constructed using `event channels' carrying `quality events'.

We will now show a comparison of downstream versus upstream adaptation strategies, to illustrate their performance tradeoffs. As a consequence of these performance tradeoffs, it is important for a *flexible* QoS infrastructure to support different adaptation configurations.

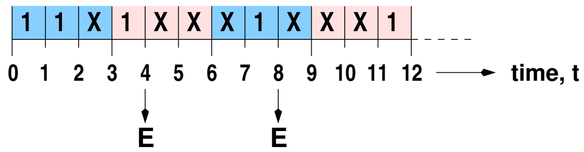

Figure 5. Example CPU schedule. `1' indicates the execution of process, $p_1$, `X' indicates the execution of a process other than p_1, and `E' indicates the generation of an event.

A Comparison of Downstream versus Upstream Adaptation. Figure 5 shows an example CPU schedule. The following example shows the tradeoffs between downstream and upstream adaptations, *ignoring any delays in transferring events between service managers*. As is the case throughout this paper, this example assumes events are transferred only between CPU and network service managers.

Consider the situation where each unit of time a process p_1 (indicated by `1' in Figure 5) executes, it generates one packet of information. Moreover, each packet that is generated has to be transmitted by the network service manager immediately, otherwise it is considered late and is discarded. Suppose, the network service manager must ensure that one packet is transmitted every three time units. Consequently, p_1 requires enough CPU time to generate at least one packet every three time units. That is, p_1 needs at least one unit of CPU time every three time units, assuming there are no overheads in scheduling and transmitting packets that are generated. At the network service manager, the packets generated by p_1 have a corresponding loss-tolerance. Initially, this loss-tolerance is $2/3$, indicating that at most two out of a possible three packets generated every window of three time units can be discarded and, hence, lost.

Downstream Adaptation. Consider now, the situation in which the CPU service manager observes that at time, $t = 4$, p_1 has executed for one time unit in the interval $[t = 3, t = 4]$. At $t = 4$, the CPU service manager schedules a different process (denoted by `X' in Figure 5), which is considered more important than p_1. Furthermore, the CPU service manager does not service p_1 again in the interval $[t = 3, t = 6]$. If the CPU service manager uses knowledge of its future scheduling decisions, it can decide at $t = 4$ to generate a *forward-going* event, which is sent to the network service manager, to try to guarantee that the most recent packet generated by p_1 must be transmitted, otherwise the loss-tolerance of $2/3$ is violated. That is, if the network service manager doesn't transmit the most recent packet generated by p_1, then none of p_1's packets will be transmitted in the interval $[t = 3, t = 6]$. The handler for the forward-going event, executed by the network manager, sets the loss-tolerance of p_1's packets to $0/1$, in an attempt to guarantee that none of p_1's packets are discarded in the next time unit. This reduction in loss-tolerance can be used by the network manager's service policy to schedule p_1's most recent packet before any other packets, *assuming no other packets are more important*. Observe also, at $t = 8$, downstream adaptation generates another event to try to guarantee the packet just generated by p_1 is transmitted.

Upstream Adaptation. Consider the alternative situation, where a monitor in the network service manager, observes that, at $t = 5$, two time units have passed in the current window of three time units, and no packets from p_1 have been transmitted. The assumption here is that, although a packet was generated at $t = 4$, the network service manager did not transmit p_1's packet, because some other packet was considered more important. However, at $t = 5$, an *upstream* event is sent from the network service manager to the CPU service manager. This event triggers a handler in the CPU service manager, to try to ensure that CPU cycles are immediately allocated to p_1, so that it can generate one packet before the current time window has ended. The problem here is that the network service manager is unaware of the CPU service manager's schedule, and p_1 will not be scheduled at $t = 5$. Consequently, no new packets will be generated by p_1, and the original service constraint, that at least one packet every three time units is transmitted, will be violated.

The point of this example is to show that upstream adaptation can perform worse than downstream adaptation, if the allocation of resources by a service manager, earlier in the logical path along which information flows, cannot be determined. In the above example, the CPU service manager is earlier in the information path than the network service manager, because data must be generated before it is transmitted, and generation of data requires CPU cycles. Furthermore, the network service manager does not have access to CPU scheduling information, which is why it cannot know when to generate an event, if such an event is necessary to adapt service in the future. Conversely, downstream adaptation can perform worse than upstream adaptation, if the allocation of resources by a service manager, later in the logical path along which information flows, cannot be determined. In the above example, the forward-going event to the network service manager, that sets the loss-tolerance to $0/1$ only guarantees p_1's most recent packet is transmitted *if no other packets are more important.* If the CPU service manager does not have access to packet scheduling information in the network service manager, then the CPU service manager does not know when events should be generated, if events are necessary to adapt service in the future.

