QoSockets: a New Extension to the Sockets API for End-to-End Application QoS Management

*P. G. S. Florissi**
System Management Arts (SMARTS)
14 Mamaroneck Avenue, 3rd Floor
White Plains, NY
USA
patricia.florissi@smarts.com

Y. Yemini, D. Florissi
DCC Lab, Columbia University
500 West 120th Street, Room 450
New York, NY
USA
{yy,df}@cs.columbia.edu

Abstract

Distributed multimedia applications are sensitive to the Quality of Service (QoS) delivered by underlying communication networks. The main question this work addresses is how to adapt multimedia applications to the QoS delivered by the network and vice versa. We introduce QoSockets, an extension to the sockets mechanism to enable QoS reservation and management. QoSockets automatically generates the instrumentation to monitor QoS. It scrutinizes interactions among applications and transport protocols and collects in QoS Management Information Bases (MIBs) statistics on the QoS delivered. The main advantages of QoSockets are the following. (1) Support of single API for transport layer QoS negotiation, connection establishment, and data transmission; and of single API for OS QoS negotiation. (2) Support of a single QoS negotiation protocol. (3) Generality across application QoS needs. (4) Automatic management of application QoS needs. QoSockets are available for Solaris and Linux and support RSVP, ATM adaptation, ST-II, TCP/UDP, and Unix native protocols.

Keywords

Quality of Service (QoS) management, management of distributed systems and applications.

1. Introduction

Traditional network applications can operate under a broad range of network performance behaviors. They can tolerate very large end-to-end latency, accommodate greatly varying bandwidth, recover from loss, and endure dynamic fluctuations in latency and bandwidth. In contrast, distributed multimedia applications are very sensitive to the performance behavior of networks [12]. A multimedia conference can be very sensitive to significant latency. Video streams

* Work performed while Patricia was pursuing her Ph.D. degree in the DCC Lab at Columbia University.

require guaranteed bandwidth. Speech streams become incomprehensible under excessive jitter (i.e., dynamic fluctuations of latency).

Recent studies of QoS delivery have focused on the design of network mechanisms to assure QoS. These range from the design of *Asynchronous Transfer Mode* (*ATM*) [4] protocols and switching mechanisms to the design of multimedia transport protocols. Mostly, these mechanisms have focused on regulating competition for network resources among traffic sources. They involve resource allocation, flow, and admission control techniques used by packet/cell and transport layers. References [1,10,14,18] cover some important contributions in this field.

This work uniquely focuses on the design of application-layer mechanisms to support effective adaptation to and control of QoS delivery. We introduce QoSockets, an unified set of extensions to the sockets Application Program Interface (API) [17]. that hides the heterogeneity in transport protocols support of QoS assurance. QoS support at the transport layer ranges from no QoS support (such as in UDP [17]), to limited support (such as in TCP [17]), and up to intricate assistance (such as in RSVP [1]). Even when providing considerable QoS support, protocols disagree on the metrics for negotiation and on the negotiation mechanism supported [11]. *The lack of transport homogeneity renders development of portable applications difficult in many ways.* (1) Application programmers need to be aware and handle the gap between the QoS needed and the one effectively supported by a particular provider. (2) Applications must handle details of QoS negotiation protocols. (3) Differences in the semantics of QoS assurance cause applications to work differently under different system configurations.

Similar complexities in assuring QoS result from OS heterogeneity. For example, if the OS does not support real time computations, an application may miss the deadline to decode a video frame in time to display it. Such *application needs to know about OS performance behavior and react accordingly.* These issues are beyond the scope of this paper, but we refer the reader to [3,5,8,16] for a good coverage on the subject.

QoSockets add to existing communication APIs (such as Berkeley sockets [17]) the ability to specify QoS constraints (e.g., delay or jitter) of a transport protocol. The main contributions of QoSockets are: (1) Support of single API for transport layer QoS negotiation, connection establishment, and data transmission. (2) Support of a single QoS negotiation protocol. (3) Generality across application QoS needs. (4) Automatic monitoring of the QoS delivered by the underlying system and automatic detection of violations of QoS assurance. (5) Support of a flexible mechanism to dynamically select most appropriate QoS transport providers given specific application requirements.

