Interactors: Capturing Tradeoffs in Bandwidth versus CPU Usage for Quality of Service Constrained Objects *

Richard West and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

Complex distributed applications, including virtual environments, and real-time multimedia require performance guarantees in the end-to-end transfer of information across a network. To make such guarantees requires the management of processing, memory, and network resources. This paper describes the Dionisys end-system quality of service (QoS) approach to specifying, translating, and enforcing end-to-end, object-level QoS constraints. Dionisys differs from previous work on QoS architectures by supporting QoS constraints on distributed shared objects, as well as multimedia streams. Consequently, we introduce 'interactors', which capture the QoS constraints and resource requirements at each stage in the generation, processing, and transfer of information between multiple cooperating objects. Using interactors, Dionisys is able to coordinate both thread and packet-level scheduling, so that information is processed and transmitted at matching rates. However, there are tradeoffs in the use of CPU cycles for scheduling and the need to meet QoS constraints on the information transferred between interacting objects. We show, empirically, the packet scheduling frequency to minimize CPU overheads while maximizing bandwidth usage.

*This work is supported in part by DARPA contract #B09332478, and by the British Engineering and Physical Sciences Research Council with grant #92600699.

1 Introduction

Researchers are currently investigating system support for complex distributed applications, including groupware applications [1], virtual environments (e.g. DIVE[2]), and distributed interactive simulations (DIS) [3]. A DIS, for example, provides a group of users with the illusion of a single, coherent virtual world, although its computations may actually execute on distributed, heterogeneous hosts, and its data sources may be derived from physically separated real-time sensors. Simulation *entities*, or objects, such as vehicles, virtual sensors, or obstacles may be generated dynamically or exist as stored state jointly defining the virtual world and participating in the simulation. Each user is exposed to a potentially unique 'view' of this virtual world, unaware of the physical location of other users and computers physically controlling and modeling various entities.

Quality of service constraints in a DIS application are formulated at the level of simulation objects. Moreover, these constraints tend to be highly dynamic. Namely, they may change from one object invocation to the next, they may change at intervals determined by simulation speed, by reaction rates of human participants, or by the possible rates of change in input behavior. For example, a virtual flight navigation system might require one user piloting a plane to be updated frequently with complete, detailed (loss-free) state information about other planes in the nearby air-space, to avoid collision. When two plane entities are sufficiently far apart, their state-changes with respect to position, orientation and graphical detail may not need to be updated as frequently. Furthermore, as two objects move apart in the virtual world, each user controlling one of the entities might only require approximate detail (rather than highest resolution graphical detail) about the other entity. Similarly, when viewing continuous sensors in this virtual world, certain sensors might be important at some particular time (e.g., a radar sensor when targeting an enemy plane) whereas other sensors might be irrelevant (e.g., a GPS location indicator during enemy engagement). Moreover, sensor visibility changes over time, depending on pilot viewpoint, for example.

It has become apparent, not only from DIS applications but also our own work on shared virtual worlds[4], distributed games[5], and battlefield simulations, that most such applications exhibit dynamic QoS constraints. These dynamic QoS constraints must be utilized to attain necessary levels of end-to-end performance. For instance, it is neither reasonable nor viable to display continuously all possible sensor readings to all participants in a virtual world simulating rooms in a Navy vessel described in [4]. In addition, given the complexity of future battlefield simulations in which terrain, participating actors (like tanks), and sensors must be simulated realistically, it will not remain viable to replicate completely all objects across all participating machines. This is especially true when such objects arrive and depart dynamically or when participating hosts are situated in the field (ie., they are 'thin' hosts) and/or must communicate across relatively 'weak' communication links. In summary, it must be assumed that each host maintains a cached (or shadow) copy of the state information about an entity modeled at a remote host. These facts also imply a linkage between information consistency and QoS constraints: (1) a suitable consistency protocol must be used to maintain the correct level of consistency between replicated state information, and (2) timing or quality-of-service (QoS) constraints must be placed on the update and access to distributed shared objects whose state has been partially replicated[5]. This linkage has been recognized by other researchers, as well, including work by Yavatkar et al[6].

The *interactor* abstraction offered by Dionisys captures the end-to-end QoS constraints experienced in the highly dynamic distributed applications targeted by our work. An interactor is an association between cooperating shared objects spanning a network and all the resources, both communication and computation bound, that support object exchanges. Interactors capture the QoS constraints and resource requirements at each stage in the generation, processing, and transfer of information between multiple cooperating objects. Using interactors, Dionisys is able to coordinate both thread and packet-level scheduling, necessary to maintain service constraints on information exchanged between application-level objects. In an end-to-end QoS system like Dionisys, it is often necessary to invoke the packet scheduler at the correct rate to a) meet the service constraints on multiple objects sharing the same network link, and b) keep the network pipe as full as possible when there is information to transmit. The problem is that by executing the packet scheduler too frequently, CPU cycles are wasted due to executing the scheduling algorithm, which reduces the number of CPU cycles in any time interval that are available to other threads. This is an example of one of the resource tradeoffs experienced in complex applications, where multiple resource types must be allocated and managed to maximize the service level granted to all activities.

Traditional QoS research has focussed almost exclusively on the network level (e.g., the Tenet work[7]), without also providing support for end-to-end constraints. Recent work in end-to-end QoS support, such as QoS-A (Campbell et al[8]), and the End System QoS Framework at Washington University [9] supports application-specific QoS constraints, and the mapping of these constraints to the network level. However, these architectures are tailored to multimedia video and audio applications. In comparison, we are interested in complex applications like those described above, which exhibit dynamic generation of information, and require updates to information exchanged between cooperating, distributed shared objects in real-time.

