QUIC: A Quality of Service Network Interface Layer for Communication in NOWs *

R. West, R. Krishnamurthy, W. K. Norton, K. Schwan, S. Yalamanchili, M. Rosu[†] and V. Sarat

Critical Systems Laboratory

Georgia Institute of Technology
Atlanta, GA 30332

Abstract

This project explores the development of a hardware/software infrastructure to enable the provision of quality of service (QoS) guarantees in high performance networks used to configure clusters of workstations/PCs. These networks of workstations (NOWs) have emerged as a viable high performance computational vehicle and are also being called upon to support access to multimedia datasets. Example applications include Web servers, video-on-demand servers, immersive environments, virtual meetings, multi-player 3-D games, interactive simulations, and collaborative design environments. Such applications must often share the interconnect with traditional compute intensive parallel/distributed applications that are usually driven by latency requirements in contrast to jitter, loss rate, or throughput requirements. The challenge is to develop a communication infrastructure that effectively manages the network resources to enable the diverse QoS requirements to be met. The major components of QUIC include (1) use of powerful, processors embedded in the network interfaces, (2) scheduling paradigms for concurrently satisfying distinct QoS requirements over multiple streams, (3) re-configurable hardware support to enable complex scheduling decisions to be made in the desired time frames, and (4) a flexible and extensible virtual communication machine that provides a uniform interface for dynamically adding hardware/software functionality to the network interfaces (NIs). This papers reviews the goals, approach and current status of this project.

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

1 Introduction

The continuing rapid decrease in the cost of both processor and network components has led to a tighter integration of computation and communication. The result has been an explosion of network-based applications characterized by the processing and delivery of continuous data streams and dynamic media [1, 6] in addition to servicing static data. Numerous examples can be drawn from web-based applications, interactive simulations, gaming, visualization, and collaborative design environments. The changing nature of the workload and cost/performance tradeoffs has prompted the development of a new generation of scalable media servers structured around networks of workstations interconnected by high speed system area network (SAN) fabrics. However the ability to construct scalable clusters that can serve both static and dynamic media is predicated on successfully addressing two major issues.

The first is that node architectures are based on a CPU-centric model optimized for uniprocessor or small-scale multiprocessor applications. This can lead to significant inefficiencies for distributed applications. Specifically, while CPU and wire bandwidths have been increasing rapidly over the years, memory and intra-node I/O bandwidths continue to improve at much slower rates, resulting in a performance gap that will continue to widen in the foreseeable future. This implies that interactions between the network and hosts utilizing main memory are expensive. Additional costs arise for such interactions from overheads due to I/O bus usage [4, 5], communication protocol implementations (e.g., if interrupts are used [2]), and interactions with the host CPU's memory management and caching infrastructure [3]. Consequently, network-based applications that produce, transport, and process large data sets suffer substantial losses in performance when these data sets must be moved through the memory and I/O hierarchies of multiple nodes.

The second issue is the workload and performance characteristics of this new generation of network-based applications. Data types, processing requirements and performance metrics have changed placing new functional demands on the systems that serve them. Several key attributes are as follows [1, 6].

1. **Real-Time Response:** Interactivity and requirements on predictability, such as when to service video streams, make real-time response important. Such timing constraints cannot be met unless the network resources can be scheduled and allocated effectively.

2. **Shift from Quantitative to Qualitative Metrics:** With the new applications there has also been a shift in metrics that define their performance [1, 7]. Traditional quantitative metrics such as latency and bandwidth give way to more qualitative metrics such as jitter and real-time response. The smooth update of a video stream or the response time to a user access request is more important than minimizing transmission latency. The metrics clearly affects the choice of implementation techniques. For example packets may be dropped in a video or audio stream without compromising quality whereas this would be unacceptable for most transaction processing applications.

3. **Hardware Limitations:** The service and transfer of video streams and images places increasing demands on the memory and I/O bandwidths at a time when the bandwidth gap between the CPU and memory and I/O subsystems is growing. Furthermore the available physical bandwidth to the desktop and to the home will grow by several orders of magnitude. Wire bandwidths into the cluster that serves these machines with media will follow or lead this trend. This will exacerbate the "bandwidth gap" between the CPU and the wire, thereby making qualitative metrics more sensitive to the same.

4. **Heterogeneity:** Systems such as real-time media servers need to service hundreds and, possibly thousands, of clients typically each with their own *distinct* quality of service (QoS) requirements, such as packet dropping vs. reliable message transmission, low latency vs. jitter, or throughput vs. latency. We must concurrently meet diverse service requirements with the same set of hardware and software resources.

