A VC-based API for Renegotiable QoS in Wireless ATM Networks*

T.-W. Chen†, P. Krzyzanowski‡, M. R. Lyu, C. Sreenan§ and J. Trotter
Bell Laboratories
Lucent Technologies, Murray Hill, NJ 07974

Abstract

Quality of Service (QoS) support for multimedia applications has been widely discussed in the context of high speed wired networks. As interest increases in wireless ATM networks that extend the connection to a wireless endpoint, the issue of QoS over a wireless link has to be addressed. In this paper, we focus on the provision of QoS at the application level in a wireless environment. Our work includes the design of an application programming interface (API) that allows applications to specify and renegotiate the QoS level during a call; as well as the implementation of such API in a wireless ATM testbed: the SWAN system [1]. Experiments are performed to verify the efficiency of this scheme, and the results reveal quality control for multimedia applications despite changing network conditions.

1 Introduction

This work is motivated by the rising popularity of wireless data networking and the desire for mobile, multimedia communications. Wireless technology has recently begun to take off and it is predicted that the market will reach 600 billion dollars by the year 2010 [5]. Wireless networking is inherently unreliable and various forms of interference result in changing bandwidth availability and low effective bandwidths due to high error rates. These problems are exacerbated as users move around. Behavior of this kind requires a fresh look at how such networks can be used to support applications which demand some degree of predictability. We adopt the approach of ATM, in which QoS is used to form a service contract between applications and the network. We build on that work by recognizing that an unreliable wireless network demands a more dynamic approach to resource usage. Many applications can deal with varying bandwidth availability once provided with sufficient knowledge of the resource climate. Typical examples include audio and video applications which can alter their rate or encoding to match the available bandwidth or deal with different error rates [3, 4]. Our contribution is a QoS scheme which builds on this notion of adaptation by providing explicit renegotiation. This is similar in spirit to the feedback mechanisms for non real-time traffic in ATM, but differs in that we aim to provide feedback right up to the application level, not just to the sending host [6]. Thus we incorporate renegotiation as a key part of our QoS API (application programming interface).

In this paper, we report our work of realizing a QoS scheme described above. The remainder of this paper is organized as follows: Section 2 briefly describes the wireless ATM network used as a testbed for our QoS design, implementation and experiments. Section 3 lays out the design policy of our QoS API and the mechanism to support it. In Section 4 we address the major functions provided in our system, while in Section 5 the experimental results are analyzed. Finally, the conclusion and future plans are described in Section 6.

2 The SWAN Environment

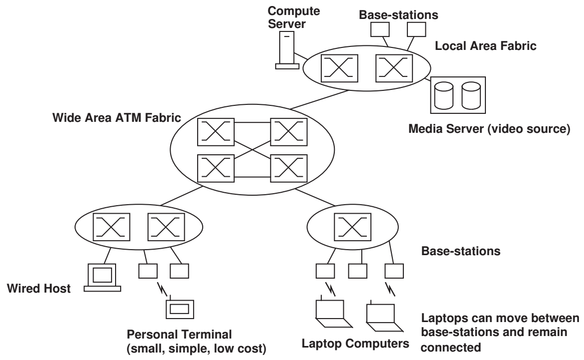

Figure 1: SWAN system architecture

We used the SWAN (Seamless Wireless ATM Network) system [1], which is a testbed for wireless networked computing. It consists of mobile units which are usually laptops, and base stations which are connected to a backbone network. As indicated in Fig. 1, both the base stations and laptops are equipped with a radio interface known as the FAWN (Flexible Adapter for Wireless Networking) [7] card that allows them to communicate with each other wirelessly. Each base station has a range of 100 feet inside a building, providing access to a local area network for mobiles in its vicinity. The mobiles communicate with the base stations and with each other, allowing them to create ad-hoc networks that continually change as the mobiles move around.

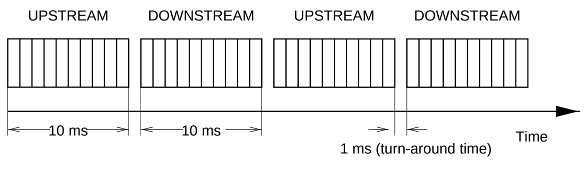

Figure 2: The SWAN TDD scheme

* appeared in IEEE ICUPC'97

† Currently with Computer Science Department, University of California, Los Angeles.