Applications such as those described earlier are not fully addressed by current research on real-time object-based systems, which have focussed on 1) how to associate real-time constraints with a single object or a single invocation [10, 11], 2) how to translate or support some object-level constraint on a single invocation to some thread/communication constraints, and then use conventional scheduling techniques to deal with real-time constraints [11, 12], or 3) how to model/control some entire object-based real-time system 'from the outside'[12, 13]. In contrast, Dionisys is a runtime infrastructure that provides the necessary abstractions to maintain QoS constraints in synchrony with each application's execution. Furthermore, Dionisys provides control over both CPU and network resources required by cooperating objects.

In the next section, we characterize the QoS constraints on objects in applications like those described above. We then show, in section 3, the necessary stages in the translation of object-level constraints down to the network-level. Section 4 describes the Dionisys system and related issues in the support of end-to-end QoS constraints. An experimental evaluation of the system is given in section 5, while conclusions and future work are described in section 6.

2 QoS Constraints on Objects in Distributed Real-Time Applications

Applications like distributed interactive simulations and virtual environments have the following object-level characteristics:

- *Dynamic exchanges between objects* - information exchanged between objects interacting across a network is generated dynamically, as a result of changes in the working environment of the application. For example, information is exchanged between cooperating objects as a result of two entities moving within range of each other in a virtual environment application.

- *Variable rates of exchanges between objects* - object exchanges are often bursty. Namely, there are periods of time when very few exchanges take place, whereas at other times, many exchanges may occur between objects. Factors affecting exchange rates include the proximity of entities in virtual environments, such that the frequency of updates about the location of such entities increases as entities move closer together.

- *Variable resolution of exchanges between objects* - the information exchanged between objects may vary in detail, depending on the importance of detail at the time information is exchanged, for instance. If two entities in a virtual environment or DIS application are semantically 'closer', it might be necessary to increase the resolution of information exchanged between the objects corresponding to each entity.

QoS constraints on information exchanged between objects in applications like those described earlier include:

- *Delay* - this specifies the total tolerable end-to-end delay on information exchanged between objects, before the information is no longer valuable.

- *Loss-tolerance/fidelity* - at the object-level, this constraint specifies the level of detail of information that must be exchanged. Consider a system that maintains consistency of shared objects to some degree. It may not always be necessary to exchange the most up-to-date copy of a shared object. Instead, it may be acceptable to exchange object information that was known to be up-to-date x time units in the past. Furthermore, it may not be necessary to ensure complete coherence between multiple copies of an object. Fidelity constraints bound the degree of loss or detail of information exchanged between objects.

- *Throughput* - it is necessary to specify the amount of information to be communicated amongst hosts per unit of time. Throughput is bounded by the maximum and minimum frequency of updates to shared objects of known granularity (size).

- *Consistency constraints* - these constraints bound the degree of coherence between shared objects. Strict coherence of objects limits scalability in terms of the total number of objects that can be replicated, since more object exchanges are required as the number of replicas increases. The actual format of these constraints could be in terms of some *semantic function* [5], which specifies the conditions about *when* and *with which objects* updates should occur. The specification of these semantic-based consistency requirements can only be made in the context of the application. For virtual environment applications, we can impose not only 'temporal' constraints (which specify *when* updates should be applied) but also 'spatial' constraints on consistency. Spatial consistency constraints ensure that updates are applied to a copy of a shared object only when that object is within a certain range of another process's (user's) object. For example, one user ($p1$) manipulating one object might not care to know about another user's ($p2$) object but $p2$ might want to know about $p1's$ object in shared space because it is within a range of interest to $p2$.

Information exchanged between objects is carried in invocations, which are ultimately scheduled and then marshalled into messages. At the network-level, messages are fragmented into packets and scheduled for transmission with packets carrying information for other object interactions. The following section describes the mapping of object-level constraints to the network level in the context of Dionisys.

3 Mapping Object-Level Constraints to the Network Level

Dionisys supports a layered hierarchy, where three layers of abstraction are used for shared object applications like those described in sections 1 and 3. Namely, the layers are the application/object layer, consistency management layer and then the network layer. Within each layer, there are 'modules' of code that generate and process messages to exchange information between objects. For example, in a virtual environment application, a user maneuvers her vehicle 'object' across virtual terrain which causes the consistency management layer to exchange the modified state about this object with appropriate objects in other processes. The consistency management layer must perform certain operations, such as executing a user's semantic function to determine 'when' and 'with whom' to exchange, as well as generating messages that hold the state information, ordering and buffering messages not yet ready for transmission, dispatching messages that are ready to be transmitted, and buffering/delivering messages received from the network layer. Similarly, the network layer has to perform fragmentation and reassembly of messages, as well as various traffic management functions. Aurrecoechea et al [14], give an excellent summary of the key traffic management mechanisms. Some of these will be described later, in the context of Dionisys.

At each level in the system, QoS constraints on the information exchanged between objects are translated to a form suitable for the next layer. We begin by describing the translation of object-level constraints to consistency-level constraints.

