A Quality-of-Service Enhanced Socket API in GNU/Linux

Hasan Abbasi, Christian Poellabauer, Karsten Schwan, Gregory Losik
College of Computing,
Georgia Institute of Technology,
Atlanta, GA 30332
{habbasi,chris,schwan,losik}@cc.gatech.edu

Richard West
CS Department
Boston University
Boston, MA 02216
richwest@cs.bu.edu

Abstract

With the increase in available network bandwidth and reduction in network latency, more emphasis has been placed on end-host QoS to provide quality of service guarantees for distributed real-time applications, such as video streaming servers, video conferencing applications, and VoIP applications. End-host QoS requires support from the underlying operating system (OS) in the form of network schedulers and traffic controllers. GNU/Linux supports a variety of network schedulers for the provision of end-host QoS, but no easy-to-use API is provided between applications and system-level traffic controllers. Moreover, existing APIs lack the ability to link QoS attributes with stream or datagram sockets and to associate QoS information with individual packets allowing sophisticated network schedulers to make scheduling decisions based on such information or to link different streams such that their QoS management is coordinated. We propose an extension to the standard Berkeley socket interface that enables applications to easily create and manage connections with QoS requirements. This new API – called QSockets – includes per-socket traffic classification that allows the application developer to link a socket connection with a scheduler so that all data going out on this socket is scheduled accordingly. We further allow an application to pass packet-specific parameters to the scheduler for fine-grained scheduler control. We evaluate our approach with a multimedia application using QSockets and a real-time packet scheduler.

1 Introduction

1.1 Background

There are a considerable number of applications today that require some form of guarantee from the *Network Subsystem* with regards to Quality of Service (*QoS*) parameters for reliable operation. Such applications are usually continuous media applications for soft real-time video and audio streaming, video-conferencing and Internet telephony. There has been a consistent push toward creating an Internetworking infrastructure that can provide seamless guarantees to such applications. The two most commonly accepted proposals are *Integrated Services* and *Differentiated Services* [2].

These proposals, however, require substantial modifications to the underlying network model for providing end-to-end guarantees. With high speed intranets available today, it is possible to obtain reasonable guarantees for delay and bandwidth using only end-system QoS. While these guarantees are not absolute they do provide a good approximation without requiring any major changes to the physical network.

In [3] the authors show that even in the presence of network level QoS support, it is often necessary for the end-system to provide not only service differentiation but also traffic shaping. Moreover this shaping has to be provided on a per-application basis to provide maximum functionality for the application and system developers.

Despite having comprehensive scheduling functionality in GNU/Linux [7], there is no support (in terms of either API or classifiers) for exposing this functionality to the application. Instead the traffic controller functionality is designed for use as a node on a DiffServ network. The lack of application level differentiation as well as a comprehensive API for addressing the shaping of application traffic forces the developer to resort to application level rate control. Such rate control is inefficient but highly flexible. Each stream and even individual messages can be shaped according to the needs of the application. While flexible, using such methods can be inefficient and does not scale well for high system loads, especially when each stream and individual messages have to be shaped (eg. scheduled) according to the application's needs [9].

1.2 Contributions

We provided a comprehensive API for creating and attaching socket streams to end-system QoS attributes. The API called QSockets is based on the BSD socket API and follows the same semantics. We argue that using this API for end-system QoS does not impose any substantial overhead in terms of socket creation, destruction, or usage. We also argue that there is no substantial programmatic overhead associated with using our API. In fact we provide functionality for using the API with minimum modifications to the core application.

An interesting characteristic of QSockets is its provision of functionality for dynamically retrieving scheduling and transmission information for use by adaptive multimedia applications. For instance, QSockets-based network feedback can be used to throttle or speed up an application's rate of packet transmission. Similarly, using a socket based classifier for the GNU/Linux traffic controller, we demonstrate how outgoing traffic may be classified on a per-socket basis, thereby allowing us to provide per-stream QoS attributes for scheduling network streams (using the socket-to-socket connection abstraction). Moreover, by associating per-packet QoS attributes, stream scheduling can be made 'smarter', by re-ordering a packet stream according to packet priority, reliability, or deadlines. With this functionality, QSockets approaches the functionality of application-level rate control, but offers the efficiency of system-level packet scheduling.