In general, if a service manager, SM_{source}, does not have access to the service information in another service manager, SM_{sink}, then SM_{source} does not know when an event should be generated, if an event is necessary to adapt the service of SM_{sink} in the future. This is true even if the delay in delivering an event from SM_{source} to

SM_{sink} is known, and taken into account by SM_{source} when generating the event.

The above example illustrates a situation where both downstream and upstream adaptation strategies are useful. The flexibility of the Dionisys event-based adaptation mechanism allows both types of adaptation strategies to be implemented, as well as others. The next section evaluates several adaptation strategies using a simple video server application, like the one described in Section 2.1. Moreover, these adaptation strategies emphasize the differences in service quality depending on how service adaptation and coordination is implemented. Again, we focus on the issue of adapting and coordinating CPU and network-level services, using both CPU and network service managers.

4. Experimental Evaluation

We ran a series of experiments, using Solaris 2.6 on a dual Pentium Pro 180MHz PC, serving a client machine connected via 10Mbps ethernet, to show tradeoffs in different service adaptation strategies. These strategies differed in *how*, *when* and *where* service adaptations took place. A video server was constructed as in Section 2.1. Server processes generated video frames, while client processes decoded and played back frames arriving from the network. The CPU service manager used a static priority preemptive scheduling policy, to schedule all application processes (one per video stream), while the network service manager scheduled all packets of video frames from the heads of shared memory ring buffers (again, one ring buffer per stream), using dynamic window-constrained scheduling (DWCS)[28]. Since the network service manager was implemented as a Solaris kernel thread, in the Dionisys daemon process, the packet scheduler did not access the ethernet device directly, but via sockets.

In the first set of experiments, 1000 MPEG-1 I-frames[4] from a `Star Wars' sequence were generated for three different streams, s_1, s_2 and s_3. The QoS attributes associated with each stream were minimum, maximum and target frame rates. The QoS attributes for s_1, s_2 and s_3 were 10 ± 2, 20 ± 2 and 30 ± 3 frames per second (fps), respectively. A QoS violation occurred when the actual transmission rate was out of the specified range for the corresponding stream, at the time it was monitored by the network service manager.

In order to schedule all three streams to meet their QoS requirements, the CPU scheduling priorities were dynamically adjusted. Moreover, priorities were adjusted in accordance with the necessary CPU cycles, at each point in time, to guarantee the required generation rate and, ultimately, the required transmission rate of frames from each video stream.

In these experiments, the following adaptation configurations were compared:

- **Downstream Adaptation.** The CPU service manager monitored each stream's frame *generation* rate: if the actual generation rate was not equal to the target rate, an event was sent to the CPU service manager and (forward to) the network service manager.

[4] 160x120 pixels per frame, and 24 bits per pixel.

- **Upstream Adaptation.** The network service manager monitored each stream's frame *transmission* rate: if actual transmission rate was not equal to the target rate, an event was sent to the network service manager and (back to) the CPU service manager.

- **Intra-SM Adaptation.** Both CPU and network service managers independently monitored each stream's generation and transmission rates, respectively, and sent events to themselves if a stream's target rate was not equal to its actual rate.

In all cases, the monitor functions ran every 10 milliseconds, to ensure that QoS state information was sampled fast enough. The CPU service manager executed the same handler for each stream. The handler attempted to alter the Solaris `RT' (real-time) priority of the corresponding stream-generator process by an amount that was a *linear* function of the difference in target and actual service rates. To ensure one stream was not unduly serviced at the cost of other streams, the CPU service manager executed a *guard function*, which range-checked priority assignments. Rate control in the CPU service manager was triggered, if the highest-priority frame-generator process had an actual rate greater than the maximum required generation rate. This forced the CPU scheduler to run as a *non-work-conserving* scheduler.

The network service manager ran the same handler function for all three streams. The handler function adapted DWCS scheduling parameters to alter the allocation of service in a specific window of time. That is, more or less consecutive packets of video frames from the same stream were transmitted in a given time interval. Rate control in the network service manager was activated if the actual rate of a stream exceeded its maximum transmission rate.