‡ Currently with InVenGen Inc., N.J.

§ Currently with AT&T Labs Research, Florham Park, N.J.

It is worth noting that a TDD (Time Division Duplex) scheme shown in Fig 2 is used to share the bandwidth between the base station and the mobile host. As described in [1, 2], the traffic of each direction alternatively transmits a data burst of 10 ATM cells at a time and then switches to receiving mode for data from the other direction. Due to the overhead introduced by the TDD scheme and the ATM cell structure used in SWAN, the achievable bandwidth is 240 Kbps in each direction.

3 Design of QoS-VC API

The goal for the design of this API is to use existing interfaces and facilities provided by widely accepted operating systems, instead of creating an ad-hoc system or proposing a new, proprietary interface. As SWAN was first developed under Linux, a UNIX-like system, it will be rather easy for a programmer to access data and manipulate QoS on each VC just like an ordinary device under UNIX. Based on this consideration, the QoS-VC API instantiates each ATM Virtual Circuit as a unique character device, and the QoS negotiation is considered on a per-VC basis. In this section, we describe the detail of each API function, the mechanism to realize this QoS-VC scheme and the functions that achieved by this scheme.

3.1 The API

- `int open(char *vc_dev_name, int flags)`
  The `open()` function activates a VC by opening a VC device specified by `vc_dev_name`, which corresponds to an I/O queue in the device driver (described later). Applications may request any VC as long as it is not exclusively used by other entities. `open()` returns the handler for access to the VC. If open() returns -1, it means that it failed to open the VC that the application specified.

- `int write(int vc, char *buff, int nbytes)`
  `write()` attempts to move up to `nbytes` of user data stored in `buff` to a VC outbound buffer for delivery. If `nbytes` is larger than the available space in the outbound queue, `write()` returns the number that have actually been moved. Within the `write()` function, user data is broken into ATM cells whose headers contain the appropriate VC information.

- `int read(int vc, char *buff, int nbytes)`
  The `read()` functions attempts to copy up to `nbytes` from the VC inbound queue to the space user specified space, `*buff`. If `nbytes` is larger than the data in the receiving buffer, `read()` only returns the actual number of bytes that can be read from the queue. `read()` removes the ATM cell header from each cell in a inbound queue then resembles them before they are passed to applications.

- `int ioctl(int vc, int QoS_request, long argument)`
  The `ioctl()` function is the key function for QoS renegotiation. Through `ioctl()`, applications may request that the system to provide the required bandwidth and delay over each VC. The parameters implemented for the QoS negotiation are listed in Table 1. The default values shown in the table indicate that if the application does not make any QoS request, UBR service is provided with zero bandwidth reserved. That is, the best effort service will be provided. `ioctl()` may also be invoked during a call, in case applications need to renegotiate a new QoS contract.

QOS_REQUEST	ARGUMENT	DEFAULT
VC_SERVICE	ABR,CBR,UBR	UBR
VC_MIN_BW	n (Kbps)	0 (Kbps)
VC_PREF_BW	m (Kbps)	0 (Kbps)
VC_MAX_DELAY	d (μsec)	∞ (μsec)

Table 1: ioctl(): parameters

- `int signal(int QoS_SIGNAL, void *qos_handler(int))`
  The UNIX system call `signal()` cannot be classified as a QoS API. However, since it is necessary to have a interface for the network system to inform applications about the change in achievable QoS, `signal()` is selected for this purpose. The `signal()` function allows applications to setup an interrupt handling routine, `qos_handler()`, to handle the change in QoS. After a QoS handling routine is used by the application, the system can inform the application once it detects a change in radio quality such that a QoS contract can no longer be supported. Thereafter, instead of sacrificing the QoS by sending inappropriate traffic into the network, the applications may adjust their output to adapt to the current network condition.

- `int close(int vc)`
  Contrarily to `open()`, `close()` terminates a VC connection and frees up the I/O queue.