3.1 Mapping Object-Level Constraints to Consistency-Level Constraints

Consider a virtual environment application, as described earlier, in which QoS constraints consist of:

- *throughput*, (T) - defined as $T = \frac{sn_{max}}{t}$, where n_{max} is the maximum number of updates over some time, t, and s is the size of each shared object (assumed constant for simplicity);
- *maximum allowable end-to-end (application-level) delay*, (δ); and
- *fidelity*, (f) - expressed as the minimum fraction of information exchanged during updates. For simple 'memory' objects, this will be the minimum fraction of replicated object state information.

For the throughput specification, the application-level attributes are (n_{max}, t) and s. From these two attributes, the actual throughput T is derived as shown. The consistency protocol need never exchange more than $\frac{n_{max}}{t}$ update messages per unit time with each remote object. Whenever the consistency protocol decides it is time to exchange information, it must ensure each message carries at least the fraction of information specified by the fidelity constraint, i.e. the minimum information conveyed in each update message is fs. Finally, the consistency protocol must subtract the time it spends preprocessing messages for transmission (i.e. header insertion and extraction into the outgoing and out of incoming messages, respectively), and buffering message updates from the total end-to-end delay parameter.

Note that at any point in time, an application process may dynamically change any of its QoS constraints in terms of f, δ, n_{max} and s. Changes to s could be conveyed to the underlying consistency protocol to allow updates to be *aggregated* for multiple objects into one message. In this way, the actual objects themselves remain the same size, but multiple object exchanges are transmitted together. Alternatively, s could be increased or decreased to vary the amount of information associated with an object e.g. more or less graphical information about an object in a 3D virtual environment.

We now extend this example to include specification of *when* updates to shared objects should be exchanged between *which* objects in remote processes. The application-level processes specify a semantic function that describes to the underlying consistency management system exactly the 'when' and 'with whom' constraints. This function can also be used to dynamically adjust the four parameters described above: f, δ, n_{max} and s.

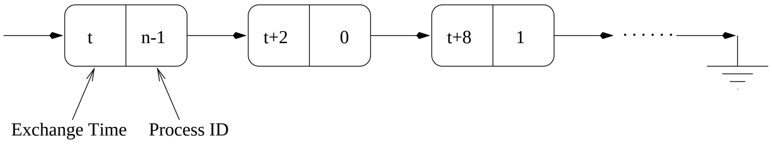

Figure 1: *A sample exchange-list. Each list-member is an (exchange_time, process_id) pair. Only those objects requiring future exchanges need their corresponding (exchange_time, process_id) pairs in the list. The list is ordered by earliest exchange-time first and not process id.*

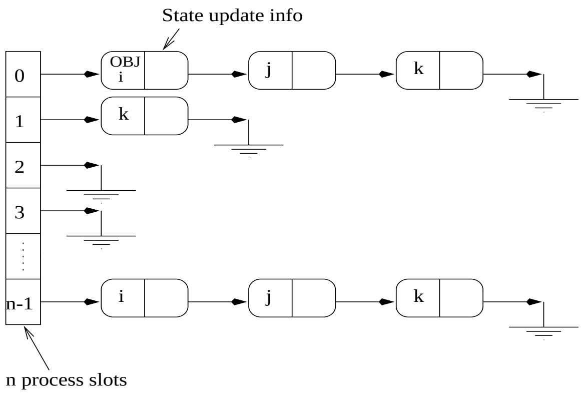

Figure 2: *A sample slotted-buffer at process 2. Each slot may have a unique set of object updates to eventually send to the corresponding process. In this example, process 3 is not in the exchange-list so the consistency management system does not need to buffer and, hence, send updates to process 3. Similarly, we never buffer updates for the local process, shown here as process 2.*

The consistency management system maintains an exchange-list, which is a time-ordered list of $(exchange\_time, process\_id)$ pairs (for example, as shown in Figure 1). Note that the 'when' constraints specified in the application-level semantic function place a *lower* bound on the time between successive updates sent between objects in pairs of processes. The upper bound on updates sent between processes depends on the value of n_{max}. In fact, the minimum time between successive $(exchange\_time, process\_id)$ pairs for the same $process\_id$ in the exchange-list must be $\frac{t}{n_{max}}$. Associated with the exchange-list is another structure: a queue of update messages that are destined for each remote process (i.e a slotted-buffer with one slot for every system-wide process as shown in Figure 2).

These two structures enable the consistency management system to control *when* and *with whom* updates occur, based on semantic information provided by the application in the form of a semantic function. This is exactly the approach taken in the S-DSO system [5]. The buffered update information destined for each process depends on the fidelity constraint f applicable at the time the exchange is due to take place.

In summary, the attributes specified at the application level are: (n_{max}, t), s, δ and f, as well as implicit 'when' and 'with whom' constraints derived from a user-defined semantic function. At the consistency management level, n_{max}, s and f are unchanged, but s and f are used to calculate the minimum state information to be carried in update messages. (Note that the maximum information in any one message is s). δ is translated into δ_c, (where $\delta_c < \delta$) since we need to consider the worst-case overhead (which we shall call d_{c_max}) of executing the consistency management protocol and possibly buffering update messages before they are eligible for transmission. The 'when' constraints define exactly when a message is ready to be exchanged, and δ_c is used to place an upper bound on the valid interval from when the object is modified at the local process to when the destination process must see the updated copy. As far as the consistency management layer is concerned, a messaged is exchanged when there is at least one buffered object for a remote process, and the *exchange_time* with that process has been reached. By exchanging a message, the consistency management layer passes the message down to the network layer.

3.2 Mapping QoS Constraints from the Consistency Management Level to the Network Level

The update messages passed down from the consistency management layer to the network layer might be very large, comprising kilo- and even megabytes of information. These messages will be fragmented into smaller (fixed-size) packets at the network level[1], and each fragment must be assigned a QoS constraint representative of the QoS constraints across the whole message.