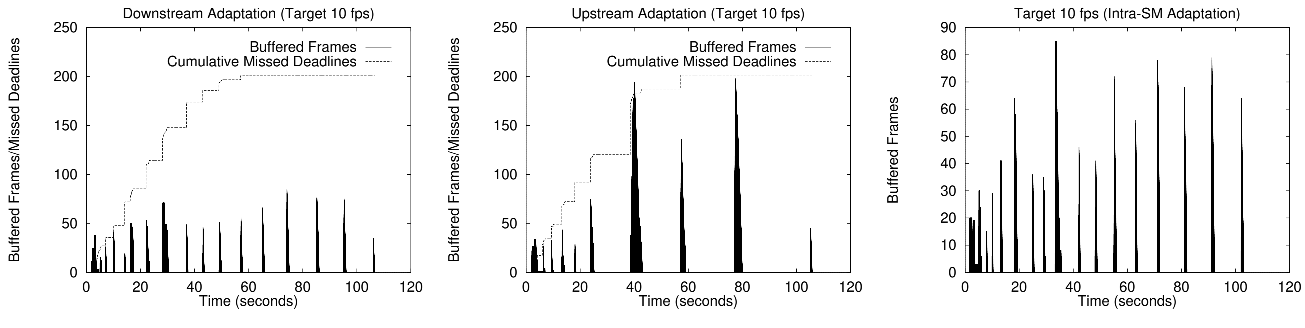

Figure 6. Buffering of the 10fps stream, s_1, with (a) downstream, (b) upstream, and (c) intra-SM adaptation. Cumulative missed deadlines are also shown for downstream and upstream adaptations.

Buffering: Figure 6 shows upstream adaptation has greater variance in buffer usage. The results are shown for s_1, but the same observations apply to all streams, irrespective of their target rate. Downstream and intra-SM adaptations show similar behavior: the maximum number of backlogged frames for a stream and the periods in which the ring buffers are empty is less with both downstream and intra-SM adaptations, than with upstream adaptation. Observe that with upstream adaptation, the network monitor functions raise events too late, to request the generation of more frames. That is, control events are generated when nothing can be done until future data is generated. By contrast, downstream adaptation can lead to adaptation changes `in-band' with the *current* data, as it traverses the logical path of resources, to its destination.

Other researchers have proposed an adaptation strategy for real-rate scheduling[25], that is similar to intra-SM adaptation. In real-rate scheduling, buffer fill-levels are monitored, as opposed to generation and transmission rates. Depending on the fill-level of the buffer, the producer (e.g., CPU SM) would increase or decrease its service, while the consumer (e.g., the network SM) would take the opposite action. This approach to real-rate scheduling requires service managers to share information regarding buffer fill-levels. To compensate for the different delays in accessing this information, the buffer is filled no more or no less than specific levels. However, suppose a service manager accessing this information was the other side of a large latency network. It may be that the latency to monitor the state of the buffer is too large, and adaptations will not be enacted fast enough to meet the necessary quality of service. Such large latencies can cause a buffer to completely fill, or completely empty, thereby affecting service rate. This is what is happening with the upstream adaptations in our experiments. We purposely did not share state information between the CPU and network service managers, thereby observing the cost of communication between threads to coordinate service.

In our experiments, the delay in sending an event to a handler within the same service manager thread was $101.3\mu S$, while it was as high as $10.2mS$ between service manager threads. Inter-thread event handling is more expensive than intra-thread event handling, due to the time-slice of a thread being fixed by the operating system at $10mS$. Consequently, if the network service manager observes CPU service should be adapted (which is the case with upstream adaptation), there is a delay of more than $10mS$ before the service change can be enacted. With rate control, the CPU is the critical resource, in that frames cannot be transmitted if they have not been generated and, in order to generate frames, a frame-generator process needs CPU cycles. Observe that with downstream adaptation, we also send an event from a monitor to a handler within the CPU service manager. The delay in transferring this event to a handler is only $101.3\mu S$. Consequently, CPU cycles are allocated with finer-grained control in downstream adaptation than upstream adaptation. Moreover, with downstream adaptation, events can be sent in-band, and in synchrony, with the flow of data between service managers. This is not possible with upstream adaptation, since service adaptations cannot typically be enacted before subsequent data is transferred along the same logical path. This means the time between when adaptation is necessary, and the time it is applied may involve adaptation on an entirely different data set with upstream adaptation. Consequently, upstream adaptations typically occur out-of-band with the flow of data.

Rate Control: Figure 7 shows the actual transmission rates of all three streams using the different adaptation strategies. As can be seen, upstream adaptation requires more time to reach steady state, and there are larger