QoSockets may extend most socket mechanisms, including Unix socket interface or the WinSock interface.

This paper will concentrate on the design of QoSockets. The very important issue of performance of QoSockets are left for a future publication due to space limitation. Interested readers may want to refer to [6] for some early experimental performance evaluations for some sample applications.

The reminder of this paper is organized as follows. Section 2 describes how an application developer uses QoSockets to specify QoS constraints in a communication and how QoSockets allocate underlying system resources. Section 3 discusses how the QoSockets runtime hides from application developers the heterogeneity of transport protocols. Section 4 shows the QoSockets mechanisms for application level QoS management. Finally, Section 5summarizes.

2. Specification of QoS Constraints in QoSockets

This section overviews QoSockets QoS specification by using sample examples. QoSockets supports two types of QoS metrics: *resource level QoS metrics* and *application specific QoS metrics*. Resource level QoS metrics provide performance measures of the underlying system in which an application operates. These include *universal communication QoS metrics* and QoS metrics related to computations. The universal QoS metrics are loss, permutation, rate, end-to-end delay, jitter, and connection recovery time (they are formally defined in [6]). QoSockets use these metrics to allocate and manage necessary underlying system resources. Application specific QoS metrics are application dependent performance measures of communications. For example, a video conference application may specify resource level QoS metrics such as *rate* and *delay* to indicate how the runtime should allocate communication resources. In addition, it may define application specific metrics that indicate how synchronized its audio and video streams should be, that is, if each video and corresponding audio frames arrive at the same time.

Resource level and application specific QoS metrics differ in purpose. Both specify how to perform QoS monitoring but only the first specifies allocation of system resources. For example, the *delay* metric specifies indirectly the strategy to allocate bandwidth and buffers while *rate of late messages* only specifies how to monitor the stream. It is left for future work mapping of application specific QoS monitoring into resource allocation strategies.

QoSockets use a coercion mechanism to upgrade a less restrictive constraint until it matches a more restrictive one. During binding time, it checks the QoS specification in the communicating ports and tries to find a QoS allocation that will satisfy the most stringent requirements. One simple example is a connection where the sender specifies a minimum rate of 1 Mb/s while the receiver needs at least 500 Kb/s. The coercion mechanism will try to allocate at least 1 Mb/s for this connection.

Figure 1 specifies the *qos_ty* data type that enables the declaration of universal QoS metrics. For each universal metric, applications can specify a tolerable threshold value (field *value*), windows over which the metric should be measured (field *window*), and if the threshold can be coerced (field *coercion*) when binding with other sockets.

Applications specify constraints on a per port basis by associating a different *qos_ty* object with each port. For example, values 3 and 5 in the fields *value* and *window* of *delay* for port *p* specifies that the average delay cannot have a value higher than 3 ms over intervals of 5 s on communications over *p*. The runtime uses

size, when specified, to optimize resource allocation. Ports can support a maximum of *multiple* concurrent connections at a time. A positive value in *combined* indicates that the *min_rate* and *rate* QoS constraints refer to the rate of all the connections combined. When *combined* is not specified, each connection generates the *min_rate* and *rate* specified. The following section discusses how constraints can be associated to ports.

```
/* Definition of a QoS metric */
typedef struct qos_metric {
   int value        /* Tolerable threshold for QoS metric */
   int window;      /* How often (in seconds) the QoS metric must be measured */
   int coercion;    /* If the threshold can be coerced at binding time */
} qos_met_ty;

/* Definition of Universal QoS Metrics in QoSockets */
typedef struct qos {
qos_met_ty loss;       /* Loss not higher than 10 to the power -loss.value */
qos_met_ty permt;      /* Permutation is tolerated (if value not 0) */
qos_met_ty min_rate; /* Mean rate measured in messages/s */
qos_met_ty rate;       /* Mean rate measured in messages/s */
qos_met_ty peak;       /* Peak transmission rate is not higher than peak.value */
qos_met_ty delay;      /* End-to-end delay measured in ms */
qos_met_ty jitter;     /* Jitter measured in ms */
qos_met_ty recovery; /* Any recovery must take less than recovery.value */
int size;              /* Maximum message size */
int multiple;          /* Maximum of multiple connections concurrently */
int combined ;         /* Measure the QoS on all connections combined */
} qos_ty;
```

Figure 1: Specification of QoS metrics in QoSockets.