We now address the issue of translating constraints from the consistency management level to the network level.

Translating delay requirements: Ferrari [15] states that the method for translating delay bounds has to deal with the effects of delays in the individual layers, and the effects of message fragmentation on the requirements. Let d_{n_max} be the maximum processing delay in the network-level code, due to message fragmentation/reassembly, packet-header generation and removal, and

[1] The term 'network layer' is used loosely to describe the communication-oriented layer underneath the consistency management layer and might better be described as the transport layer. This work is not confined to one particular network protocol.

packet scheduling. δ_n is the maximum tolerable network delay i.e. latency to communicate a complete message across a communication link between two network-level endpoints. Let d_{c_max} be the maximum delay incurred by a message in the consistency management layer. Thus:

$$\delta_c = \delta - d_{c\_max}$$

$$\delta_n \approx \delta - 2 * (d_{c\_max} + d_{n\_max})$$

$$\delta_n \approx \delta_c - d_{c\_max} - 2 * d_{n\_max}$$

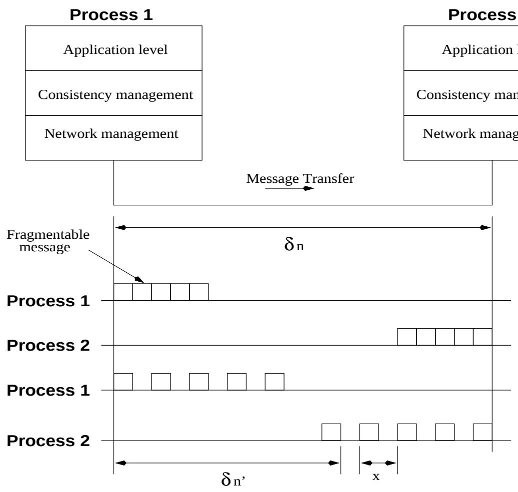

Figure 3: Relationship between message delay bound at the network level and delay bound δ'_n of the first fragment in a message.

The formulation of δ_n assumes that the worst-case overhead of processing at the network and consistency management levels at both endpoints is the same, which is probably a reasonable approximation. Figure 3 shows the relationship between δ_n and the delay bound of the first (and hence subsequent fragments) of a message. If x is the maximum inter-fragment time (assumed constant) and p is the number of fragments (or packets) in a message, then:

$$\delta'_n = \delta_n - x(p-1)$$

Each subsequent fragment in the same message has a delay bound of x time units after its predecessor. We can think of this delay bound for each packet as a *relative deadline* after which the packet is useless.

Translating throughput requirements: At the network level, throughput constraints are converted to bandwidth requirements. For a given link-bandwidth B, $T \leq B$. More precisely, for each of the p packets in a message, let the packet headers comprise a fraction h of the packet payload length. Let each packet have a payload of l bytes. If the worst-case message size is s bytes then the total number of bytes for all packets in a message is $l(1+h)\lceil\frac{s}{l}\rceil)$, where $p = \lceil\frac{s}{l}\rceil$ and the throughput requirements at the network level are at most $\frac{n_{max}(l(1+h)(\lceil\frac{s}{l}\rceil))}{t}$ bytes per second, for an upper limit of n_{max} updates to a shared objects over some interval of t seconds.

Translating fidelity requirements to packet-level loss constraints: As stated above, fidelity requirements specify the fraction f of shared state information that must be exchanged. The consistency manager can inform the network resource manager of the actual fraction of state information (as a proportion of the size of each shared object) inside each message. If the message contains all s bytes (i.e. 100%) of the shared state information and $f < 1.0$ then the packet scheduler in the network manager has the flexibility of discarding or scheduling $\lfloor(1-f)p\rfloor$ packets later than their delay constraints allow. If the message actually contains f' fraction of total state s, where $f' \geq f$ then the loss constraint on packets for this message allows $f' - f$ fraction of all p packets to be lost[2] or late. That is, $\lfloor(f'-f)p\rfloor$ packets can be lost. Note that we have made no assumptions about which packets can or cannot be lost. In reality we would not want to have consecutive packet losses but would prefer packet losses to be evenly dispersed throughout the message. More informative constraints would be required for this, such as the allowance of x packet-losses for every y packets to be sent.

4 QoS and Resource Management for Multiple Processes on the Same Host

We now consider the situation when multiple application processes on the same host each require that their own QoS constraints on shared objects must be met. We start by defining some key terms, relevant to the Dionisys system.

[2] We use the term 'loss' to mean any packet that fails to be delivered to the destination within its maximum delay bound, δ.

4.1 Paths, Modules and Interactors

As stated earlier, an *interactor* is an association between cooperating shared objects spanning a network and all relevant resources, both communication- and computation-bound, that support object exchanges. Interactors capture the QoS constraints and resource requirements at each stage in the generation, processing, and transfer of information between multiple cooperating objects. Interactors extend the notion of a 'path'[16, 17, 18] by capturing associations between multiple interacting objects and their resource requirements. In contrast, a path is a point-to-point logical connection, along which messages flow through a set of processing modules and resources.