The QUIC project studies the issues inherent in Quality of Service management for cluster machines. Our project focuses on the functionality of the network messaging layer in providing QoS guarantees. Our approach is based on the development of an extensible QoS management infrastructure for which we carefully select components that are to be implemented within programmable network interfaces (NIs). In Section 2 we provide a brief overview of the project while a description of the overall Quality Management infrastructure can be found in Section 3. The rest of the paper describes the approach taken in the implementation of each of key QUIC components.

2 Project Overview and Goals

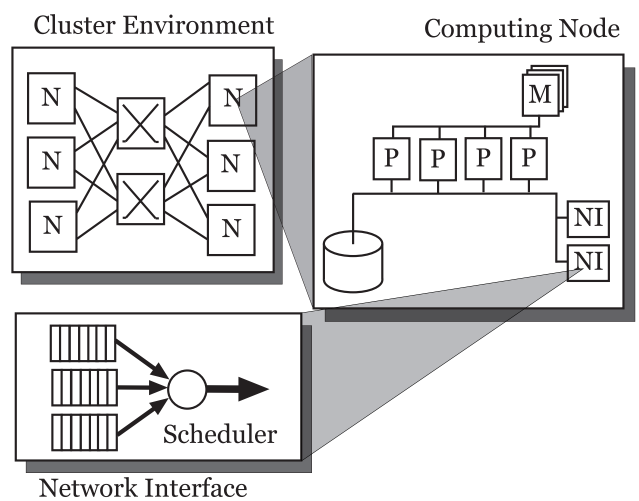

Figure 1: An Overview of the QUIC Development Infrastructure.

This project has a strong experimental component and therefore our infrastructure is biased towards rapid prototyping and evaluation. An overview of the QUIC infrastructure is illustrated in Figure 1. Our development environment is a cluster of 16 quad-Pentium Pro nodes interconnected via Intel's i960 based Intelligent I/O (I2O) network interface cards running the VxWorks operating system and interconnected via 100 Mbits/sec Ethernet. Concurrent development proceeds under the Windows NT and Solaris node operating system environments. Individual nodes have multiple network interfaces, multiple CPUs and eventually will coordinate services from multiple nodes[13].

Our goal is the development of an effective quality management infrastructure that can service a large number of connections, each with distinct service requirements, while minimizing host-NI interactions for NIs whose functionality can be easily and dynamically extended. Our approach also investigates the extension of NI functionality to include computations on data streams as they pass through the network interface. By performing such stream computations 'on the fly' as data is passed through the interface we can avoid costly traversals of the memory hierarchy and thereby obtain finer control over service quality, for example real-time response. We are investigating supporting such stream computations through the use of programmable hardware in the form of dynamically configurable field programmable gate arrays (FPGAs) within the NIs. By placing quality management functionality "close" to the wire and I/O components attached to the I2O boards, we expect to enable QoS guarantees at higher levels of resource utilization than commodity clusters will otherwise permit. The following sections describe the individual aspects of the QUIC project.

3 QUIC Quality Management Infrastructure

The QUIC QoS infrastructure has several components that jointly permit the implementation of a variety of quality management functions and policies applied to the information streams into and out of CPUs.

- At the core of QUIC resides the extensible NI software architecture, which is designed for runtime extension with functions that manage the information streams used by particular applications. Two types of functions are supported: (1) *streamlets* that operate on the contents of data being streamed out of or into hosts, via the NIs on which the QUIC infrastructure's core components reside, and (2) *scheduling* or quality management functions that manage such streams, typically by applying certain scheduling algorithms to streams as they pass through NIs and the NI-host interface. This paper's principal focus is on these scheduling functions and on the manner in which they are applied to streams.

- QUIC offers two types of interfaces to applications: (1) a communication interface that directly supports its information streams, using standard communication protocols enhanced with the ability to specify scheduling functions or streamlets applied to them, and (2) an extension interface using which applications can define new quality management (scheduling) functions or streamlets to be applied to their information streams.