A *NULL* value for a *qos_ty* object field indicates that the application chooses not to specify that particular constraint and leaves it up to the runtime. Alternatively, the application provides QoS constraints and let the QoSockets runtime choose the best service provided by the transport for the request. Such range of option provides a rich set of semantic QoS negotiation possibilities for applicatios.

QoSockets runtime bridges the gap between the QoS assurance model chosen by applications and the one deployed by a network. In addition, it automatically monitors the execution of applications and dynamically re-negotiates QoS with the network to match application demands. Section 4 discusses how data collected during monitoring permits clever network management policies to adjust QoS delivery according to observed QoS behavior. QoSockets provide a single API for QoS specification that is independent of underlying transport mechanism details. The

runtime translates abstract QoS specifications into service requests specific to the underlying transport.

The semantics offered by QoSockets are that (1) it negotiates QoS with the network on a *best effort basis* and that (2) violations must be handled by applications. In a best effort QoS delivery, networks multiplex their resources in an effort to best fit the QoS requested without wasting resources. Stochastic models are used to characterize data traffic sources and predict when and where resources are needed. Nevertheless, there is no guarantee that violations will not occur during transient overload periods. As a consequence, QoSockets applications must be designed to handle violations and to adapt accordingly.

3. QoSockets Connection Establishment Protocol

QoSockets *unify* several connection establishment protocols in one, promoting code portability and reuse. Figure 2 shows the time line for the typical communication scenario using QoSockets and Figure 3 shows the QoSockets API system calls. In Figure 2, rectangles represent QoSockets function calls and the straight arrows represent execution flows. Time increases from top to bottom direction. Two execution flows are depicted: the Sender application and the Receiver application. The dashed arrows represent events handled by the QoSockets runtime concurrently with the execution of other tasks. These events are triggered by the system call where the arrow initiates. The balloon indicates when QoS negotiation happens.

Ports in QoSockets are identified by name (of type string) and do not need to be bound to a specific transport level port number. This feature increases code portability by preventing application failure due to conflicts on the allocation of transport level addresses. The name of a QoSockets port and its QoS requirements are defined at allocation time. Inports and outports are allocated, respectively, through the *qos_alloc_inport* and *qos_alloc_outport* operators calls. In Figure 2, Receiver calls *qos_alloc_inport* to allocate inport *ma* (short for Massachusetts) with the QoS requirements expressed in the variable *rqos* of type *qos_ty* (Figure 1). At the end of the call, variable *rp* points to a descriptor for the allocated port. The last three arguments are optional and indicate the family (Unix internal protocol, Internet protocol, etc.), type (stream socket, raw socket, etc.), and protocol (if a specialized one like ICMP [17], SPP [17], etc., is needed). These arguments can have a *NULL* value in which case the QoSockets runtime automatically selects them based on the QoS requirements and on the protocols supported by the communicating machines. Consider, for example, an application running on a distributed environment that supports AAL and TCP. QoSockets runtime selects AAL when the application specifies QoS constraints and TCP when it does not.

The binding mechanism in QoSockets incorporates QoS negotiation between peer applications in the sockets mechanism. In Figure 2, the call to *qos_export* publishes to other QoSockets applications all the information associated with inport *rp*, such as its name and QoS requirements. QoSockets publish port related information by using name servers [9] to store and access the information published. On the sender side, *qos_import* binds outport *sp* with the inport *ma* available on

machine *mit.edu*. Operator *qos_import* first accesses the name server to retrieve information on a particular inport. It checks the QoS restrictions of *sp* with the ones retrieved from the name server and decides whether or not they are *compatible* (that is, if there is a QoS allocation that can satisfy both ports). If they are, the ports are bound and connection can be established any time after that. Otherwise, *qos_import* returns an error and indicates why they are not compatible.