3.2 Mechanism

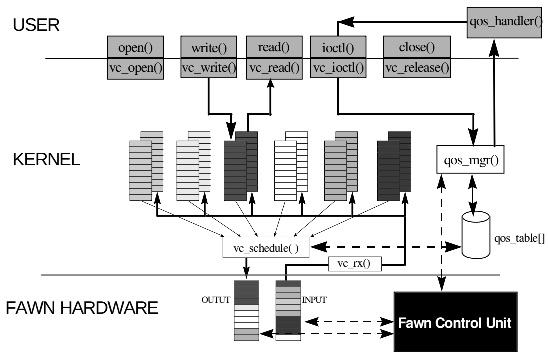

Figure 3: Multiple queue driver

Fig. 3 illustrates the system block diagram and the relationship between user applications and FAWN hardware. The QoS-VC API acts as an interaction interface between user applications and hardware. Applications communicate with the QoS mechanism through this API. A group of priority queues are dynamically allocated in the kernel space, each of these queues is mapped to an individual VC (in Fig. 3, only six queues are activated). Once a VC is opened and its QoS is negotiated through `ioctl()`, which interacts with the QoS manager (`qos_mgr()`) for service and bandwidth specification, the QoS manager translates the requested service type and bandwidth in terms of time slots for carrying data cells in each data burst. This information will be kept in the QoS table (`qos_table()`) that will later be referred by the VC scheduler (`vc_schedule()`). The QoS manager is also responsible for monitoring the overall link quality and providing feedback directly to applications when the requested QoS can not be satisfied or when a better service is available. This feedback is implemented through the UNIX `signal()`, as described in Section 3.1.

The VC scheduler reads packets from those activated queues and sends them to the FAWN hardware for transmission. It serves these multiple queues in a "round-robin" fashion which allows a control of QoS granularity such that one circuit will not dominate the data path with a large chunk of data.

3.3 System Functions

QoS reservation and QoS renegotiation are two major goals we achieve by providing QoS API and VC mechanisms. They are the keys to providing multimedia traffic support in wireless networks. The details of how these are implemented are described below.

3.3.1 QoS Reservation Three types of QoS are considered in the current implementation: service type, bandwidth and delay. The support for each of these service qualities are described below:

- **Service Type**
  Currently, ABR, CBR and UBR are three service types supported by SWAN through the VC scheduler. Based on the QoS information stored in QoS table, the VC scheduler computes the necessary service rate to each queue. A prioritized round-robin scheme allows the VC scheduler in each time frame to provide sufficient service for all CBR queues first, then the ABR, and if there is any bandwidth left it will be used for UBR queues.

  It is the QoS manager's responsibility to make sure that the total of CBR and ABR queues will not require bandwidth that exceeds the link capability. This is described next.

- **Bandwidth Reservation**
  In SWAN TDD system, the allocated bandwidth can be represented in terms of the number of time slots devoted to a connection. For example, a connection granted with one slot in each data burst is served at the bit rate of 24 Kbps (240 Kbps/ 10 slots) in SWAN TDD scheme.

  To reserve bandwidth, the QoS manager converts each bandwidth request into the necessary number of time slots to meet the requirement. If the amount of time slots cannot be allocated, the bandwidth request is rejected.

- **Delay Control**
  In SWAN, as the low bandwidth radio communication only exists between the mobile host and the base station, most of the communication delay is considered as a combination of the queueing delay and the radio transmission delay. Since the radio transmission delay is usually fixed and cannot be avoided, our work focuses on the control of queueing delay.

  The QoS manager determines the queue length based on the requested delay and the requested bandwidth from user applications. In short, the longer of the tolerable delay is, the longer the queue will be. A side effect of this approach is that packets may be lost internally due to the queue overflow. Thus, users may need to consider the trade-off between the packets loss rate and the queueing delay.

3.3.2 QoS Renegotiation As stated in the introduction, QoS renegotiation is very necessary in a wireless network because wireless communication is unreliable and many applications do have the ability to adapt to different network conditions. Thus, our work proposes a feedback mechanism to informs applications of the change in QoS. The applications can benefit from this mechanism by simply setting up handlers such that when they are notified of the change in QoS, they could adjust their data rate based on the latest QoS information.