Interesting aspects of the GNU/Linux QSockets implementation include the following. First, our experience is that kernel-level packet scheduling performs at levels of granularity and predictability that far outpaces previous user-level approaches [9]. This is particularly important for interactive media applications that have smaller packet sizes, like VOIP. Second, it turned out to be awkward to use the existing GNU/Linux QoS infrastructure for the application-level traffic scheduling required by our multimedia codes. This is because the Linux infrastructure was intended for DiffServ networking and was therefore, placed 'under' Linux TCP and UDP protocol stacks. This makes it difficult to implement the application-level framing necessary for 'lossy' media scheduling, where frames that are likely to miss their deadlines upon arrival are discarded at the source instead of the sink, thereby reducing network bandwidth needs. Finally, expanding the selective lossiness implemented by our kernel-level packet scheduler [9], an extension of QSockets being implemented now permits end users to place application-specific packet filters and manipulations into the network path, using safe 'kernel plugins'. Our future work will use QSockets with a variety of applications and in the context of a broader research effort addressing kernel-level support for QoS management for standard Linux kernels, called Q-fabric [4].

1.3 Related work

An effort similar to QSockets can be found in [5]. However this approach was tailored to work only with CBQ. Moreover, this approach did not allow applications to create a socket connection that could be scheduled. In [3] and other papers, researchers have shown that it is useful to have a traffic controller in the kernel, however no significant API has been provided for using this functionality to the application developer. Furthermore, the concentration of such efforts is on traffic controller for nodes in a DiffServ network. Migrating Sockets [10] describes a similar mechanisms to ours, but for end-to-end QoS. However migrating sockets uses custom scheduling in Solaris, whereas we concentrate on using an available scheduling infrastructure in GNU/Linux. Another approach to the same problem is presented in [6], which uses an extension to the existing socket interface for addressing QoS support for multimedia applications. However this approach was designed to work only on Integrated Service networks, which already provide QoS support. The new socket API was only used for packet marking.

1.4 Overview

The remainder of this paper is structured as follows. In Section 2 we give an architectural overview of the current GNU/Linux traffic controller interface, an architectural overview of the QSockets infrastructure and an overview of the QSockets API. In Section 3 we describe how to use the QSockets API for modifying both existing and new applications. Evaluation of the work, including micro-benchmarks and application benchmarks are presented in Section 4. Finally we present our conclusions and discuss future research directions in Section 5.

2 Traffic controller

The GNU/Linux traffic controller was designed to provide Differentiated Services in a GNU/Linux network. The traffic controller handles packets that are being queued for transmission on a network device (Figure 1)

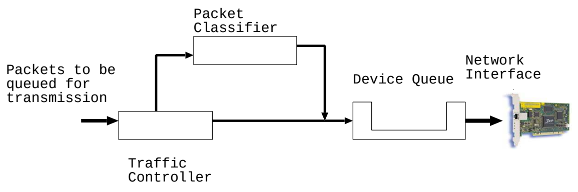

FIGURE 1: *Flow of data packets through the traffic controller*

Once the IP layer has processed the packet it hands the packet over to the device for output queuing. At this point the packet is processed by the traffic controller. The traffic controller classifies the packets and queues them according to rules established previously. The output queuing process can be used to rate limit network traffic, to re-order the output queue and to provide delay/bandwidth guarantees for the network link (Figure 2).

The traffic controller consists of three major components:

- Queuing Disciplines
- Classes
- Filters

For each network device in a system that supports output queuing, there exists a queuing discipline, **Qdisc**, attached to this device. It is this queue to which all packets to be sent on this device are queued for transmission. The Qdiscs supports methods for enqueuing and dequeueing a packet. The Qdisc may be associated with a set of **classes** and **filters** (also called classifiers). The filters are used as a classification mechanism for assigning an outgoing packet to a specific class and each class contains a Qdisc also. In this way a hierarchical set of Qdiscs, classes and filters can be created for maximum flexibility in scheduling of outgoing traffic. A more thorough description of the traffic controller can be found in [1].