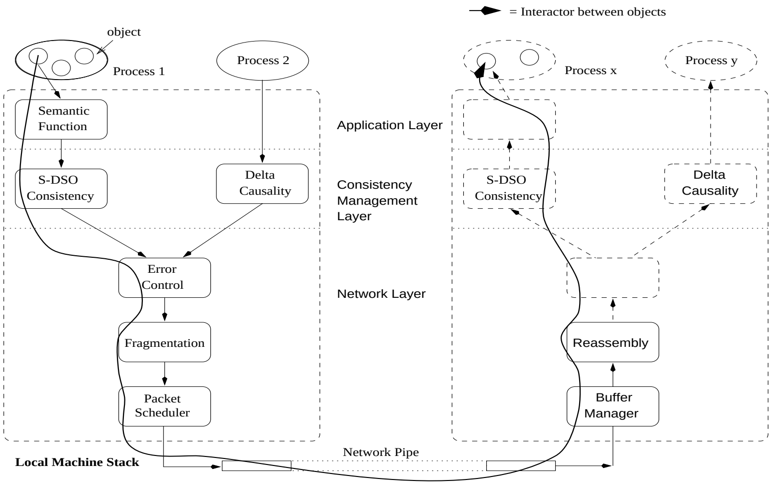

Figure 4: An example showing interactors between objects in different processes. Interactors are the end-to-end associations between cooperating objects and all resources along the path between those objects. Interactors also capture the QoS constraints at all stages of information flow between objects.

The modules of a path and, hence, an interactor must be scheduled on available processors to generate and process the messages that carry shared information. Note that two or more application processes executing on the same host might require some or all the same processing modules. Likewise, two or more processes executing on two different processors in a multiprocessor host might actually use the same code module at the same time. In this case, two copies of the code must execute simultaneously on each processor.

An example of interactors is given in Figure 4. Two processes access different consistency protocol modules in the consistency management layer but use the same error control, fragmentation and packet scheduling modules in the network layer of the sending host.

4.2 Dionisys: Runtime Support for End-to-End QoS

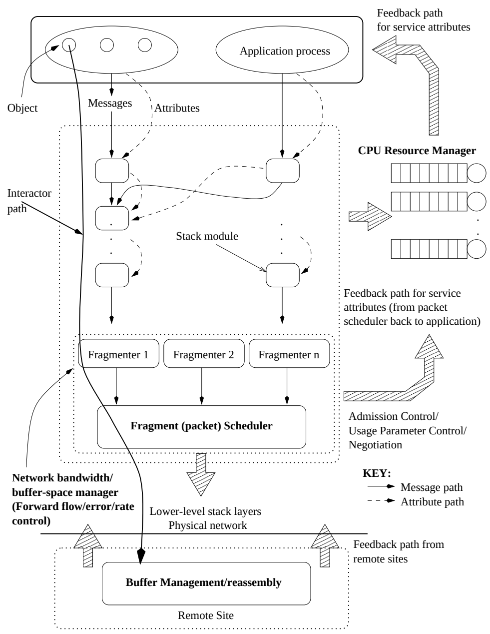

Figure 5: Dionisys: Runtime Support for End-to-End QoS

Dionisys (see Figure 5) is a runtime system to support end-to-end QoS in distributed applications. It can support multimedia applications, such as video-on-demand and also shared object applications, such as distributed virtual environments with real-time state-sharing constraints. Dionisys will meet the needs of such applications, comprising multiple processes executing on the same host.

As discussed earlier, the QoS constraints specified at the application/object level are passed down between layers in the system. These constraints take the form of attributes which follow a logical path along with the messages through the modules. At each module, the constraints are mapped to a form semantically meaningful to that module.

In Dionisys, the CPU resource manager is responsible for binding modules to threads (or processes) and scheduling threads on processors. The network manager is responsible for fragmenting and reassembling messages, packet scheduling, and policing/flow/error control. The feedback path passes quality of service attributes back to the application so that they may be dynamically adjusted to maximize available resources. Usage control (or policing) ensures that individual message or interactor-level service constraints are maintained while message-processing is taking place.

4.2.1 CPU Resource Management in Dionisys

CPU resource management specifically involves the allocation of CPU cycles to the servicing of protocol processing modules. When a process wishes to send a message across the network, via a given interactor, the CPU resource manager knows all the protocol modules associated with that interactor[3]. Hence, the CPU resource manager must schedule for execution a sequence of modules on the available processors. A sequence of modules is like a single task with multiple phases, such that one phase cannot start until the previous (dependent) phase has completed.

There are two specific operations the CPU resource manager must perform in controlling the usage of CPU resources:

- *Binding protocol processing modules to threads*: Ivan-Rosu and Schwan [19], and Hutchinson and Peterson[18] showed that by 'shepherding' modules together to run in a single thread, message latency is reduced, since a message that has been processed by one module can be immediately processed in the next module, executing within the same thread.

- *Scheduling threads on available CPUs*: At every level in the end-system, the service constraints on messages must be mapped to a meaningful form for the lower layers. In terms of modules, this means that we must also consider the computation time of each module when mapping attributes from one module to the next. That is, for delay constraints, the overhead of scheduling and executing modules in threads must be taken into consideration.

CPU resource management is complicated by the existence of multiple interactors, each requiring protocol-module processing[4], and each having different service constraints. The resource manager can exist on uni- and multiprocessor machines and must make the best use of available resources at all times.

A number of well-known scheduling policies can be used for thread scheduling. Mercer [20] has investigated the use of both earliest-deadline first and rate-monotonic scheduling in his work

[3] In Dionisys, interactors are specified by a *processing signature* at the object-level, which describes all the modules in that interactor. For state-based applications, we need only worry about specification of the correct consistency and real-time constraints and the management layers invoke the correct modules.