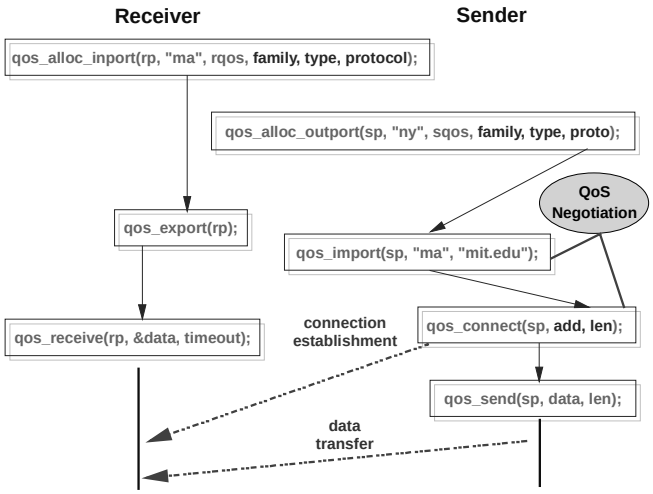

Figure 2: QoSockets function call sequence to establish a communication.

QoSockets establish connections as follows. Operator *qos_connect* triggers the connection establishment at the sending side. It blocks until connection establishment has been initiated. The last two optional arguments are used to identify the connecting inport by its physical address, when necessary. The transport service provider for the communication (if no transport was specified when the connecting ports were allocated) is allocated by *qos_connect*.

At the receiver side, QoSockets runtime frees applications from servicing connection requests and connection establishment details. The QoSockets runtime process incoming requests, accepting or rejecting connections based on their QoS needs and on the QoS offered by the transport service provider chosen for the communication. At the sender side, QoSockets runtime manages connection establishment confirmations or rejections without exposing applications to such details.

In synchronous protocols [2], connection might have already been established by the time *qos_connect* returns. In asynchronous protocols [18], connection establishment has only been initiated when *qos_connect* returns.

Operators *qos_send* and *qos_receive* are for data transmission. Operator *qos_send* sends through *sp* a message that is up to *len* bytes long stored in *add*. Similarly, *qos_receive* blocks for up to *timeout* milliseconds waiting for a message to arrive for *rp*. If a message arrives within the time period specified, *qos_receive* retrieves it and stores it in the memory designated by *data*.

qos_alloc_inport(inport_ty *port_ref, qos_ty qos_ref, int family, int type, int protocol);	Allocates an inport with the QoS constraints specified in *qos_ref* for transmission over a given communication *family*, *type*, and *protocol*. Returns in *port_ref* a reference to the inport created.
qos_alloc_outport(outport_ty *port_ref, qos_ty qos_ref, int family, int type, int protocol);	Allocates an outport with the QoS constraints specified in qos_ref for transmission over a given communication family, type, and protocol. Returns in port_ref a reference to the outport created.
qos_export(inport_ty *port_ref, char *external_name);	Publishes the QoS constraints and protocol specific addresses associated with *port_ref*. The information published is identified by *external_name*.
qos_import(outport_ty *port_ref, char *external_name, char *machine_name);	Connects port_ref to the port identified by external_name available on machine_name.
qos_connect(outport_ty *port_ref, struct sockaddr *addr, int addrlen);	Connects *port_ref* to the address specified in *addr*. *addrlen* has the size of the *addr* data structure.
qos_send(outport_ty *port_ref, char *data_ref, int len);	Sends *len* bytes of data stored in *data_ref* through *port_ref*.
qos_receive(inport_ty *port_ref, char *data_ref, struct timeval *timeout);	Blocks for a maximum of *timeout* waiting for data to arrive in *port_ref*. Saves in *data_ref* the first message that arrives before *timeout* expires.
qos_wait_inport_connected(inport port_ref, struct timeval *timeout);	Blocks for a maximum of *timeout* waiting for a connection to be established in *port_ref*.
qos_wait_outport_connected(outport port_ref, struct timeval *timeout);	Blocks for a maximum of timeout waiting for a connection to be established in port_ref.
qos_bind(port_ty *port_ref, struct sockaddr *addr, int addrlen);	Assigns the protocol level address specified in addr to port_ref. addrlen has the size of the addr data structure.

Figure 3: QoSockets API system calls.

Operator *qos_send* and *qos_receive* block until a connection is fully established on the port transmitting data. Blocking may be avoided by using the operators *qos_wait_outport_connected* and *qos_wait_inport_connected* before the first call to *qos_send* and *qos_receive*, respectively. These operators block until connection has been fully established, but do not transmit any data.