This proposed feedback mechanism is also implemented in the QoS manager which has direct access to the radio hardware to learn the current status of the radio. For example, when the QoS manager detects the decrease in radio bandwidth, it first reduces the service to the UBR traffic; if such a reduction is not sufficient to guarantee the requested bandwidth for all ABR and CBR traffic, it then reduces the ABR/CBR service rate to its minimum requirement. Finally if the bandwidth is still not sufficient, QoS manager will prorate the assigned bandwidth on all the CBR and ABR circuits and signal the corresponding handlers implemented in the applications to notify the change of QoS. Upon receiving the signal from the QoS manager, the handler in each application can decide whether to accept the newly assigned QoS by, for example, reducing the frame rate or sampling rate etc., or to terminate the connections, or to renegotiate a new QoS with the QoS manager.

4 Experimental Results

Several multimedia applications have been modified to verify the performance of our QoS VC scheme. These applications include a video program that send image frames in a raw data format (bit map) or compressed video using H.263 [4], as well as a file transfer program. All these applications display a significant quality improvement over the original programs that used interface that provided in traditional network protocol (TCP/IP). Using these programs, we performed several experiments and we summarize the results in this section.

4.1 Bandwidth Management

Firstly we study the performance of our QoS VC scheme by using three different traffic sources: one datagram, *d1* and two videos, *v1* and *v2*. *d1* is a packet generator which can generate data as high as the full link capacity (240 Kbps). It is executed in the background during the whole experiment to represent a heavily loaded system. *v1* is an uncompressed video session which transmits an uncompressed video snapshot of about 18K bytes at every 2 seconds (72 Kbps). It starts at time = 15 second and transmits for a total of 35 frames. *V2* is another uncompressed video session similar to video 1, but the frame rate is doubled (1 frames/sec, about 144 Kbps) and it transmits for 100 frames in total. In an ideal QoS-guaranteed environment, both *v1* and *v2* should be granted for 72 Kbps and 144 Kbps, respectively, and *d1* should only use the bandwidth that is left available.

Fig. 4 shows the experimental result using the QoS VC scheme. The lines shown are the bandwidth of each traffic observed at the receiver. In this figure, we can see that constant bandwidth is reserved for *v1* and *v2* which use the CBR service, while *d1* uses all the residual bandwidth since it is designed to use the ABR service and will thus be granted for the residual bandwidth in our experiment. The changes of the bandwidth used by the ABR *d1* is exactly what we expected in a QoS-guaranteed environment.

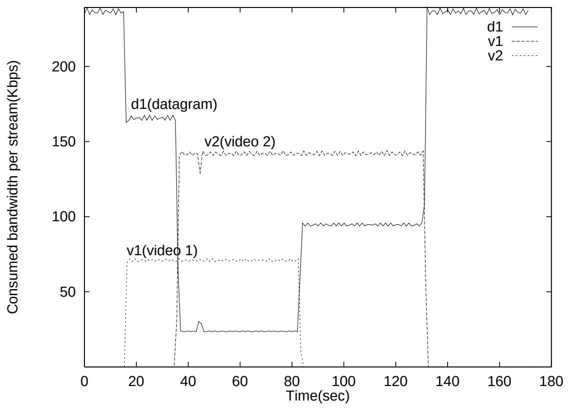

Figure 4: Received bandwidth using VC-QoS

As a comparison, we repeated the same experiment using UDP which was also developed in [1] and doesn't provide any QoS guarantee. The result is shown in Fig. 5. In this figure, we observe that due to the lack of a QoS mechanism, the amount of bandwidth that a connection can utilize is related to how aggressive the traffic source is. As *v1* generates data at 72 Kbps, it is less "aggressive" than *d1*. Therefore between time = 15 and 90 seconds, the quality of *v1* suffers by having to compete with *d1*. Note *v2* is more aggressive (144 Kbps) than *v1*, thus between the 35th sec. and the 90th sec., the observed bandwidth shows that all these three sessions get about 1/3 of the bandwidth (*v1* is in fact slightly less than the other two). When *v1* stops at the 90th second, video 2 and *d1* both get half of the bandwidth. In this experiment, none of the video sessions get the bandwidth they require, and both of them suffer from fluctuations in bandwidth that they get. Note that the fluctuation between time = 15 sec. and 35 sec. is more significant than that between time = 90 sec. and 150 sec. This is because *v1* transmits video frame slower (0.5 frame/sec) and tends to fall behind the competition with *d1*. However *v2* transmits video in a faster pace (1 frame/sec), so it can share the bandwidth with *d1* more competitively. Moreover, the decrease in observed maximum bandwidth in this experiment (220 Kbps) comparing with the previous one (240 Kbps) in QoS VC scheme is due to the overhead in UDP/IP headers.