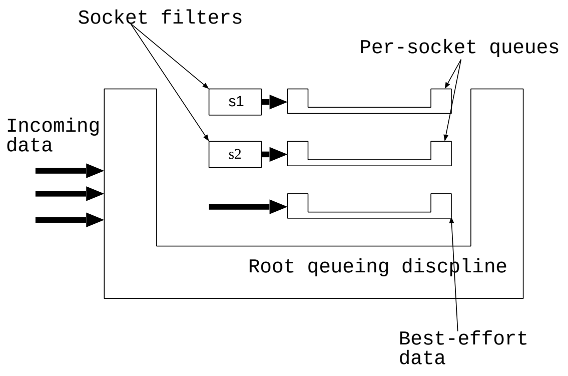

FIGURE 2: *Traffic controller overview*

2.1 Architectural overview

The QSockets architecture works on top of the traffic controller infrastructure. A new system call is added, **sys_qsocketcall** which works as a multiplexer (similar to sys_socketcall). The QSockets architecture (Figure 3) is divided into three functional parts:

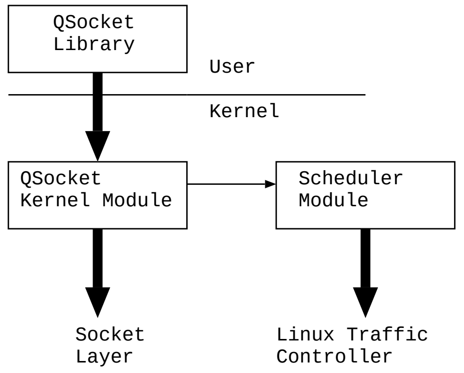

FIGURE 3: *QSockets structure*

1. **QSockets manager**. The QSockets manager is the controller module for QSockets operations. It provides the actual QSockets interface to applications. The manager module provides scheduler independent setup functionality. The bulk of the work however is done in the scheduler modules.

2. **Scheduler module**. The manager module provides a registration functionality for scheduler modules. The scheduler module exposes functionality to the manager module for adding, removing, and changing scheduled streams. Use of the scheduler modules allows QSockets to provide equivalent functionality

for many different scheduler types. This functionality can be further extended to support multiple types of schedulers simultaneously, although currently QSockets is limited to providing support for a single scheduling discipline in a system. Each scheduler module defines a set interface for adding a stream, for removing a stream, for changing a stream, and for initializing itself as the root Qdisc.

3. **Socket classifier**. We have provided a socket classifier which allows us to classify outgoing traffic using the associated socket as a classification attribute. The socket classifier allows an application to use the already available socket abstraction for identifying a data flow. The classifier works on the inode number of the socket descriptor (although this is hidden from the application). Using the natural socket abstraction allows an application developer to add advanced features into the application for differentiating flows and smarter scheduling. For example a video streaming application can stream multiple flows to the same client at different rates. In the presence of an overloaded network the application can adapt by reducing the frame rate of a low priority flow. Although, as stated earlier, QSockets does not necessarily allow end-to-end QoS support, it can be used for service differentiation at the end-host, resulting in similar effects.

2.2 API overview

The QSockets API follows the BSD Socket API closely, following the same functional semantics, but new functionality is added through further extensions.

2.2.1 Socket creation

Sockets are created using the **qsocket** call. Apart from the domain, type and protocol an additional parameter is specified in **struct qos_params**. The structure contains information about which device to attach the classifier to, the scheduler type and the scheduling attributes. At socket creation a socket classifier is also created which classifies outgoing packets according to the associated socket inode.

2.2.2 Changing scheduler attributes

One of the advantages of using a simple API for specifying QoS parameters is that the scheduling attributes can be modified with changing application requirements. In QSockets the parameters are modified with the **qchange** call. The qchange call takes the file descriptor of the socket and the new QoS parameters in a qos_params structure.

2.2.3 Passing message specific parameters to the scheduler

One of the major contributions of this work is the ability to let the application specify per-message scheduling information. This scheduling information can be used for making smart scheduling decisions. For example, a video streaming application can specify a frame boundary for transmitted video frames. The scheduler can use this information to schedule the entire frame as a group. In the case of a lossy packet scheduler, the scheduler can use this information to drop entire frames instead of pieces of a frame. An application can also associate more information with each buffer. A drop priority can inform a scheduler not to drop certain messages (for instance the I-frames in an MPEG video stream). Also a deadline attribute can be used for creating per-message deadlines.
The calls **qsend** and **qsendto** are used for passing packet specific flags to the scheduler. The calls mirror the socket API **send** and **sendto** calls, but an additional parameter is passed containing a set of packet flags.