4. Managing QoS Delivery

The network management and the application QoS adaptation strategies will accomplish better results through coordination. Network management may improve the QoS in application streams by allocating alternative routes. Applications may operate under QoS degradation by adapting their streams to the QoS received. Without coordination, these activities may settle for unsatisfactory or unstable operational points. For example, upon congestion at a switch, SNMP [13] managers may decide to allocate alternative routes and, concurrently, applications may reduce

their transmission rates. *Both* applications and managers need to understand requested and delivered QoS to coordinate their efforts.

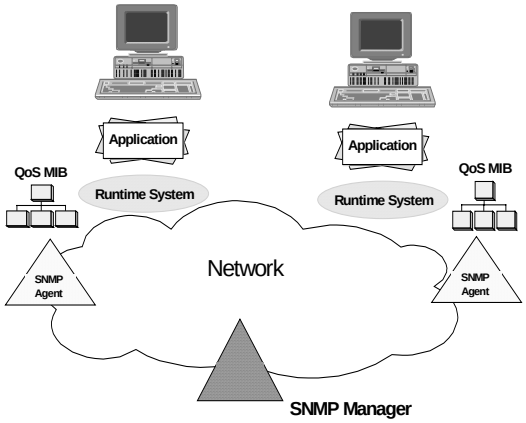

Figure 4: Overall architecture for instrumentation and access of QoS MIBs.

One of the main contributions of QoSockets is integrating application level QoS management and underlying system management. The QoSockets management architecture consists of QoS MIBs [15] and SNMP agents that provide QoS MIB access to SNMP managers. The objects in these MIBs deal with application level information, such as video frame delays and voice stream jitter. The information is partitioned among MIB groups according to applications, outports, inports, and programmable (application specific) metrics. The proposed architecture has the main novel advantages:

- It includes a mechanism to disclose application level QoS performance to underlying system managers. By accessing QoS MIBs, managers of transport layer connections can identify, for example, the application that is using a particular connection, the QoS the application requested, and the QoS that is being delivered to it. This information may be used to decide alternative allocation policies or to pin point applications that are overloading network resources.
- It contains the necessary information to characterize application QoS behavior. QoS MIBs store the QoS requested by applications and measurements on the QoS being delivered to them. These measurements include the value of universal metrics (such as end-end delay, jitter, loss, etc.) and application specific ones.
- It includes coordination of application and SNMP management activities. This is useful when applications and managers detect violations and try to compensate for them. They should coordinate their activities to avoid interfering with each other's decisions. For example, an SNMP manager may decide to allocate more throughput in some network links to overcome congestion while applications

may decide not to decrease their transmission rates because they are aware of this management decision.

4.1. Overview of the QoSockets Management Architecture

Figure 4 illustrates the architecture for QoS management through QoS MIBs using a generic multimedia multi-application example. The applications sample input devices (such as monitors, cameras, and microphones), broadcast them to other participants, and finally display received samples locally. The runtime instances at each site support interactions between applications and the underlying transport and OS, and store in QoS MIBs information on the QoS effectively received. Examples of such information are the amount of bandwidth allocated and received in the communication. SNMP agents embedded in the architecture provide QoS MIB access to SNMP managers.

Applications read QoS MIB fields to detect QoS violations and update them to trigger corrective actions. Consider, for example, an inport receiving video data that requires a delay not higher than 5 ms over windows of 1 s. The QoSockets runtime automatically monitors delay variations and store them in MIBs. Video play-out time may require adjustment when the average delay is lower than a certain threshold (for example, 3 ms). Applications need to query QoS MIB objects to detect such situations.

SNMP managers use QoS MIB data to manage QoS delivery based on application needs. These managers may get information about the configuration of applications running on the system and can customize their service management accordingly. For example, when the delay on a communication is higher than the application expected, a manager can initiate the establishment of an alternative connection. Managers can also use information on other SNMP MIBs to aid the analysis and control of QoS violations. For example, by monitoring ATM switch MIBs, a manager can force communication establishment to bypass congested switches.