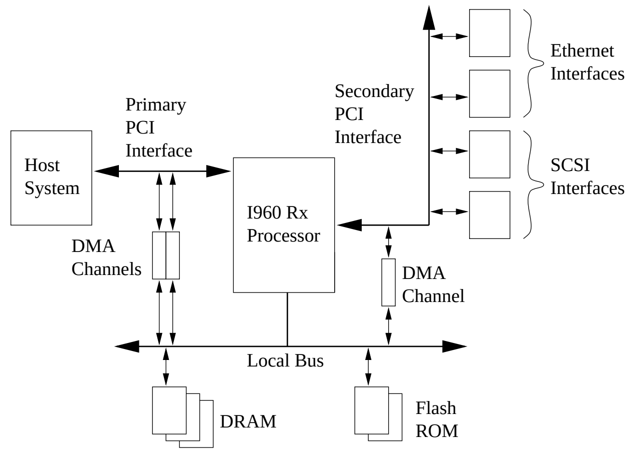

Figure 2: Architecture of the I2O Network Interface.

- QUIC also offers functions within each NI that permit the coordination among multiple NIs that jointly operate on certain information streams or cooperate to support applications. This 'control layer' of QUIC is also described below.

These functional components of QUIC are reviewed in the following sections.

3.1 Network Interface Architecture

Hardware Platform Intel's IQ80960RD66 Evaluation Platform Board serves as the underlying hardware upon which the QUIC NI software architecture is being instantiated. The board is designed to be a testbed for systems that offload I/O processing from the host. The architecture of the network interface is shown in Figure 2. The board resides in a PCI slot on the host machine and provides two network ports and two SCSI ports on an isolated PCI bus. The cards single processing unit is an Intel i960RX processor running at 66Mhz providing sufficient compute power to experiment with the movement of non-trivial computations to the interface[14, 15].

The quality management infrastructure executing within this NI will be layered on VxWorks, a real-time operating system from Wind River Systems. Our motivation for using VxWorks is its provision of cross compilation tools as well as a runtime layer using which our ideas concerning suitable NI functionality for quality management may be prototyped rapidly.

In general, the criteria for choice of an embedded kernel can be delineated very simply as follows: small footprint, lightweight, configurability, support for devices on the I/O controller card, and cross platform development and debugging facilities. When choosing an NI for this project, the first option we considered was to construct a custom kernel for the i960. Our desire to rapidly prototype a functional system led to the choice of a commercial kernel for the i960 RD card. Two viable offerings from Wind River Systems are the IxWorks system developed as part of the I2O industry initiative and the VxWorks kernel. The ROM-resident IxWorks system provides a flexible environment for I2O device driver development, for monitoring driver performance, and it offers sophisticated message passing and event queues between multiple IOP boards. However, it assumes that message transport is performed using the I2O standard. We decided not to pursue this approach due to our focus on high performance (including low latency) communications. Toward this end, we wish to experiment with alternative I2O - host communication interfaces. Such experimentation is enabled by the second commercial offering from Wind River Systems for the i960 RD environment: the VxWorks development environment. The VxWorks operating system kernel is based on the same microkernel as the IxWorks I2O system. In contrast to IxWorks, VxWorks is highly configurable, as it can be scaled from a small footprint ROMable version to a larger footprint, full-featured Operating System. VxWorks makes no specific assumptions about the I2O - host interface[20, 19].

QUIC Communication Paradigm QUIC utilizes a general model of processing stream data closer to the network interface wherein the nature of the processing may include simple data computations as well as scheduling computations. Towards this end, QUIC enables the dynamic placement of computations within the NI. A request for the establishment of a connection will identify the required computations and specify desired levels of service quality based on which additional functionality such as admission control, policing (for network bandwidth) and scheduling may be performed. At this point, we simply point out that the establishment of a connection concerns not just the allocation of communication bandwidth to enable real-time transmission, but also the identification of the computations relevant to the data stream and the allocation of appropriate computational resources so that the data may be operated on in real-time. The two classes of computations considered in our work are (1) computations that operate on the actual stream data and (2) those that concern the runtime control (ie., scheduling) for streams. In either case, concerning communication coprocessors, this implies the runtime extension of these coprocessors with functionality suited for specific streams. An architecture and suitable interfaces for this purpose is being developed and is described as follows.

Virtual Communication Machine To application programs, the NI is abstracted as *virtual communication machine* (VCM), where its visible interface to applications is one that (1) defines the computations (instructions) it is able to perform on the applications' behalf and that (2) offers functions for the machine's runtime extension to add or subtract instructions as well as reporting necessary elements of its internal state.