[4] The protocol modules can each have different computation times.

on processor capacity reserves, which focuses on CPU resource management to reserve processor cycles for end-to-end service guarantees.

4.2.2 Packet Scheduling and Network Resource Management in Dionisys

The network manager consists of the message fragmenter/reassembler, packet scheduler, and various modules for usage parameter control (UPC), and flow/error/rate control. To multiplex messages associated with different interactors efficiently, arbitrary-length application-level messages can be fragmented into packets of fixed length, by enabling the fragmenter module.

The header of each packet carries a sequence number identifying the position of the fragment in the message and a length-indicator, to indicate the length of the useful part of the fragment payload. The message header is also included in every packet. Replication allows meaningful interpretation of the packets at the destination even when some packets of a message may be lost.

The network resource manager supports user-configurable packet scheduling policies. Based upon the policy enforced, it may be necessary to have additional admission control and usage parameter control functionality, to ensure that service constraints on messages are maintained for the duration of a connection i.e., the lifetime of a path or interactor.

Service attributes may be specified for an interactor when the interactor is first established, or they may be dynamically set by a message transmitted across an interactor. Interactors can either be established before data transfer begins or at the same time as data transfer commences. The latter approach would be most applicable for short, infrequent message-exchanges. In the extreme case, a message traveling across an interactor may specify its own unique service parameters, allowing each individual message traveling along the same interactor to have its own QoS parameters. If admission control/negotiation is in effect, the network manager feeds back to the application layer the optimum level of service it can provide, given the desired service level the application requires and the maximum service the system can supply in the presence of other messages requiring service.

DWCS Packet Scheduling: As part of the Dionisys system, we have implemented a packet-scheduling algorithm, called Dynamic Window-Constrained Scheduling (DWCS). The algorithm accepts two attributes per packet. At any time, an attribute-pair applies to all packets in the same class. The attributes are:

- *Deadline* - this is the latest time a packet can *commence* service. Any packet that cannot be scheduled for transmission by its deadline can be dropped if the attribute-specification for the logical connection (i.e., interactor) allows some packets to be lost. If a packet cannot be lost it must be transmitted late.

- *Loss-tolerance* specified as a value x_i/y_i, where x_i is the number of packets that can be lost or transmitted late for every *window*, y_i, of consecutive packet arrivals in the same stream, i. For every y_i packet arrivals in stream i, a minimum of $y_i - x_i$ packets must be scheduled on time, while at most x_i packets can miss their deadlines and be either dropped or transmitted late.

DWCS orders packets for transmission based on the *current* values of their loss-tolerances and deadlines. Precedence is given to the packet at the head of the stream with the lowest loss-tolerance[5]. Packets in the same traffic class (for example, those packets forming part of a larger message) all have the same original and current loss-tolerances. Packets in the same message are scheduled in their message order. Whenever a packet misses its deadline, the loss-tolerance for all packets in the same message, m, is adjusted to reflect the increased importance of transmitting a packet from m. This approach avoids starving the service granted to a given packet stream, and attempts to increase the importance of servicing any packet in a message likely to violate its original loss constraints. Conversely, any packet serviced before its deadline causes the loss-tolerance of other packets (yet to be serviced) in the same traffic class to be increased, thereby reducing their priority.

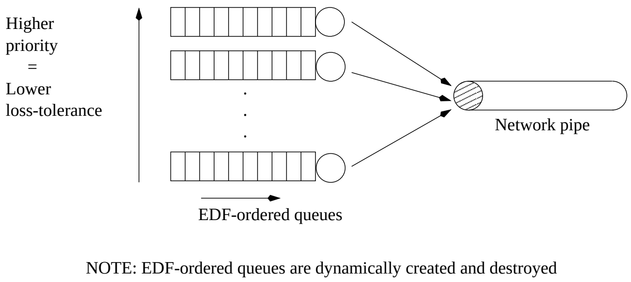

Figure 6: DWCS packet scheduling.

The DWCS algorithm actually works as follows: it places packets into logically-distinct priority queues, with packets in the same queue ordered earliest-deadline first (see Figure 6). Priorities are assigned to queues based on the current loss-tolerances of all buffered packets. Thus, all packets

[5]The exact rules for ordering packets for transmission will be described later.

in the same queue have the same current loss-tolerance. Packets are scheduled for transmission from the head of the highest-priority queue, where the highest priority queue is the one holding the lowest loss-tolerant packets[6]. The number of logically-active queues depends on the number of distinct packet loss-tolerances among all buffered packets. This means new queues may be dynamically created while existing queues are destroyed when they are empty. The loss-tolerance of a packet changes over time, depending on whether or not another (earlier) packet from the same message has been scheduled for transmission by its deadline. If a packet cannot be scheduled by its deadline, it is either transmitted late (with adjusted loss-tolerance) or it is dropped and the deadline of the next packet in the message is adjusted to compensate for the latest time it could be transmitted, assuming the dropped packet. A more detailed description of DWCS scheduling is out of the scope of this paper.

4.2.3 Trade-offs Between Bandwidth Utilization and Scheduling Overhead

The Dionisys system is implemented as a library on Solaris 2.5.1. Processes place information for packetization and scheduling in Dionisys in a shared memory region. This region is guarded by semaphores, so that queue data structures are updated atomically. If the packet scheduler, or higher level entities such as the fragmenter/packetizer run too frequently, they will force other threads to block while trying to acquire semaphores for the shared memory segment. Hence, not only does the scheduler steal cycles from other threads, it also forces executing threads to block and there are context-switch penalties for such behavior. Hence, it may be impractical and too expensive to run the scheduler in a work-conserving manner.