2.2.4 Retrieving scheduling information

A scheduler can provide the application with very detailed information about the transmission rate, the queuing delays, the packet drop rates as well as various other scheduler specific information. These attributes can be retrieved with the **qinfo** call. Apart from scheduler information, information from the network stack can also be retrieved for even more flexible adaptation. Such information includes buffer sizes, receive windows, etc.

3 API usage

In this section we describe common usage scenarios for QSockets with code excerpts. For the purpose of our examples we use the DWCS queueing discipline [8], ported to Linux kernel version 2.4.17. DWCS provides soft real time guarantees for continuous media streams, and also allows the developer to specify a loss tolerance in the presence of transient overload situations. DWCS supports three parameters, *ipg*, *oln* and *old*. The stream rate is defined by *ipg* and the loss tolerance by *old* and *oln*. A lossy stream is allowed to lose *oln* packets out of every *old* packets queued. DWCS also supports non-deadline constrained streams by providing a best-effort FIFO queue. We also added functionality to DWCS to support per-packet attributes such as frame boundaries, packet deadlines, and drop priorities. Using these attributes we can use DWCS as an EDF scheduler by specifying packet deadlines. Although DWCS is the only scheduler currently supported we plan to add support for more schedulers (such as SFQ and CBQ) in the future.

3.1 New application

When developing a new application, QSockets can be used for providing additional functionality such as adaptation and rate control without the overhead of building a new scheduling discipline.

3.1.1 Initialization

The application developer must use the **qsockinit** call to initialize the root queueing discipline on a device. This is required because of the current limitation within the QSockets architecture which allows the system to use only a single type of queueing discipline system wide. Since it effects the entire system, **qsockinit** is a privileged call and requires root access.

```
struct qos_params params;
int err = 0;

strncpy(params.dev, "eth0", IFNAMSIZ);
strncpy(params.name, "dwcs", IFNAMSIZ);

params.attributes = NULL;

err = qsockinit(&params);
```

This needs only be done once for all subsequent usages of QSockets. An error code indicating the condition is returned to indicate success or failure. An application must check the return value to determine whether the root queueing discipline has been installed successfully. If an incompatible queueing discipline is already instantiated, an error code is returned.

3.1.2 Socket creation

The following code snippet describes the creation of a scheduled socket end point.

```
struct qos_params params;
struct q_dwcs_attr attrs;
int fd;

/*set device name*/
strncpy(params.dev, "eth0", IFNAMSIZ);

/*set scheduler name */
strncpy(params.name, "dwcs", IFNAMSIZ);

/*set scheduling attributes*/
attrs.oln = 0;
attrs.old = 10;
attrs.ipg = 10;
attrs.stream_flags = DWCS_NON_DROPPABLE;

params.attributes = &attrs;

/*create scheduled socket*/
fd=qsocket(domain,type,protocol,&params);
```

On success a new socket identifier is returned to the application that can be used like another socket descriptor. Additional functionality can be exploited by the application, however, by using the **qsend** and **qsendto** calls for sending data on this socket end point. The **qinfo** call can also be used for retrieving information from the scheduler. The **qchange** call can also be used to modify the scheduling parameters of an existing application.

3.1.3 Passing message specific parameters

To add this functionality to an application the **qsendto** call must be used instead of the normal **sendto** socket call. We pass a **packet_flags** structure in with the **qsendto** call. In the example below we specify a non-droppable flag for the message as well as a packet specific deadline.

```
struct packet_flags pflags;
/*mark the packet as non-droppable*/
pflags.drop_flag = DWCS_NON_DROPPABLE;
/*attach a packet deadline to this message*/
pflags.deadline = deadline

/*call qsendto and send the
packet information down to the scheduler*/
if(qsendto(sockfd, &next_frame,
sizeof(next_frame), 0,
(struct sockaddr*)&name,
size, &pflags)< 0)
perror("qsendto");
```