Internally, the VCM's software architecture aims to provide adequate support for using the NI processor to improve the performance of streams used by network applications. Towards this end, the VCM provides an efficient environment for executing application-specific computational modules that can benefit from running 'closer to the network'. We use the term *application specific extension modules* (ASEM) to refer to such application-specific code that is 'directly' using the NI resources. In our testbed, the VCM will execute on the i960-based I2O coprocessor to which multiple disks and network links are attached. ASEMs provide a common abstraction and can be dynamically placed in the NI to process streams, perform scheduling, or manage NI resources such as disks.

Host applications communicate with the VCM through shared (host or NI) memory. The union of all these memory regions is called the VCM address space. Applications issue execution requests to the NI as VCM *tasks*. Tasks can be created and destroyed dynamically. The first VCM task of an application is created automatically when the application connects to the NI. The most common example of a task is one that executes a VCM program which in turn is comprised of core and 'added' *instructions*.

VCM programs are built as sequences of core and/or 'added' VCM instructions. Core instructions implement VCM functionality always resident on the VCM, including the functionality shared by all extensions and that required to configure the VCM's operation. Added instructions implement the extensions used by certain application programs. In other words, added instructions are functions specific to certain applications.

The VCM instruction dispatcher is implemented as an interpreter running on the NI processor. The dispatcher reads each VCM instruction, checks the availability of its parameters and activates the appropriate code. The NI processor transfers back results to the application by writing into the memory regions shared between the NI and the application.

The overhead of the interaction between applications and VCM instructions is low because of the shared-memory-based implementation. To further lower this overhead, access patterns to the shared regions are used to determine their placement in the host or the NI memory. Using shared memory also helps in decoupling the executions on the host and NI. Towards this end, applications and interface layer can build in the VCM address space arbitrary long chains of 'tasks-to-be-executed' and of 'tasks-completed' descriptors, respectively. In addition to the shared-memory-based interaction, the interface layer can also signal to application processes using signaling primitives provided in the host operating system. We note that certain elements in the proposed architecture were influenced by our experience with a small prototype built around an OC-3 ATM card (FORE SBA-200E). This commercial NI features a 25 MHz i960CA processors and 256K SRAM. Some details of our design appear next.

Extension Functionality and Interface We now comment on the 'extension' instructions part of the core instruction set. The conceptual basis for this work are (1) our previous experiences with the implementation of a VCM for FORE ATM interface boards[4] and (2) event-based mechanisms developed by our group for uniprocessor and distributed systems. The basic idea of these mechanisms is to permit applications to define events of certain types, to associate (at runtime) handlers for these events, and to create event channels to which event producers and consumers can subscribe. In our system, handlers are executed anytime an event is produced or consumed, at the producing and at the consuming side of the event channel. For online VCM extension, then, the application may produce an extension event and provide a new handler for this event type. The handler code is installed at runtime on the VCM, resulting in the creation of a new VCM instruction ready for use by the application program. Interactions of the new VCM instruction with lower level VCM facilities are resolved at installation time, as well.

The advantage of using this event-based approach is our ability to have any number of VCM's listen for extension events from any number of application programs, thereby offering a scalable approach to system extension even for large-scale machines. Some implementation details on VCM extension follow.

Each extension module is assigned one or more VCM instruction opcodes and control message IDs. An extension module is a collection of the following types of handlers:

- VCM instruction handlers invoked by the VCM instruction dispatcher upon encountering an instruction assigned to the module,
- control message handlers, invoked upon the delivery of a control message with an ID assigned to the module, and
- time-out handlers.

These handlers share the state of the extension modules, which is stored in the NI memory. We implemented five extension modules in the ATM prototype VCM. They all share the structure outlined above and they consist from 120 (the simplest) to 1170 (the most complex) lines of C code.

3.2 QUIC Runtime Environment

The QUIC runtime environment (RTE) is implemented as the VCM's runtime layer. This layer has two components, the first of which provides a set of technology-independent low-level communication abstractions. These abstractions make writing extension modules easier, as the application programmer does not have to be aware of the hardware details of a particular NI card. In addition, these abstractions are independent of the underlying networking technology (Ethernet, ATM, or Myrinet), thereby making the extension modules portable at the source code level. On top of this layer, the second component of the RTE is comprised of communication abstractions that support a collection of core services needed by most extension modules. By providing these services as part of the RTE, most of the functionality overlap between extension modules is eliminated. The implementation of this RTE component takes advantage of all the hardware support available on the NI to provide the best performance and, consequently, is highly dependent on the NI hardware.