It is observed that since the bandwidth provided to the two

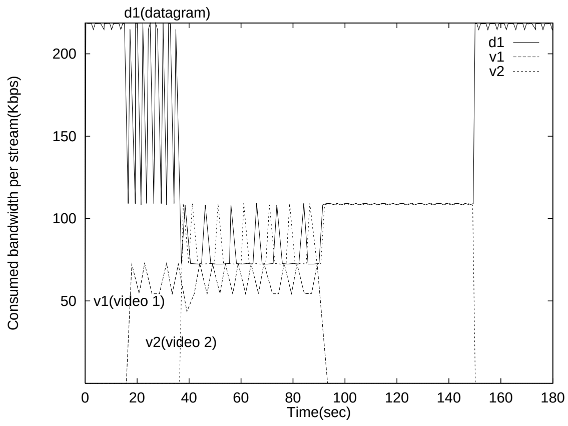

Figure 5: Received bandwidth using UDP

video sessions are less than expected, the durations for both video sessions in Fig 5 are longer than those in Fig 4. This is caused by the delay of the lower, fluctuating bandwidth obtained by both sessions.

4.2 Delay Jitter Control

To study the impact of QoS on a multimedia session, we examine the distribution of inter-frame delay of a video session. The traffic under study is also an uncompressed video source, which transmits a bitmap video frame of 19K bytes every 1.2 seconds for a total of 500 frames. We measure the inter arrival time between each frame at the receiver end. The results are depicted in Fig. 6 through Fig. 9 and summarized in Table 2. The background traffic is the datagram program similar to what we used in the previous experiments, however, it generates data at the rate of 110 Kbps such that the total bandwidth is within the given link capacity. The results are as expected: under the QoS VC scheme, the delay jitter is well controlled even in the presence of the existence of the background traffic. On the other hand, using the UDP protocol experiences a completely different result. Its video source has a decent delay variance of 0.064. when there is no background traffic; but when it comes to the existence of background traffic, the delay variance doubles and becomes unpredictable. We see the UDP (with no QoS) posts severe delay jitter problems in heavy traffic situation. Our QoS VC mechanism, on the other hand, is stable and efficient. Regardless to the network total load, it only introduces 2.6% more in delay overhead compared with UDP scheme without any load.

Protocol	QoS-VC		UDP(no QoS)	
Background traffic	Yes	No	Yes	No
Mean delay (sec)	1.224	1.224	1.474	1.193
Variance	0.056	0.056	0.112	0.064

Table 2: Summary of Delay Jitter

5 Conclusions and Future Work

In this paper, we propose a kernel level mechanism that provides applications with a direct QoS feedback from the radio link.