If the scheduler is invoked periodically, and at most one maximum size packet is serviced each invocation, we can bound the overhead of packet scheduling. The problem is, deciding on a suitable scheduling period for packets of different sizes, each demanding different transmission delays. Furthermore, execution of a periodic scheduler may result in unnecessary overhead when there is nothing to service.

In the next section, we show, empirically, the packet scheduling frequency to minimize CPU overheads while maximizing network bandwidth usage.

[6]There can be more than one logical priority queue with a loss-tolerance of zero, depending on the value of the loss-denominator for packets in the corresponding queue. However, such details are out the scope of this paper.

5 Experimental Evaluation

The experiments were performed on a pair of Sparc Ultra II, 140 MHz machines, connected via an ATM/OC3 link, with a 155 Mbps capacity. Figure 7(a) shows the bit service rate per interactor, as the number of simultaneously active interactors increases from 1 to 16, and the scheduler period (between executions) increases from 1 to 500 milliseconds. As the scheduler period is increased, the scheduler consumes fewer CPU cycles per second, at the cost of potentially lower bit service rates into the network for each interactor. (Note that the minimum overhead per invocation of the packet scheduler, when the input queue is empty, is $26(\mu s)$). In these experiments, the nominal data rate input at the application/object-level of each interactor is set to 1Mbps. Each interactor conveys video (MPEG-1) data between objects. The sample MPEG-1 data is encoded at a rate of 24 frames per second, but each message is at the granularity of a group of pictures (that is, 12 frames) and averaged 62.5 KBytes. Fragmentation is disabled, so each message is encapsulated in a separate packet.

From Figure 7(a), the bit service rate peaks when the scheduler executes once every 10 milliseconds. (Note that the scheduler services at most one packet per invocation, depending on whether the input queue is empty or not). At lower frequencies, the scheduler overhead is less but the interactor bit rate drops. When only one interactor is active, the scheduler frequency has little effect on the bit service rate. However, it becomes increasingly necessary to increase the scheduler frequency as the number of active interactors increases. Observe that below 10 millisecond periods between scheduler executions, the bit rates of each interactor actually drop a small amount. This is probably due to increased lock-accesses to the shared memory data structures in Dionisys by each interactor and the packet-level scheduler threads.

Figure 7(b) shows the bit service rate for a single active interactor, when the packet granularity is at the level of one MPEG-1 frame. That is, the average packet size is $\frac{1}{12}$ the size of packets in the first experiments. To maintain the same arrival rate of 1 Mbps for the interactor, packet arrivals for scheduling occur once every $\frac{1}{24}s$. This has the effect of increasing the lock-access frequency in Dionisys 12-fold from the previous experiments, thereby limiting the peak bit service rate. Observe that even when the scheduler executes once every 10 milliseconds, the bit service rate of the interactor is less than 800 Kbps. In the previous experiments, the courser-grained packets meant that the packet arrival rate could be reduced, thereby reducing lock-accesses while maintaining a 1 Mbps bit arrival rate per interactor. Figure 7(a) shows the bit service rate for one active interactor when the packet scheduler period is 10 milliseconds is equal to the full 1 Mbps arrival rate. It appears that if too many lock-accesses are made, then the scheduler can be blocked by application threads holding locks and executing on behalf of each interactor, which causes a form of priority inversion in Dionisys[7].

From these experiments, there is clearly a need for packet-level scheduling to occur at a rate proportional to the aggregate arrival rate of packetized information across multiple interactors. That is, a system like Dionisys must adjust the packet scheduling rate to compensate for CPU overheads when it is necessary to meet specific bit service rates across the network. In designing a runtime system like Dionisys, we must be careful of limiting factors such as lock-access frequency. Hence, we cannot simply increase the scheduler frequency, even when we have spare CPU cycles, to always guarantee an increase in bit service rates for interactors.

An adaptive system which adjusts the packet scheduling rate, μ, to match the aggregate arrival rate, $\sum_{i=1}^{n} \lambda_i$, where n is the number of simultaneous interactors and λ_i is the arrival rate for interactor i, will maximize bit service rate and, hence, bandwidth resources, without wasting CPU cycles. This assumes that blocking delays due to lock acquisition are either negligible or infrequent.