3.2 Adding to an existing application

An existing application can be easily modified to use QSockets without the addition of many new lines of code. Instead of creating a new socket and associating a scheduler with it, an existing socket can be associated with a new scheduler using the **qattach** call. Message specific parameters can also be passed down to the scheduler using **qsendto** in the way described above. When **qattach** is called it does not create a new socket to associate with a scheduler, but rather it uses and already existing socket. This allows the application developer to separate the socket creation functionality from the QSockets functionality.

```
struct qos_params params;
struct q_dwcs_attr attrs;
/*fd is the socket descriptor*/

/*set device name*/
strncpy(params.dev, "eth0", IFNAMSIZ);
/*set scheduler name */
strncpy(params.name, "dwcs", IFNAMSIZ);

/*set scheduling attributes*/
attrs.oln = 0;
attrs.old = 10;
attrs.ipg = 10;
attrs.stream_flags = DWCS_NON_DROPPABLE;

params.attributes = &attrs;

/*create scheduled socket*/
qattach(fd, &params);
```

4 Evaluation

Evaluation of an API requires consideration of two things. The API should be able to provide its functionality without significant overhead. And secondly the API's additional functionality should be useful for applications.

4.1 Micro Benchmarks

QSockets adds the following new calls: **qsockinit**, **qsocket**, **qchange**, **qinfo**, **qsendto**, **qclose** and **qattach**. We compare the execution time of each call with the execution time of the equivalent socket calls. In the case of qsockinit, qinfo, qchange and qattach there are no equivalent socket calls, but we show that the time taken for these calls is minuscule. The results displayed are the average of 10 runs. All times are in seconds. The experiments were performed on a dual P2-400 with a 10Mbps network interface.

Call	Sockets	QSockets
qsockinit	N/A	0.046406
qsocket	0.000871	0.000944
qattach	N/A	0.000085
qchange	N/A	0.000097
qinfo	N/A	0.000020
qsendto	0.001272	0.001475
qattach	N/A	0.000049
qclose	0.000063	0.000089

TABLE 1: *Execution time of API calls*

As can be seen from table 1 the QSockets call overhead is very small. The only call which has a significant execution overhead is **qsockinit** but this is called only once to initialize the QSockets interface. Since this is a privileged call it should not be placed in an application, but rather called by the system at startup. The call **qsocket** which is the replacement for the socket creation call has a very small overhead. The most frequently used call (**qsendto**) only has an overhead of 8.3% higher than the regular sendto call. We can also see that the new calls qinfo, qchange and qattach have very low execution times.
The reason for the high overhead of **qsockinit** is the creation of a scheduling discipline. However the overhead for **qsendto** is harder to explain. Compared to a normal sendto call, there is additional work that happens during the execution of the qsendto call. The additional work includes processing of the socket classifier, the copying of the packet specific flags, as well the enqueueing of the packet into the DWCS queues. We feel that the 15.9% overhead is justified by additional functionality provided by the new call.

4.2 Application benchmarks

We use two separate classes of applications that require delay guarantees from the end-host. A video server needs to transmit generated frames with a certain rate (a rate based deadline). The server must transmit the data in a timely manner or the client receives a low frame rate and high levels of jitter. We show that by using a rate based scheduler (DWCS) in a video application we can achieve a more consistent frame rate and lower jitter. We also consider a data acquisition application example where a sensor provides data to a remote client. The sensor data is deadline constrained, i.e if the data is outdated it is useless. We demonstrate that by attaching the deadline to the data we can obtain significantly higher number of valid data frames. The applications used simulated traffic which we generated random data.. The video server generated a frame with a random type and transmitted that over the network. The data acquisition application generated both the data and the deadline randomly and transmitted it to the client. In both cases the client recorded values at 5 second intervals.

4.2.1 Streaming video server

A video server must be able to preserve its guaranteed QoS level to the clients at all times, even in the presence of significant network perturbation. At the level of the end-host it requires being able to transmit data at a specific rate with low variations. To test this we developed a simulated server which generated frames randomly.