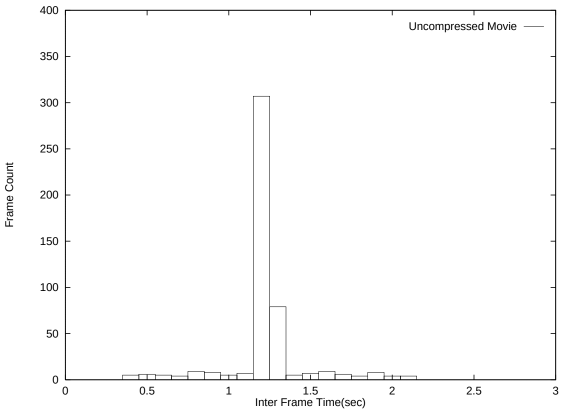

Figure 6: Delay distribution in QoS-VC without background load

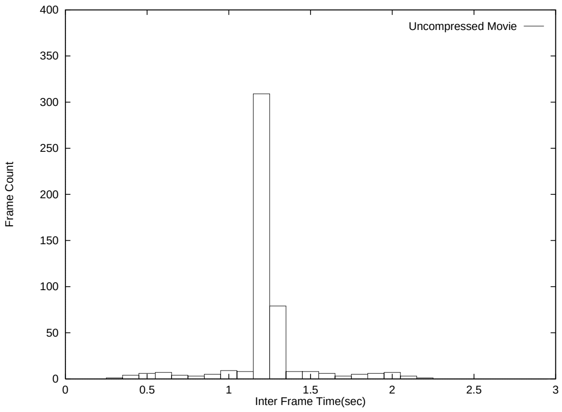

Figure 7: Delay distribution in QoS-VC with background load

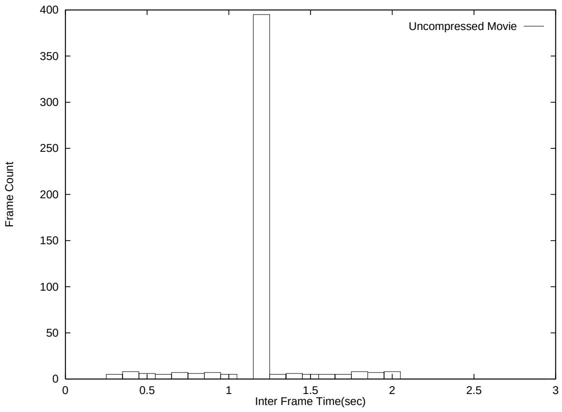

Figure 8: Delay distribution in UDP without background load

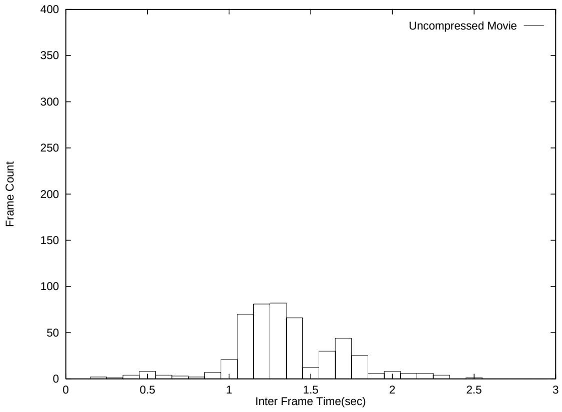

Figure 9: Delay distribition in UDP with background load

This mechanism negotiates and guarantees the service level when radio link is stable. When the link quality changes and the requested QoS cannot be satisfied, it informs applications directly through a feedback signal. Therefore, instead of an inadequate performance due to insufficient and varying link quality, the traffic source has a chance to adjust and best utilize the changing link quality without dropping the connection. We are now investigating the support for the interfacing with QoS signaling from wired network such as wired ATM. The application of our proposed QoS scheme to other wireless systems based on different MAC protocols like IEEE 802.11 is also being evaluated.

References

[1] P. Agrawal et al., “SWAN: A Mobile Multimedia Wireless Network,” in *IEEE Personal Communications*, Apr. 1996. pp. 18-33.

[2] T.-W. Chen et al.,“Renegotiable Quality of Service – A New Scheme for Fault Tolerance in Wireless Networks,” in *FTCS-27*, Jun. 1997.

[3] R. Frederick, “Experiences with Real-time Software Video Compression” at ftp://ftp.parc.xerox.com/pub/net-research/nv-paper.ps, Jul. 1994.

[4] ITU-T “Recommendation H.263: Video Coding For Low Bitrate Communication,” available at http://www.fou.telenor.no/brukere/DVC/h263_wht/.

[5] R. Schneiderman, *Wireless Personal Communications*, IEEE Press, 1994.

[6] C.J. Sreenan and P.P. Mishra, “Equus: A QoS Manager for Distributed Applications,” in *Distributed Platforms*, Editors A. Schill et al. Publishers Chapman & Hall,1996, pp 496-509.

[7] J. Trotter and M. Cravatts, ”A Wireless Adapter Architecture for Mobile Computing,” in *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Comp.*, Apr. 1994, pp. 25-31.