Principles Guiding RTE Design The NI-dependent implementation of the first RTE component depends on the speed of the NI processor, the capacity of the NI memory, and the overheads of driving the network interconnect at full speed. A single-threaded library implementation of the RTE is the right choice for NIs with limited resources. An implementation based on a kernel for an embedded operating system should only be used for NIs with sufficient resources: fast processors, large memories, and interconnect-specific hardware that support the NI processor in driving the attached interconnects. Between these two alternatives sits the RTE implementation as a multithreaded

RTE Components The RTE is comprised of several components. The key issue in including components in the RTE is that they are used by several different extension modules. Our current design includes the following components.

Control Messaging System: This component implements reliable and in-order delivery of short messages between the NI processors. These short messages are called control messages and are used by extension modules on different NIs to exchange information. We found both reliability and in-order delivery very convenient when writing extension modules. Reliability needs to be implemented in the NIs as most SANs do not guarantee 100% message delivery. As message loss rates and latencies are relatively low on SANs, implementing in-order delivery should not increase the overhead of the NI processor considerably. Ideally, the latency of control messages should be only slightly higher than the hardware-imposed limit. To achieve this performance goal, buffers for control messages are preallocated in the NI memory. In our current prototype, reliable delivery is based on a sliding window protocol. Block acknowledgments are used between interface processors to acknowledge the receipt of multiple messages although upon request, certain control messages can be acknowledged immediately. This latter feature is useful in building fault-tolerant applications. For instance, we implemented this feature in our prototype and used it in an remote write extension module. This extension module is designed to improve the performance of applications that achieve fault-tolerance by maintaining a copy of their state in the memory of a remote host.

Time-out Components: A second RTE component implements several of time-out primitives, with different granularities and precision. The extended set of time-out primitives is necessary because we expect more precision and/or finer granularity to imply higher NI processor overhead.

Message Management Components: Routines for efficient assembling/disassembling of large messages from/into arbitrary collections of memory segments, placed either in the host or NI memory, are another RTE service. Our ATM-based prototype includes routines implementing zero-copy messaging: outgoing or incoming data is moved between the network registers and final destination without any intermediate copies in the host or NI memory. Our remote memory access and bulk messaging extension modules use these routines.

Memory Management Component: The RTE includes a dynamic memory management system. Our prototype includes a heap module which is implemented as buddy system to achieve better predictability.

4 QUIC Quality of Service Management

The QUIC scheduler represents an approach that emphasizes dynamic adaptation of scheduler parameters. We are pursuing an approach wherein existing scheduling algorithms can be modeled while permitting the algorithms to be adapted over time in an application-specific manner to respond to varying QoS needs.

4.1 QoS Management Paradigm

The QUIC project is exploring a flexible scheduling paradigm to concurrently satisfy diverse QoS requirements across multiple data streams. Packet priorities are dynamically updated at run-time to enable QoS requirements to be met. Essentially the packet priority is scaled as a function of the QoS that a packet has experienced up to that point in time *relative* to the QoS requested by the packet. Such an update operation has been referred to as priority *biasing* [7, 9, 8] since the priority value is biased by the relative degradation of its service. The biasing operation couples the effect of the scheduler (e.g., queuing delay) with the QoS demand (e.g., jitter bound). This distinguishes this approach from priority update mechanisms such as age counters that do not distinguish between QoS requested by distinct connections.

For example, consider only constant bit rate (CBR) connections where QoS is measured by the bandwidth allocated to a connection. The priority of a packet can be computed as the ratio of the queuing delay to the connection's inter-arrival time. Increasing queuing delay increases its priority in successive scheduling cycles. However, the rate at which the priority increases depends on the bandwidth of the connection. Such a priority biasing mechanism couples the ongoing effect of the switch scheduler (queuing delay) with a measure of the demands made by the application (connection bandwidth). As the negative impact of the switch scheduler grows so does the priority, effectively "biasing it" with time. Thus different connections are biased at different rates, i.e., higher speed connections are biased faster.