SNMP managers use information on other SNMP MIBs to aid the analysis of QoS violations. For example, an SNMP manager may monitor ATM switch MIB values to understand the QoS in communications between applications. The SNMP manager traces, for example, cases where unexpected transmission delays are due to congestion in the switch. In such scenario, managers can automatically request the establishment of an alternative connection that bypasses the congested switch.

QoS MIBs integrate application management within general network management frameworks. This feature is important because it may become inefficient or intractable to manage distributed application activities using only lower layer information. This difficulty comes from the increasing gap between application level abstractions and underlying system entities providing services. The architecture presented provides a framework for dividing management responsibilities between SNMP managers and applications. On one hand, authorized SNMP managers can access QoS MIBs and manage the underlying system according to application needs.

On the other hand, applications can manage themselves, according to the received QoS.

4.2. An Overview of the QoS MIB Design

The QoS MIB data belongs to one of the following groups (as depicted in Figure 5):

- Application (qApp for short): Consists of the table qAppTable that contains one entry of type qAppEntry for each application running on the system. Each entry indicates the QoS provided by the underlying OS and general information on the response of the protocol stack to QoS demanding connection establishment requests. For example, the application group object qAppLSchFl stores the last time when the OS failed to schedule an application according to its timing constraints and qAppLInCnnFl stores the last time when an application had a connection establishment request rejected. This group can be seen as an extension of the NSM MIB [7] to add information about application QoS.
- Outport (qOut): Consists of the table qOutTable which has one row of type qOutEntry for each outport connection of applications in qApp. Each entry indicates the QoS negotiated for the outport and how the outport is using the connection. For example, the object qOutMaxRate indicates the rate negotiated with the network at connection establishment time, qOutMsgSent indicates how many messages have been sent so far, and qOutActTime indicates when the connection became active. A manager can calculate the average transmission rate effectively received by dividing qOutMsgSent by the time elapsed since qOutActTime. It can then compare the result with qOutMaxRate which holds the negotiated rate.

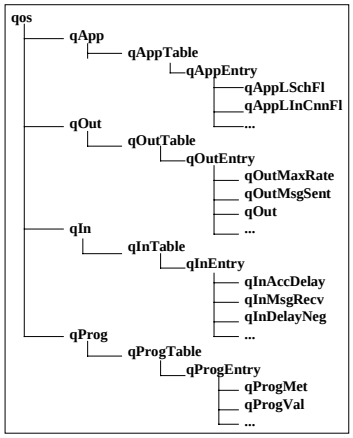

Figure 5: QoS MIB object groups.

- Inport (qIn): Consists of the table qInTable which has one row of type qInEntry for each inport of applications in qApp, similar to qOutEntry. In addition, it maintains measures on the QoS effectively delivered by the network. For

example, the qInDelayNeg, qInAccDelay, and qInMsgRecv objects indicate the transmission delay negotiated with the network, the sum of the transmission delays of all messages received, and the number of messages received, respectively. An SNMP manager uses these data to establish alternative paths for connections that are experiencing a mean transmission delay much higher than the one negotiated. The mean transmission delay is calculated by dividing qInAccDelay by qInMsgRecv.

- Programmable (qProg): Consists of the table qProgTable which has one row of type qProgEntry for each application programmed (application specific) QoS metric. An entry in qProgTable indicates the metric that is being measured, the application that requested the measurement, and the last value measured. For example, the objects qProgMet identifies a metric and the object qProgVal indicates the last value measured. Entries in this group are added or removed as applications trigger or cancel monitoring of new QoS metrics. Metrics that use the window mechanism to specify the measurement frequency are stored in this table. For example, average delay is measured over a window of time that defines what is the frequency at which such measure is to be computed.