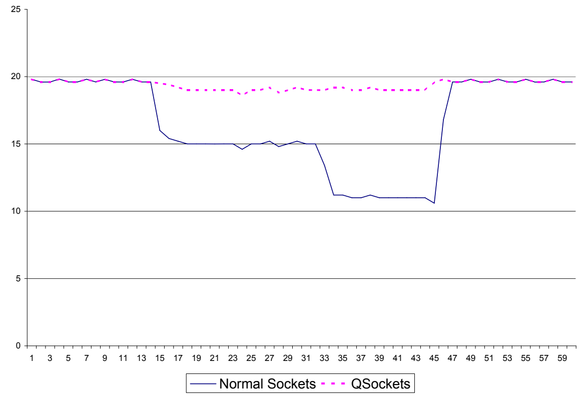

FIGURE 4: *Performance of QoS managed stream compared to unmanaged stream*

In Figure 4 we ran the video server twice, using the QSocket API and without the QSocket API, transmitting a 20fps stream. We also created two network perturbations (at times 15 and 30). The perturbation was created by started a large ftp transfer on the host machines. As can be seen the QSocket enabled video server was able to maintain a frame rate close to the ideal frame-rate of 20fps (frames per second). Without QSocket the frame-rate dropped considerably when the perturbation started. The small decrease in the frame rate occurs because some frames are being dropped by DWCS. In such a situation we preferred that only **P** and **B** type frames are dropped, since they have a lower impact on the quality of the MPEG video stream. We achieved this by marking the **I** frames as non-droppable.

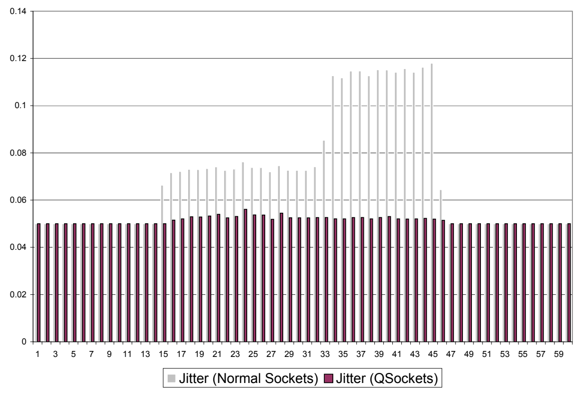

FIGURE 5: *Jitter in a QoS managed stream compared to jitter in an unmanaged stream*

Comparing the jitter in the two streams we see that the QSocket enabled video server is able to provide a much lower jitter rate. As can be seen in Figure 5 the time between consecutive frames never increases significantly even when the perturbations start.

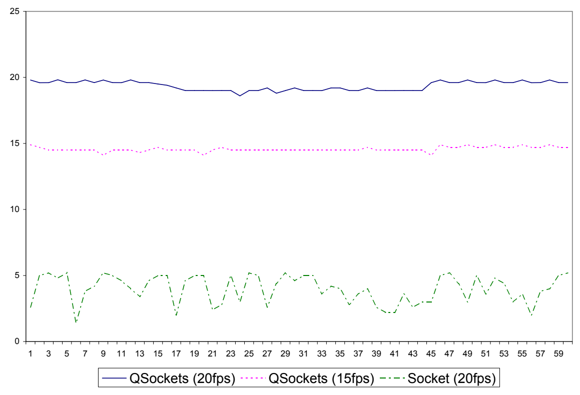

FIGURE 6: *Running video streams at different rate over QSocket*

Finally we show that when the network is overloaded multiple video streams can be sent with the required QoS level, but unmanaged streams suffer as a result. In Figure 6 we see that two managed video streams at 20fps and 15fps respectively achieve their required QoS but an unmanaged stream at 20fps is transmitted at a much lower rate. However it should be noted that there is no starvation of the best effort stream since most video streams are not work conserving.

4.2.2 Remote data acquisition application

The remote data acquisition application is comprised of a remote sensor and a client which acquires data from this client. The sensor sends out data to the client aperiodically and hence a rate based scheduler cannot be used. Instead we use an earliest deadline first scheduler (EDF). The deadlines for each message are indicated by the message parameters. We record the number of missed deadlines at the client side (where the data is too late to be useful) as well as the number of frames dropped. We use two versions of the EDF scheduler (which is based on DWCS), viz. a lossy version and a lossless version. The lossless version tries to preferentially send out data over the network. In the case of a late packet, the data will still be sent, but best effort flows cannot cause significant delays in the deadline constrained streams. The lossy version of the scheduler drops late packets, since it is assumed that the late data is useless. However by dropping late packets the system is able to send packets with achievable deadlines in time. To create the effect of network traffic a perturbation was started at time 30. Similar to the earlier experiments this perturbation was created by a starting a large FTP transfer on the host machine.