QUIC explores a hardware/software implementation of a generalized priority biasing framework. Our hypothesis is that by customizing biasing calculations by stream and data type and providing the ability to dynamically control biasing calculations we can achieve more effective utilization of network resources and thereby satisfy the QoS requirements of a larger number of communication requests. Two major issues the framework must address are: *when* is the priority of a scheme biased and *how* is it biased. QUIC currently implements a dynamic window constrained scheduling algorithm (DWCS) that provides two parameters for controlling the "when" and "how" components of generalized priority biasing. This paradigm is quite powerful in that it has been shown to be able to model a range of existing scheduling algorithms [17] while it also provides for dynamic control of scheduling parameters thus providing new avenues for optimizing the performance of heterogeneous communication streams. The remainder of this section describes the current instantiation of DWCS hosted within the network interfaces.

4.2 QUIC Quality Management - The DWCS Approach

The first parameter utilized by DWCS controls the *interval* of time between priority adjustments of its second parameter. The second parameter is simply the *biasing value*, used to decide which stream has the highest priority and, hence, which stream should be scheduled for transmission. Simply, the biasing value is dynamically adjusted at specific intervals of time. Furthermore, DWCS is flexible, that the biasing value could be adjusted based on the needs of individual streams. Our current implementation utilizes packet *deadlines* and *loss-tolerance* as the two scheduling parameters. The motivation for this choice and detailed description are provided in the following.

Applications, such as real-time media servers need to service hundreds and, possibly, thousands of clients, each with their own quality of service (QoS) requirements. Many such clients can tolerate the loss of a certain fraction of the information requested from the server, resulting in little or no noticeable degradation in the client's perceived quality of service when the information is received and processed. Consequently, loss-rate is an important performance measure for the service quality to many clients of real-time media servers. We define the term *loss-rate*[16, 12] as the fraction of packets in a stream either discarded or serviced later than their delay constraints allow. However, from a client's point of view, loss-rate could be the fraction of packets either received late or not received at all.

One of the problems with using loss-rate as a performance metric is that it does not describe when losses are allowed to occur. For most loss-tolerant applications, there is usually a restriction on the number of *consecutive* packet losses that are acceptable. For example, losing a series of consecutive packets from an audio stream might result in the loss of a complete section of audio, rather than merely a reduction in the signal-to-noise ratio. A suitable performance measure in this case is a *windowed loss-rate*, i.e. loss-rate constrained over a finite range, or *window*, of consecutive packets. More precisely, an application might tolerate x packet losses for every y arrivals at the various service points across a network. Any service discipline attempting to meet these requirements must ensure that the number of violations to the loss-tolerance specification is minimized (if not zero) across the whole stream.

Some clients cannot tolerate any loss of information received from a server, but such clients often require delay bounds on the information. Consequently, these type of clients require deadlines which specify the maximum amount of time packets of information from the server can be delayed until they become invalid. Furthermore, some multimedia applications often require jitter, or delay variation, to be minimized. Such a requirement can be satisfied by restricting the service for an application to commence no earlier than a specified earliest time and no later than the deadline time.

To guarantee such diverse QoS requires fast and efficient scheduling support at the server. This section describes the features specific to a real-time packet scheduler resident on a server (specifically designed to run on either the host processor or the network interface card), designed to meet service constraints on information transferred across a network to many clients. Specifically, we describe Dynamic Window-Constrained Scheduling (DWCS), which is designed to meet the delay and loss constraints on packets from multiple streams with different performance objectives. In fact, DWCS is designed to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant and/or delay-constrained, heterogeneous traffic streams.

4.3 The DWCS Scheduler

DWCS is designed to maximize network bandwidth usage in the presence of multiple packets each with their own service constraints. The algorithm requires two attributes per packet stream, as follows:

- *Deadline* – this is the latest time a packet can *commence* service. The deadline is determined from a specification of the maximum allowable time between servicing consecutive packets in the same stream (ie., the maximum inter-packet gap).
- *Loss-tolerance* – this is 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, depending on whether or not the attribute-based QoS for the stream allows some packets to be lost.

At any time, all packets in the same stream have the same loss-tolerance, while each successive packet in a stream has a deadline that is offset by a fixed amount from its predecessor. Using these attributes, DWCS: (1) can limit the number of late packets over finite numbers of consecutive packets in loss-tolerant or delay-constrained, heterogeneous traffic streams, (2) does not require a-priori knowledge of the worst-case loading from multiple streams to establish the necessary bandwidth allocations to meet per-stream delay and loss-constraints, (3) can safely drop late packets in lossy streams without unnecessarily transmitting them, thereby avoiding unnecessary bandwidth consumption, and (4) can exhibit both fairness and unfairness properties when necessary. In fact, DWCS can perform fair-bandwidth allocation, static priority (SP) and earliest-deadline first (EDF) scheduling.