#	Object	Syntax	Description
01	qOutProtocol	OBJECT IDENTIFIER	Identification of the protocol being used for this connection
02	qOutLoss	INTEGER	Probabilistic message loss rate $(10^{(-qOutLoss)})$
03	qOutPermut	“yes” | “no”	Indication of tolerance to permutation
04	qOutMinRate	INTEGER	Minimum number of messages per second
05	qOutMaxRate	INTEGER	Maximum number of messages per second
06	qOutPeak	INTEGER	Peak number of messages per second
07	qOutDelay	INTEGER	Maximum propagation delay
08	qOutJitter	INTEGER	Maximum jitter
09	qOutRecTime	INTEGER	Maximum time tolerated for recovery
11	qOutMsgSize	INTEGER	Maximum message size in bytes
12	qOutManager	OBJECT IDENTIFIER	Entity currently controlling communication QoS violations

Table 1: Configuration Outport Group Objects

4.3. QoS MIB Data per Outport

This section illustrated the QoS MIB data of output port group. The reader can refer to [6] for other groups and for more details. The information stored by the columnar objects of the MIBs presented here are classified in one of the following categories: (1) Identification: used to describe a particular instance of an object; (2) Configuration: used to identify how resources were allocated for a service; (3) Operational behavior statistics: used to analyze the actual performance delivered by

the underlying system, and (4) Coordination: used to synchronize management actions between applications and SNMP managers.

The goal of the outport group is to inform about outport connections, their QoS requirements, how they are being utilized, and to coordinate management of their QoS performance between applications and SNMP managers. This group also includes information on connection problems and recovery performance.

Identification objects store the local and remote addresses of the communicating machines, identifiers of the applications involved, and the transport layer port numbers of the connection. If a connection is currently presenting problems, managers use such objects to identify the applications involved and properly notify them. Similarly, if an application terminates abruptly, managers can look in the outport MIB for its connections and gracefully terminate them.

Configuration Outport Group Objects

Table 1 shows the configuration objects present in the outport group. SNMP managers use configuration objects to guide the allocation of communication resources per outport connection. The qOutMsgSize object indicates the maximum size of messages transmitted on a connection. The qOutProtocol object identifies the transport protocol serving the connection. The qOutLoss, qOutPermut, qOutMinRate, qOutMaxRate, qOutPeak, qOutDelay, qOutJitter, and qOutRecTime objects identify the QoS constraints negotiated for the outport. These data enable an accurate analysis of the resources allocated per connection.

#	Object	Syntax	Description
01	qOutCnnFail	Counter32	Total number of connection failures
02	qOutAccRecTime	INTEGER	Total time spent recovering
03	qOutEstTime	TimeStamp	Time of connection establishment
04	qOutActTime	TimeStamp	Time when the traffic became active
05	qOutMsgSent	Counter32	Total number of messages sent
06	qOutVolume	Counter32	Total volume of data sent in kilobytes
07	qOutLstMsg	TimeStamp	Time when the last message was sent through the connection
08	qOutLstFail	TimeStamp	Time of last connection problem
09	qOutStatus	“up” | “down”	Status of the connection

Table 2: Operational Behavior Statistics Outport Group Objects

Consider, for example, an application that receives radiology images and occupies most of the communication resources on a machine. If other applications are unable to open connections, a local SNMP manager can use qOutMaxRate, qOutPeak, and qOutMsgSize object instances to calculate how buffering resources are currently distributed. The manager may then realize that the amount of bandwidth negotiated by the radiology application corresponds to a great percentage of the resources the machine has available. A manager might force the radiology application to downgrade the QoS negotiated making possible for other applications

to communicate concurrently. The qOutManager object enables management coordination between applications and SNMP managers.

Operational Behavior Statistics Outport Group Objects

Table 2 illustrates operational behavior outport group objects. QoS managers use qOutCnnFail and qOutAccRecTime object instances to estimate recovery time from connection failures and manage QoS performance accordingly. For example, the average recovery time can be calculated by dividing qOutAccRecTime by qOutCnnFail. Thus, an application unable to send data over a connection due to a failure can decide whether to open an alternative connection or to wait for recovery based on the mean recovery time.

SNMP managers use operational behavior objects, such as qOutMsgSent, and qOutVolume, to evaluate how much of the resources allocated by an application are actually being used, and to re-negotiate QoS if the utilization ratio is low. An SNMP manager reduce a communication allocation from 30 frame/s video to 15 frame/s if the application has not sent more than 15 frames/s recently. By detecting under-utilization, managers can allocate resources more efficiently.