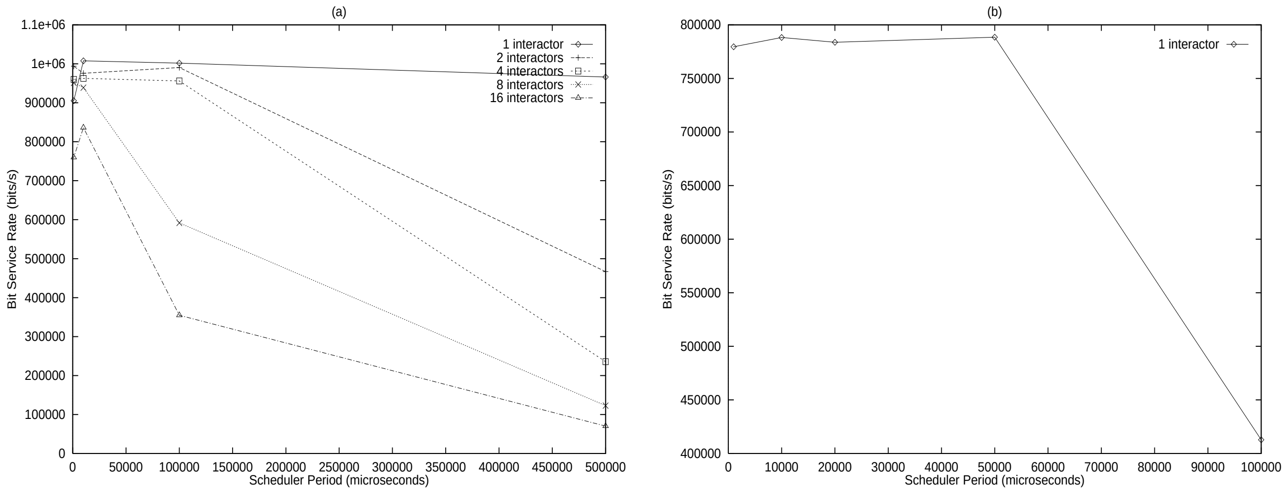

Figure 7: Bit service rate versus scheduler frequency: (a) Effects of servicing multiple simultaneous interactors with course-grained packet sizes; (b) Bit service rate for one interactor on finer-grained packets.

[7]The Dionisys packet scheduler runs as a real-time (RT) class kernel-thread in Solaris.

6 Conclusions and Future Work

This paper describes the Dionisys end-system quality of service (QoS) approach to specifying, translating, and enforcing end-to-end, object-level QoS constraints. Dionisys differs from previous work on QoS architectures by supporting QoS constraints on distributed shared objects, as well as multimedia streams. The key concept in Dionisys is the notion of an 'interactor', which captures the QoS constraints and resource requirements at each stage in the generation, processing, and transfer of information between multiple cooperating objects. Using interactors, Dionisys is able to coordinate both thread and packet-level scheduling, so that information is processed and transmitted at matching rates. However, there are tradeoffs in the use of CPU cycles for scheduling and the need to meet QoS constraints on the information transferred between interacting objects. We show, empirically, the packet scheduling frequency to minimize CPU overheads while maximizing bandwidth usage for simple interactions between objects exchanging video information.

We are currently evaluating the Dynamic Window Constrained Scheduling (DWCS) policy for packet-level scheduling. We intend to show that DWCS has the ability to meet weighted-fair bandwidth allocation in the same way that fair-queueing algorithms can, except that DWCS can coordinate the allocation of bandwidth amongst various interactors without prior knowledge of the maximum number of interactors that may coexist. By contrast, fair queueing algorithms need a-priori knowledge of all traffic classes that will communicate over a link, so that weights can be allocated to each class in a hierarchical partition tree. DWCS uses constraint information on a per-interactor basis, which is semantically-independent of the constraints on other interactors. Fair-queueing algorithms use weights which are meaningless without relation to other traffic-class weights.

Dionisys is still in its early stages of development and we hope to evaluate the completed system on the complex distributed applications we mentioned earlier, such as highly-dynamic, richly-detailed virtual environments, and tele-medicine applications.

References

[1] 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*, pp. 207–217, ACM, 1994.

[2] C. Carlsson and O. Hagsand, "Dive-a platform for multi-user virtual environments," *Computers and Graphics*, vol. 17, pp. 663–669, November-December 1993.

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

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

[5] 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.

[6] R. Yavatkar, "MCP: A protocol for coordination and temporal synchronization in multimedia collaborative applications," in *Proc. 12th IEEE ICDCS*, pp. 606–613, IEEE, 1992.

[7] D. Ferrari, A. Banerjea, and H. Zhang, "Network support for multimedia - a discussion of the tenet approach," *Computer Networks and ISDN Systems*, vol. 26, pp. 1267–1280, 1993.

[8] A. Campbell, "A quality of service architecture," in *ACM SIGCOMM Computer Communication Review*, ACM, April 1994.

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

[10] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A high-performance endsystem architecture for real-time corba," *IEEE Communications Magazine*, vol. 14, February 1997.

[11] A. Gheith and K. Schwan, "Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications," *ACM Transactions on Computer Systems*, vol. 11, pp. 33–72, April 1993.

[12] D. I. Rosu, K. Schwan, S. Yalamanchili, and R. Jha, "On adaptive resource allocation for complex real-time applications," in *18th IEEE Real-Time Systems Symposium, San Francisco, CA*, IEEE, Dec. 1997.

[13] P. Gopinath, T. Bihari, and K. Schwan, "Object-oriented design of real-time software," in *10th International Real-time Systems Symposium, Los Angeles*, pp. 194–201, IEEE, Dec. 1989.

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

[15] D. Ferrari, "Client requirements for real-time communication services," *IEEE Communications Magazine*, vol. 28, pp. 76–90, November 1990.

[16] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman, "Scout: A communications-oriented operating system," Tech. Rep. TR-94-20, The University of Arizona, Department of Computer Science, June 1994.

[17] D. Clark, "The structuring of systems using upcalls," in *Proceeedings of Tenth ACM Symposium on Operating Systems Principles*, pp. 171–180, ACM, December 1995.

[18] N. Hutchinson and L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[19] D. Ivan-Rosu and K. Schwan, "Improving protocol performance by dynamic control of communication resources," Tech. Rep. GIT-CC-96-04, Georgia Institute of Technology, College of Computing, February 1996.

[20] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reservation for multimedia operating systems," in *IEEE International Conference on Multimedia Computing and Systems*, IEEE, May 1994.