4.4 DWCS Algorithm

Dynamic Window-Constrained Scheduling (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. Packets in the same stream all have the same original and current loss-tolerances, and are scheduled in their order of arrival. Whenever a packet misses its deadline, the loss-tolerance for all packets in the same stream, s, is adjusted to reflect the increased importance of transmitting a packet from s. This approach avoids starving the service granted to a given packet stream, and attempts to increase the importance of servicing any packet in a stream 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 stream to be increased, thereby reducing their priority.

The loss-tolerance of a packet (and, hence, the corresponding stream) changes over time, depending on whether or not another (earlier) packet from the same stream 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 stream is adjusted to compensate for the latest time it could be transmitted, assuming the dropped packet *was* transmitted as late as possible.

Pairwise Packet Ordering
Lowest loss-tolerance first
Same non-zero loss-tolerance, order EDF
Same non-zero loss-tolerance & deadlines, order lowest loss-numerator first
Zero loss-tolerance & denominators, order EDF
Zero loss-tolerance, order highest loss-denominator first
All other cases: first-come-first-serve

Table 1: Precedence amongst pairs of packets

Table 1 shows the rules for ordering pairs of packets in different streams. Recall that all packets in the same stream are queued in their order of arrival. If two packets have the same non-zero loss-tolerance, they are ordered earliest-deadline first (EDF) in the same queue. If two packets have the same non-zero loss-tolerance and deadline they are ordered lowest loss-numerator x_i first, where x_i/y_i is the current loss-tolerance for all packets in stream i. By ordering on the lowest loss-numerator, precedence is given to the packet in the stream with *tighter* loss constraints, since fewer consecutive packet losses can be tolerated. If two packets have zero loss-tolerance and their loss-denominators are both zero, they are ordered EDF, otherwise they are ordered highest loss-denominator first. If it is paramount that a stream never loses more packets than its loss-tolerance permits, then admission control must be used, to avoid accepting connections whose QoS constraints cannot be met due to existing connections' service constraints.

Every time a packet in stream i is transmitted, the loss-tolerance of i is adjusted. Likewise, other streams' loss-tolerances are adjusted *only if* any of the packets in those streams miss their deadlines as a result of queueing delay. Consequently, DWCS requires worst-case $O(n)$ time to select the next packet for service from those packets at the head of n distinct streams. However, the average case performance can be far better, because not all streams always need to have their loss-tolerances adjusted after a packet transmission.

Loss-Tolerance Adjustment. Loss-tolerances are adjusted by considering x_i/y_i, which is the original loss-tolerance for all packets in stream i, and x'_i/y'_i, which is the current loss-tolerance for all queued packets in stream i. The basic idea of these adjustments is to adjust loss numerators and denominators for all buffered packets in the same stream i as the packet most recently transmitted before its deadline. The details of these adjustments appear in [17, 18]. Here, it suffices to say that DWCS has the ability to implement a number of real-time and non-real-time policies. Moreover, DWCS can act as a fair queueing, static priority and earliest-deadline first scheduling algorithm, as well as provide service for a mix of static and dynamic priority traffic streams.

4.5 Programmable Hardware Support

The explosive growth in the functionality of configurable or programmable hardware in the form of field programmable gate arrays (FPGAs) is changing the architecture of information systems that deal with data intensive computations [10, 11]. For example, we have seen the advent of configurable computing systems where programmable hardware is coupled with programmable processors. Hardware/software co-design has emerged as an associated design paradigm where the programmable hardware components (in the form of FPGAs) and software components (executed on processors) are designed concurrently with efficient trade-offs across the HW/SW boundary. While this paradigm is largely targeted towards embedded computer systems, we can apply relevant concepts to the design of intelligent network interfaces.

FPGA devices effectively represent hardware programmable alternatives to system level application specific integrated circuit (ASIC) designs and at a drastically reduced cost. Modern devices can be dynamically and incrementally re-programmed in microseconds, have increased memory on chip, and operate two to four times faster than current chips. FPGA devices perform particularly well on regular computations over large data sets. This technology naturally fits in architectures that stream and operate on large amounts of data as in the bulk of emerging network-based multimedia applications. The hardware functionality in the interfaces can be re-programmed "on the fly" almost as if we were swapping out custom devices.