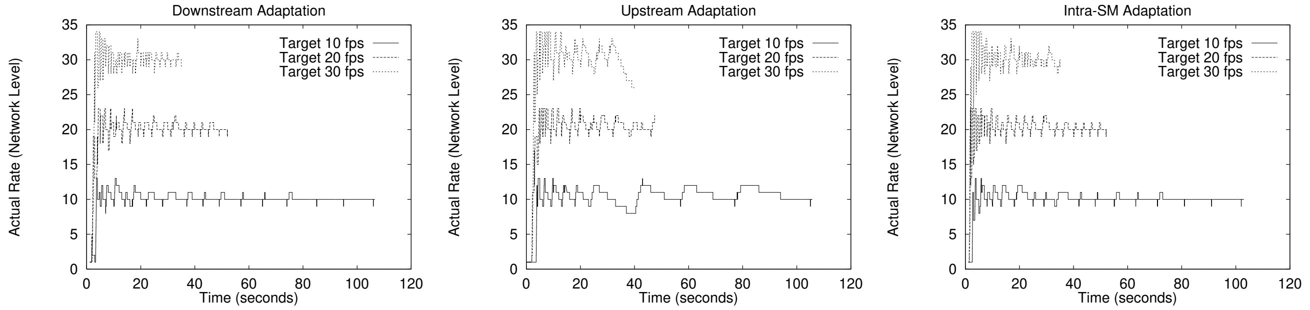

Figure 7. Network service rate with (a) downstream, (b) upstream, and (c) intra-SM adaptation.

peak-to-peak oscillations as the target rate is tracked. With upstream adaptation, there are larger discrepancies between CPU and network-level service rates, which is the reason for larger buffer variations. Furthermore, since the actual and target service rates differ more frequently with upstream adaptation than with downstream and intra-SM adaptations, more events are also generated, as shown in Figure 8(a), for s_3. Note that with downstream adaptation, network-level events are not actually generated. However, Figure 8(a) shows how many network-level events would be generated, due to different target and actual service rates, at the time service is monitored by the network service manager.

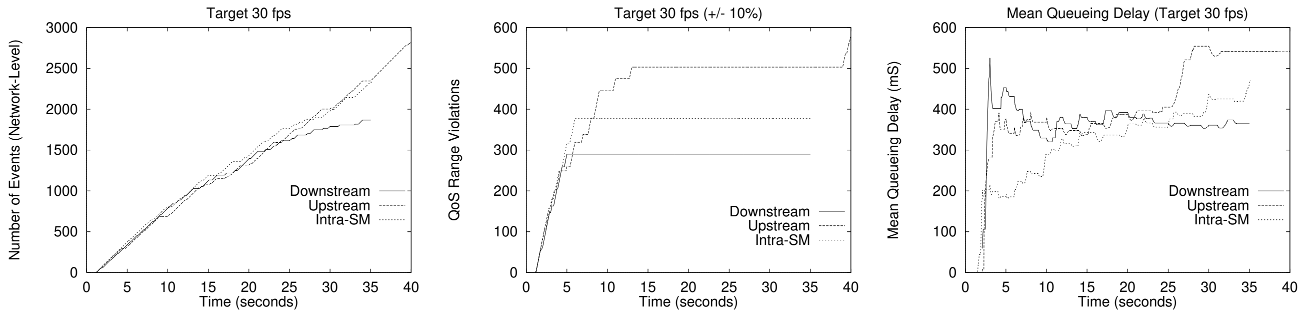

Figure 8. (a) Cumulative number of network events generated by each adaptation method, (b) cumulative QoS range violations, and (c) mean queueing delay versus time for buffered frames, of the 30fps stream, s_3.

QoS Range Violations: Not surprisingly, Figure 8(b) shows that there are most QoS range violations with upstream adaptation. However, intra-SM adaptation is also worse than downstream adaptation. This is because there is no *coordination* between CPU and network service managers. Consequently, intra-SM adaptation requires more time than downstream adaptation to reach the steady-state, in which service rate is within range of the maximum and minimum allowed values.

Mean Queueing Delay and Missed Deadlines: Due to the highest variance in the number of buffered frames with upstream adaptation, there is also a markedly higher variation in mean queueing delay too (as shown in Figure 8(c), for s_3). With upstream adaptation, the sudden increases in delay occur when each stream's queue length (ring buffer fill-level) increases. Furthermore, greater mismatches between service rates at the CPU and network-levels lead to more variation in buffered frames, and more variation in mean queueing delay. As a direct consequence of buffering variations and, hence, queueing delays, there are potentially higher numbers of consecutive frames with missed deadlines. Figures 6(a) and (b) show that downstream adaptation causes fewer consecutive missed deadlines than upstream adaptation, when the deadlines on consecutive frames of s_1 are $100mS$ apart. Upstream adaptation causes a large step in the number of missed deadlines around $t = 40$ seconds. None of the step increases in missed deadlines with downstream adaptation are as large.