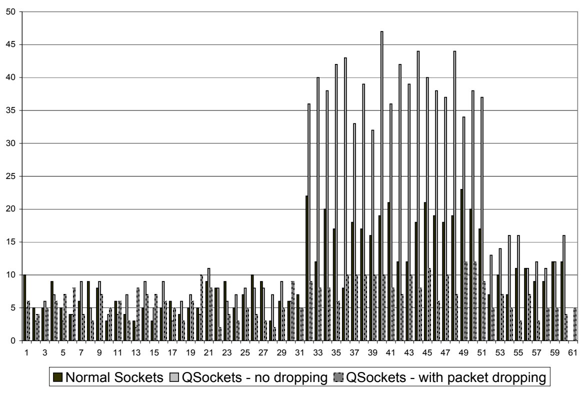

FIGURE 7: *Missed deadlines for every 5 second interval in a remote data acquisition application*

In Figure 7 we see that the number of late or lost data frames are much lower when using QSocket. Furthermore by dropping late frames we achieve very good results. Looking at the breakdown of late or lost frame per-interval we see that after the perturbation is started the non-lossy and the non-QSocket flows both have significant increases in the number of late packets. However the lossy flow suffers from no late packets and is able to drop fewer data frames then the other two flows. This is a significant reduction in the number of deadlines missed by the scheduler.

4.3 Results discussion

As can be seen from the experiments QSockets is a useful API for developing scheduled applications on top of the standard Linux traffic controller interface. Using some of the additional functionality within QSockets we can develop applications that are better able to achieve their desired levels of QoS. From the results in Figure 7 and 6 we see that it is possible to obtain even better levels of QoS by using a scheme for packet dropping. This gives the application developer more flexibility in dealing with situations where the network cannot handle the generated traffic at the desired level of QoS.

5 Conclusion and future work

We presented QSockets, an API for end-host QoS support in GNU/Linux, using the existing functionality of the traffic controller, instead of creating our own. QSockets also extends the functionality of the traffic controller by allowing the developer to pass in a per-packet attribute with each message to be used in making a smarter scheduling decision. We showed that QSockets provides significant advantages to continuous media and remote sensor data applications, and provides these advantages at a low overhead cost. In the future we would like to add additional schedulers to the QSockets framework, including CBQ and SFQ. This will allow QSockets to be used for more general purpose applications which require bandwidth guarantees. We would also like to add further functionality into the traffic controller infrastructure for finer grained traffic classification using application specified, dynamically compiled filters.
We also have plans to compare the performance overheads of positioning the scheduler at a level above the protocol stack, where additional application specific information can be provided to the scheduler. During the deployment of QSockets we realized that because of the positioning of the scheduler below the protocol stack, the traffic controller cannot make traffic shaping decisions on the basis of the message content. We want to compare the costs and the benefits of using an alternate implementation of the traffic controller.

References

[1] W. Almesberger. Linux network traffic control — implementation overview.

[2] W. Almesberger, J. Salim, A. Kuznetsov, and D. Knuth. Differentiated services on linux, 1999.

[3] M. Bechler, H. Ritter, G. Schafer, and J. Schiller. Traffic shaping in end systems attached to qos-supporting networks. In *Computers and Communications, 2001. Proceedings.*

Sixth IEEE Symposium on , Vol., 2001, pages 296– 301.

[4] C. Poellabauer, H. Abbasi, and K. Schwan. Cooperative run-time management of adaptive applications and distributed resources. In *Proceedings of the 10th ACM Multimedia Conference*, 2002.

[5] G. Vaddi and P. Malipatna. An API for Linux QoS Support.

[6] P. Wang, Y. Yemini, D. Florissi, J. Zinky, and P. Florissi. Experimental QoS Performances of Multimedia Applications. In *Proceedings of IEEE Infocom 2000*.

[7] A. Werner. Linux traffic control — implementation overview. Technical Report SSC/1998/037, EPFL, November 1998. ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz.

[8] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. In *ICMCS, Vol. 2*, pages 87–91, 1999.

[9] R. West, K. Schwan, and C. Poellabauer. Scalable Scheduling Support for Loss and Delay Constrained Media Streams. In *Proc. 5th Real-Time Technology and Applications Symposium*, Vancouver, Canada, 1999.

[10] D. K. Y. Yau and S. S. Lam. Migrating sockets — end system support for networking with quality of service guarantees. *IEEE/ACM Transactions on Networking*, 6(6):700–716, 1998.