We are motivated to include FPGA devices in the NIs for two specific reasons. The first is to host priority biasing calculations. Such calculations are inherently parallel with priorities being computed across many connections. A large number of relatively simple independent computations can be effectively supported within FPGAs. However, the overhead of such computations can render software NI implementations of certain biasing calculations either infeasible or greatly reduce the link utilizations that can be achieved. The second reason is the ability to host certain classes of data stream computations in the network interface. For example, data filtering, encryption, and compression are candidates for implementation with FPGAs available in the NI. Such computations can be naturally performed on data streams during transmission rather than via (relatively) expensive traversals through the memory hierarchy to the CPU. The VCM environment can provide access to these hardware devices via extension modules that can be used to load configuration data into the FPGAs. Thus, the abstractions used to extend the NI functionality dynamically are the same for functions implemented in software or programmable hardware. Our goal is to leverage this FPGA technology to enable more powerful yet (relatively) inexpensive network interfaces that can substantially enhance the performance of network applications.

5 Concluding Remarks

The goal of the QUIC project is the development of an effective quality management infrastructure that can service a large number of connections, each with distinct service requirements. Towards this end we are constructing an experimental infrastructure using a cluster of PCs interconnected by fast ethernet using i960 based interface cards. At the center of our efforts is an extensible software and quality management infrastructure. By placing quality management functionality "close" to the wire and I/O components attached to the I2O boards, we expect to enable QoS guarantees at higher levels of resource utilization than commodity clusters will otherwise permit. Our current efforts are geared towards creating a rapid prototyping environment to provide a basis for experimentation and investigation.

References

[1] K. Dienfendorff and P. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, vol. 30, no. 9, pp. 43-45, September 1997.

[2] Richard P. Martin and Amin M. Vahdat and David E. Culler and Thomas E. Anderson. Effects of Communication Latency, Overhead and Bandwidth in a Cluster Architecture. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[3] Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, Deborah A. Wallach, and William E.

Weihl. Efficient implementation of high-level languages on user-level communication architectures. *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, 1995.

[4] Marcel-Catalin Rosu and Karsten Schwan and Richard Fujimoto, Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach. *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, August 1997.

[5] M. Rosu, K. Schwan, and R. Fujimoto. Supporting parallel applications on clusters of workstations: the virtual communication machine-based architecture. *Cluster Computing*, pp. 1029, November 1997.

[6] C. E. kozyrakis and D. Patterson. A new direction for computer architecture research. *IEEE Computer*, vol. 31, no. 11, pp. 24-32, November 1998.

[7] A. A. Chien and J. H. Kim. Approaches to quality of service in high performance networks. *Proceedings of the Workshop on Parallel Computer Routing and Communciation*, pp. 1-20, June 1997.

[8] J. H. Kim. Bandwidth and latency guarantees in low-cost high performance networks. Ph.D. Thesis, Department of Computer Sciences, University of Illinois, Urbana-Champaign, January 1997.

[9] D. Garcia and D. Watson. ServerNet II. *Proceedings of the Workshop on Parallel Computer Routing and Communication*, pp. 119-136, June 1997.

[10] J. Villasenor and W. H. Mangione-Smith. Configurable computing *Scientific American*, pp. 66-71, June 1997.

[11] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, H. Spaanenburg. Seeking solutions in configurable computing. *IEEE Computer*, vol. 30, no. 12, pp. 38-43, December 1997.

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

[13] I_2O Special Interest Group. www.i2osig.org/architecture/techback98.html.

[14] Intel. *i960 Rx I/O Microprocessor Developer's Manual*, April 1997.

[15] Intel. *IQ80960Rx Evaluation Platform Board Manual*, March 1997.

[16] Jon M. Peha and Fouad A. Tobagi. A cost-based scheduling algorithm to support integrated services. In *IEEE INFOCOMM'91*, pages 741–753. IEEE, 1991.

[17] Richard West and Karsten Schwan. Dynamic window-constrained scheduling for multimedia applications. Technical Report GIT-CC-98-18, Georgia Institute of Technology, 1998. To appear in the 6th International Conference on Multimedia Computing and Systems, ICMCS'99, Florence, Italy.

[18] Richard West, Karsten Schwan, and Christian Poellabauer. Scalable scheduling support for loss and delay constrained media streams. Technical Report GIT-CC-98-29, Georgia Institute of Technology, 1998.

[19] WindRiver Systems. *VxWorks Reference Manual*, 1 edition, February 1997.

[20] WindRiver Systems. *Writing I_2O Device Drivers in IxWorks*, 1 edition, September 1997.