5. Conclusions

This paper presents the QoSockets APIs that promote code portability and reusability by sheltering heterogeneity in the QoS functions offered by several transport protocols. The main contributions of such approach are: (1) A single API that is independent of transport layer specifics. The same application can use services from several transport protocols without any modification. (2) The runtime offers a single QoS negotiation mechanism which automatically bridges gaps among different transport protocol providers. (3) Upgrades to support new protocols or new OSs can be accomplished by extending the runtime with the interface to the new architecture components. (4) The QoSockets runtime can automatically select the most appropriate transport given QoS requirements. (5) The runtime can automatically monitor the QoS delivered with low overhead. The collected data may be accessed by other local applications as well as external SNMP managers.

QoSockets also include an architecture for QoS management using QoS MIBs. QoS MIB data identify how communication and processing resources are allocated and utilized by applications. Applications use QoS MIB data to detect QoS violations and adapt accordingly. SNMP agents in the architecture provide QoS MIB access to SNMP managers that may use this information to manage resources according to the QoS delivered to applications. QoS MIB objects also include control information to coordinate QoS management between applications and SNMP managers.

Multiple experiments have been conducted using QoSockets, which are reported in [6]. As expected, they simplified many aspects of the implementation of such applications, including multiple protocol support, automatic QoS monitoring, and portability. It was shown that the overhead introduced by QoSockets is not negligible (200 μsec in a SPARC 20) but it is constant for any message size. The throughput is not considerably affected because the typical time to generate a message is much

larger than the time to process QoSockets API and protocols. For more details on the overhead, we refer the readr to [6].

A partial prototype of QoSME has been released for public access. It runs on Solaris 2.5 and Linux and supports communication on RSVP, ATM adaptation layer, ST-II [18], UDP/IP, TCP/IP, and Unix internal protocols.

References

[1] R. Braden, L. Zhang, D. Estrin, S. Herzog, and S Jamin. Resource ReServation Protocol (RSVP) -- Version 1 Functional Specification. Internet Draft, 1995.

[2] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP Volume 1*. Prentice Hall, NJ, 1991.

[3] G. Coulson, A. Campbell, and P. Robin. Design of a QoS Controlled ATM Based Communication System in Chorus. *IEEE JSAC,* May 1995.

[4] M. De Prycker. *Asynchronous Transfer Mode: solution for Broadband ISDN*. Ellis Horwood Limited, Second Edition, 1993.

[5] D. Feldmeier. *A Framework of Architectural Concepts for High Speed Communication Systems.* Technical Report, Bellcore, Morristown, May 1993.

[6] P. G. S. Florissi. QuAL: Quality Assurance Language. PhD Thesis, Computer Science Department Columbia University, New York, NY, 1995.

[7] N. Freed and S. Kille. Network Service Monitoring Management Information Base. Internet Draft, 1995.

[8] R. Govindan and D. P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles,* USA, SIGOPS, vol. 25, pp 68-80, 1991.

[9] F. Halsall. *Data Communications, Computer Networks and Open Systems.* Addison Wesley, 1992.

[10] D. B. Hehmann, R. G. Herrtwich, W. Schulz, T. Schuett, and R. Steinmetz. Implementing HeiTS: Architecture and Implementation Strategy of the Heidelberg High Speed Transport System. In *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video,* IBM ENC, Heidelberg, Germany, November 1991.

[11] S. Keshav. Report on the Workshop on Quality of Service Issues in High Speed Networks. *ACM Comp. Comm. Review,* vol. 22, no. 1, pp. 6–15, January 1993.

[12] A. Lazar. Challenges in Multimedia Networking. In *Proceedings of the International Hi-Tech Forum,* Osaka, Japan, February 1994.

[13] M. T. Rose. *The Simple Book.* Prentice-Hall, 1993.

[14] H. Schulzrinne and S. Casner. RTP: A Transport Protocol for Real-Time Applications. Internet Draft, draft-ietf-avt-rtp-05, 1995.

[15] W. Stallings. *SNMP, SNMPv2, and CMIP*.Addison Wesley, 1993.

[16] Stankovic. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer, Issue on Scheduling and Real-Time Systems,* June 1995.

[17] W. R. Stevens. *UNIX Network Programming*. Prentice-Hall, New Jersey, 1990.

[18] C. Topolcic. Internet Stream Protocol, Version 2 (ST-II). Internet Requests for Comments (RFC) 1190, October, 1990.