In summary, different adaptation strategies are possible with the Dionisys QoS infrastructure, using `quality events', monitors and handlers. Upstream adaptation can lead to poorer quality control, if adaptation latencies are significant. This is analogous to the problem associated with feedback congestion control[16], in that, if the time to inform the producer that it should reduce its generation rate, is greater than the maximum time available to effectively apply such an adaptation, then the consumer can be flooded with too much information. Moreover, upstream adaptation typically occurs out-of-band with the flow of data. With downstream adaptation, the delay between generating a control event and enacting the necessary service change can be coupled with the time to transfer application data along the logical path between service managers. That is, downstream adaptation events can be sent in-band, and in synchrony, with the flow of data. By contrast, intra-SM adaptation works well if: (1) *coordination* between resources and, hence, service managers is not an issue, or (2) there is access to shared state information, which allows groups of service managers to implicitly coordinate their own service adaptations, while avoiding the cost of communicating quality events between one another. Finally, observe that the above adaptation strategies differ in *how*, *when*, and *where* service is adapted. Since no single adaptation strategy is always better than another, it is important for a quality of service infrastructure to be *flexible* in the way it manages per-application quality of service.

4.1. QoS Management

We have shown how the Dionisys QoS infrastructure is flexible enough to support different adaptation configurations, all of which are necessary in different situations. We now show how the flexibility of the Dionisys QoS infrastructure allows quality functions to be embedded into monitors and handlers, to influence how resources are best allocated amongst competing applications.

In this set of experiments, frame-generator processes generated 1000 frames for stream, s_1, 2000 frames for s_2 and 3000 frames for s_3. Again, the target rates of s_1, s_2 and s_3 were 10, 20 and 30 frames per second, respectively. The minimum and maximum service rates were now all constrained to be $\pm 10\%$ of the target rates. Furthermore,

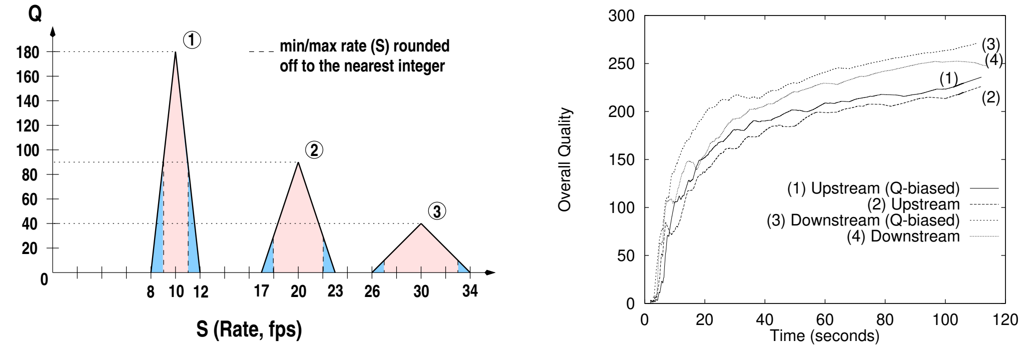

Figure 9. (a) QoS functions for each of the three streams, and (b) overall quality versus time.

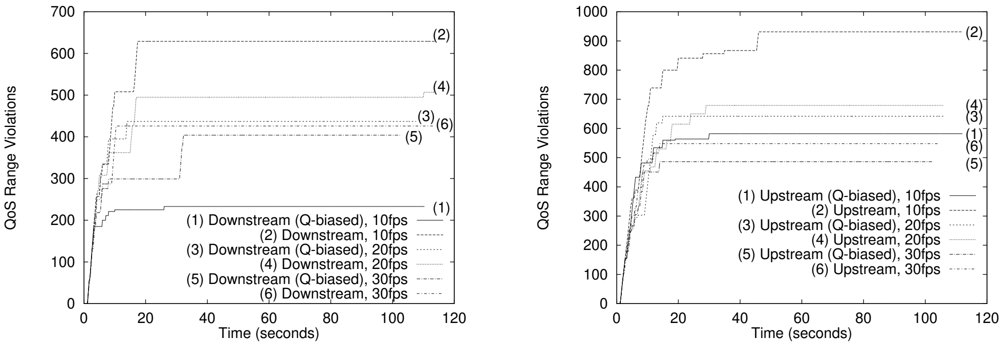

Figure 10. Cumulative QoS violations using (a) downstream, and (b) upstream adaptation.

to model the effect of dynamically-changing resource availability, s_2 and s_3 were blocked for exponentially-distributed idle times, after generating each set of 1000 frames. The exponential idle times averaged 3 seconds.

The same monitoring and handling functions, as in the previous set of experiments, were used here. As before, the CPU service manager executed a handler function that attempted to alter the Solaris `RT' (real-time) priority of the corresponding stream-generator process by an amount that was a *linear* function of the difference in target and actual service rates. The extent to which a Solaris priority was altered was a function of the stream's own quality function.

The actual quality functions that were used are shown in Figure 9(a). These functions show the quality (Q), as perceived by each stream, for the corresponding service rate, S. For example, the priority of s_1 is adjusted by a factor of 3 more than the priority of s_2 (based on the ratio of the gradients of the quality functions for s_1 and s_2). In essence, this implies that s_1 is 3 times more critical than the s_2, when the actual and target service rates of both streams differ by the same amount. Likewise, s_2 is three times more `quality critical' than s_3. These quality functions can be incorporated into monitors and/or handlers of different managers, assuming there exists a functional translation[10, 8, 15] from the service offered by a given service manager to application-perceived quality. Exactly what a given quality means to an application is application-specific.

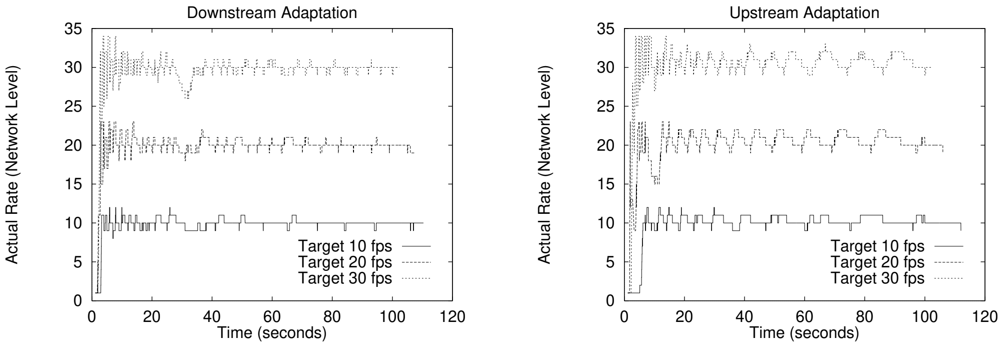

Figure 11. Network service rate, with (a) downstream adaptation and Q-biasing, and (b) upstream adaptation and Q-biasing.

Figure 9(b) shows how overall quality is improved, using different adaptation strategies, when CPU service is biased to the more quality critical stream. The `Q-biasing' graphs refer to the results when the quality functions shown in Figure 9(a) are used. By contrast, all other graphs show results when CPU service is adjusted *equally* for competing streams, even though one stream's service is really more critical than another's. The overall quality, $Q_{overall}$, was calculated from the sum of the average sampled quality of each stream. That is, $Q_{overall} = \sum_{i=1}^{m}(\frac{\sum_{j=1}^{n_{s_i(t)}} Q_j(s_i)}{n_{s_i}(t)})$, where m is the number of streams, $n_{s_i(t)}$ is the number of times the service quality of s_i is monitored (sampled) by time t, and $Q_j(s_i)$ is the jth sampled value of Q for stream s_i. In these experiments, the optimal value of $Q_{overall}$ is 310, which is the sum of the peak values of Q for each stream from Figure 9(a). The overall quality using downstream adaptation and Q-biasing approaches 87% of the optimal value, when $t = 110$ seconds.

Figure 10 shows how the more important streams have significantly fewer QoS violations than the less important streams, for both upstream and downstream adaptation strategies. That is, s_1 has significantly less QoS violations when using per-stream quality functions to adapt CPU service, compared to the case when all streams are treated equally. However, s_2 and s_3 show less improvement using Q-biasing, because they are less *quality critical* than s_1.

Finally, Figure 11 shows network service rates for all three streams, using downstream and upstream adaptations, respectively. As can be seen, s_1 spends less time away from its target rate than both s_2 and s_3, because s_1 is more quality critical. Observe also, the actual service rate of s_1 in Figure 11(b) is more often closer to the target rate than in Figure 7(b). The QoS infrastructure tracks service of the more important streams better, in order to improve overall quality.

In summary, the flexibility of the Dionisys QoS infrastructure allows quality functions to be embedded into monitors and handlers, to influence how resources are best allocated amongst competing applications. Consequently, service managers can use this information to improve overall quality across all applications. This is similar to work done by others, using value[17], reward[1] and payoff[19] functions.

5. Related Work

Many research groups have proposed different architectures and middleware[4, 13, 27, 20, 2, 26], to provide runtime quality of service guarantees on information exchanged between hosts in a distributed environment. However, the Dionisys QoS infrastructure that we propose, focuses on the issues of supporting application-specific extensions for quality of service management, in a flexible manner.

Abdelzaher and Shin[1] propose an end-host architecture for QoS-adaptive communication that can support hard and soft real-time guarantees. A user specifies a QoS contract that consists of a series of alternative QoS levels, the rewards for achieving a certain QoS, and the penalty for violating required QoS. The aim is to maximize the aggregate reward across all users, by selecting the QoS level for each user that achieves the best overall quality as perceived by everyone. Dionisys complements this work by restricting the search-space of services (and, hence, resources) that must be adapted to meet application-specific QoS. Event channels restrict service adaptations to a subset of all the possible adaptations that are possible in a system attempting to maximize overall quality. Furthermore, Dionisys can utilize Abdelzaher and Shin's notion of reward functions, so that service managers can decide how best to allocate service amongst competing applications.

The Darwin project at CMU[6, 7] is aimed at providing network support for application-oriented QoS. Central to this network architecture is the notion of a *service broker*, which is responsible for locating available resources that can potentially be used to meet application requirements, and identifying those resources among all candidates that maximizes overall quality to all applications. In Dionisys, brokers are replaced by service managers, which coordinate resource allocation and adaptation using event channels. In effect, Dionisys' resource management is *distributed* amongst service managers linked by application-specific event channels.

The SWiFT toolkit[11] has been used to construct feedback control components in a modular fashion, to support adaptive resource management[12]. One such application of feedback control using SWiFT is a real-rate CPU scheduler[25]. This is very similar to the way in which the CPU service manager in Dionisys can monitor and adapt its own service on behalf of different applications. Our work compliments this research by showing how Dionisys can support different adaptation configurations (combinations of feedback control loops) to coordinate multiple service (and, hence, resource) managers. In coordinating multiple service managers, resource management is essentially distributed, rather than centralized. We believe this approach is beneficial when the overheads of a global resource manager would be too costly. Observe that a global resource manager has to gather monitoring information about the state of all resources, which may involve passing such information across a wide area network, thereby incurring significant delay. After acquiring such information, a global resource manager then has to decide how best to allocate m resources amongst n applications using hints provided to it by application-specific handlers.

The ability of Dionisys to support application-specific monitors and handlers is similar to the service extensions described in SPIN[3] and the Exokernel[9]. As in SPIN, Dionisys supports the dynamic linking of service extensions, which happen to be linked into the address spaces of service managers. Similarly, the Exokernel supports application-level management of hardware resources by exporting these resources through an interface to library operating systems. Exokernel and SPIN differ from our work by focusing on protection issues, instead of QoS issues for applications with dynamic QoS requirements.

Other related work includes EPIQ[15], which is targeted at supporting end-to-end QoS for complex, real-time applications, that are inherently pipelined (that is, the applications have a series of dependent tasks, where the output of one task is the input to another). This work involves the QoS characterization of pipelined applications and the corresponding management of quality. In fact, the architecture used in the EPIQ project involves brokers for each task in a sequence of tasks that make up the application. Each broker monitors QoS from its producer tasks and negotiates with the brokers of these producer tasks (as well as consumer tasks and resource managers) to maintain QoS. Hence interactions among QoS brokers are confined to those between producer/consumer pairs, whereas Dionisys involves interactions between any service managers linked by event channels.

Finally, our work complements other work at the Georgia Institute of Technology, including FARA[22, 23], and `payoff functions' [19] for characterizing the value of a given quality to an application. Payoff functions can be incorporated into monitors in Dionisys, to influence the type of event that is generated when service adaptations must occur to achieve quality of a specific value. In FARA, an adaptation and resource allocation core makes the decisions about resource reallocation, whereas Dionisys uses application-specific handlers in the service managers to influence such decisions.

6. Conclusions

This paper describes an event-based mechanism in the Dionisys Quality-of-Service infrastructure, that allows applications to specify: (1) *how* observed quality of service should be adapted to maintain required quality of service, (2) *when* adaptations should occur, and (3) *where* adaptations should occur. In Dionisys, service managers execute application-specific functions to monitor and adapt service (and, hence, resource usage and allocation), in order to meet the quality of service requirements of adaptable applications. This approach allows service managers to provide service in a manner specific to the needs of individual applications. Moreover, applications can monitor and pin-point resource bottlenecks, adapt their requirements for heavily-demanded resources, or adapt to different requirements of alternative resources, in order to improve or maintain their quality of service. Likewise, service managers can cooperate with other service managers and, by using knowledge of application-specific resource requirements and adaptation capabilities, each service manager can make better decisions about resource allocation. Using `quality events' , coordinated resource management is possible between multiple service managers, each responsible for the management of one resource abstraction. Having service managers execute application-specific handlers to adapt their own service, essentially distributes control of how resource adaptation should be done. Again, such coordination and control would not be possible without a mechanism like `quality events'.

The *flexibility* of the Dionisys QoS infrastructure enables: (1) different adaptation strategies to be implemented, using `quality events', monitors and handlers, and (2) quality functions to be embedded into monitors and handlers, to influence how resources are best allocated amongst competing applications. Service managers can use quality functions to improve overall quality across all applications.

Finally, several adaptation strategies are compared in this paper: namely, `upstream adaptation', `downstream adaptation' and `intra-SM adaptation'. Upstream adaptation can lead to poorer quality control if adaptation latencies are significant. This is analogous to the problem associated with feedback congestion control[16], in that, if the time to inform the producer that it should reduce its generation rate, is greater than the maximum time available to effectively apply such an adaptation, then the consumer can be flooded with too much information. Moreover, upstream adaptation typically occurs `out-of-band' with the flow of data. With downstream adaptation, the delay between generating a control event and enacting the necessary service change can be coupled with the time to transfer application data along the logical path between service managers. That is, downstream adaptation events can be sent `in-band', and in synchrony, with the flow of data. By contrast, intra-SM adaptation works well if: (1) *coordination* between resources and, hence, service managers is not an issue, or (2) there is access to shared state information, which allows groups of service managers to coordinate their own service adaptations.

References

[1] T. Abdelzaher and K. Shin. End-host architecture for QoS-adaptive communication. In *IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[2] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal, Special Issue on QoS Architecture*, 1997.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[4] A. Campbell. A quality of service architecture. In *ACM SIGCOMM Computer Communication Review*. ACM, April 1994.

[5] C. Carlsson and O. Hagsand. Dive-a platform for multi-user virtual environments. *Computers and Graphics*, 17(6):663–669, November-December 1993.

[6] P. Chandra, A. Fisher, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Resource management for value-added customizable network service. In *Sixth IEEE International Conference on Network Protocols (ICNP'98)*, Austin, October 1998.

[7] P. Chandra, A. Fisher, C. Kosak, and P. Steenkiste. Network support for application-oriented quality of service. In *Sixth IEEE/IFIP International Workshop on Quality of Service*, Napa, May 1998.

[8] G. Coulson, A. Campbell, and P. Robin. Design of a QoS controlled ATM based communication system in Chorus. *IEEE Journal of Selected Areas in Communications (JSAC), Special Issue on ATM LANs: Implemetation and Experiences with Emerging TEchnology*, May 1995.

[9] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.

[10] D. Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine*, 28(11):76–90, November 1990.

[11] A. Goel, D. Steere, C. Pu, and J. Walpole. SWiFT: A feedback control and dynamic reconfiguration toolkit. Technical Report 98-009, OGI CSE, 1998.

[12] A. Goel, D. Steere, C. Pu, and J. Walpole. Adaptive resource management via modular feedback control. Technical Report CSE-99-03, OGI CSE, January 1999.

[13] G. Gopalakrishna and G. Parulkar. Efficient quality of service in multimedia computer operating systems. Technical Report WUCS-TM-94-04, Department of Computer Science, Washington University, August 1994.

[14] S. Greenberg and D. Marwood. Real-time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Cooperative Support for Cooperative Work, ACM press*, pages 207–217. ACM, 1994.

[15] D. Hull, A. Shankar, K. Nahrstedt, and J. W. S. Liu. An end-to-end QoS model and management architecture. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, pages 82–89, December 1997.

[16] V. Jacobson. Congestion avoidance and control. In *ACM Computer Communication Review: Proceedings of the SIGCOMM '88*, pages 314–329. ACM, August 1988.

[17] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *IEEE Real-Time Systems Symposium*. IEEE, December 1985.

[18] R. Kravets, K. Calvert, P. Krishnan, and K. Schwan. Adaptive variation of reliability. In *HPN-97*. IEEE, April 1997.

[19] R. Kravets, K. Calvert, and K. Schwan. Payoff-based communication adaptation based on network service availability. In *IEEE Multimedia Systems'98*. IEEE, 1998.

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

[21] H. M. V. Pawan Goyal and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet scwitching networks. In *IEEE SIGCOMM'96*. IEEE, 1996.

[22] 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.

[23] 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 (RTSS)*, San Francisco, USA, December 1997.

[24] S. Singhal. *Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments*. PhD thesis, Stanford University, August 1996.

[25] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *In Proceedings of the third Symposium on Operating System Design and Implementation*. USENIX, 1999.

[26] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake Distrcit, England, September 1998.

[27] C. Volg, L. Wolf, R. Herrwich, and H. Wittig. HeiRAT - quality of service management for distributed multimedia systems. *Multimedia Systems Journal*, 1996.

[28] 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. Also available as a Technical Report: GIT-CC-98-18, Georgia Institute of Technology.

[29] R. West, K. Schwan, I. Tacic, and M. Ahamad. Exploiting temporal and spatial constraints on distributed shared objects. In *Proc. 17th IEEE ICDCS*. IEEE, 1997.