The Design and Implementation of a High-Speed User-Space Transport Protocol

Glenford Mapp and Steve Pope
Olivetti & Oracle Research Laboratory
24a Trumpington Street
Cambridge UK
CB2 1QA

Andy Hopper
Computer Laboratory
University of Cambridge
New Museum Site
Pembroke Street
Cambridge UK
CB2 3QG

Abstract

Audio and video are fast becoming an integral part of new computing environments. These media have transport requirements which differ from the normal bursty computer traffic. There is therefore a need to explore transport protocols that can provide different qualities of service. User-space implementations of such protocols are particularly interesting because they can be easily tested and refined. This paper discusses the design and implementation of a high-speed user-space protocol called **A1**. Its preliminary performance in an ATM environment is presented and compared with an efficient kernel implementation of TCP/IP.

Motivation

The Olivetti & Olivetti Research Laboratory (ORL) has a long tradition of research into high speed networks and multimedia applications [**Hopper90**], [**Wray94**]. These applications have different transport requirements which can be met by a transport protocol which offers various qualities of service. The Xpress Transport Protocol (XTP) [**Strayer92**] can provide such functionality and preliminary performance of the SandiaXTP implementation [**Strayer94**] was reported in [**Mapp95**].

SandiaXTP uses a daemon process which implements the protocol and it was felt that its performance could be improved by a user-space implementation. However this effort highlighted certain issues that were more fundamental to the design of transport protocols which could only be explored with the development of a new transport protocol. This paper discusses the design and implementation of a user-space transport protocol called **A1**.

Observations on Transport Protocols

In transport protocol processing the receiver generally has more work to do than the sender. Thus reducing receiver processing should be a main goal of protocol design. Two obvious ways of addressing this issue is to make the receiver as dumb as possible and to provide the receiver with as much information as possible, even though this could increase the processing time of the sender.

As an illustration of the latter approach, consider the area of retransmission. Let us suppose that we want to tell the receiver that a packet has been retransmitted by setting a bit in its header. This obviously requires the checksum of the packet to be recomputed the first time it is retransmitted. However, if retransmission rates are very low and checksumming is fast, this overhead is worthwhile if it improves the performance of the receiver.

Latency is a critical issue for multimedia applications. Many transport protocols introduce added latency by too readily buffering data. A good example is the situation where the flow control window is full. Most protocols will buffer some amount of data until the window is opened again. However, for multimedia applications that would like the receiver to have the most current data as soon as possible, it may be preferable to throw away the data and wait for the next item of data from the hardware. In these situations it would be better for the transport protocol to signal to an application that its flow control window is full and let the application take appropriate action.

When designing transport protocols, it is essential to take into account the interaction between transport protocols and operating systems. This is particularly true in user-space protocol design. When the process is finally scheduled there may be several packets waiting to be processed. If the protocol supports bidirectional communications, these packets may consist of data sent by the remote end as well as acknowledgements(ACKs) and/or negative acknowledgements(NACKs) packets indicating that data packets sent by the local end have been received correctly at the remote end or some packets have been corrupted or lost. Most transport protocols would simply process these packets sequentially. This may not be the optimal approach.

If the local end is transmitting multimedia data and would like to maintain low jitter, then it would be beneficial to process the control packets such as ACKs and NACKs first. In the case of ACKs, this may open the flow control window allowing applications transmit data immediately. If data is being sent reliably then valuable buffer space held by already transmitted packets waiting for acknowledgements to arrive can be freed immediately. In the case of NACKs, the lost or corrupted data could be transmitted immediately, improving overall performance. Such priority treatment can also be extended to retransmitted packets if they are clearly identified.

Existing transport protocols do not consider the issues outlined above.

Protocol Design Decisions for A1

In this section some fundamental issues in the design of a transport protocol are discussed and the design decisions taken for **A1** are presented.

- **Block vs Byte Processing:** Protocols like TCP and XTP are implemented as a stream of bytes in which each byte in the stream can be identified as a individual entity. Whilst this may be good for a protocol whose main purpose is to provide a reliable bytestream, a block-based approach may be more appropriate where the transport protocol provides differing qualities of service, because for many multimedia applications receiving a corrupted set of bytes is usually of little consequence to their overall operation. Since checksums in most transport protocols are done on blocks of data and not on individual bytes, if a block is corrupted during reliable transfer the entire block will have to be retransmitted anyway. In addition we can easily construct a reliable bytestream from a block-based protocol. **A1** is implemented using blocks rather than bytes.

- **Selected Retransmission:** Selective retransmission was adopted as the mechanism for data recovery rather than using the *go-back-n* approach used in TCP. This is because it was felt that only packets that were actually lost or corrupted should be retransmitted. Multimedia applications send large amounts of data and retransmission can easily overload switches and routers.

- **Explicit Round-Trip Time Measurements:** The round-trip time, the time taken to send data and get a response, should be explicitly measured on a periodic basis. This is an important parameter as it controls how quickly the receiver can respond to certain requests from the sender. In this protocol we assign a special packet called a SYNC packet to periodically measure round-trip time on a transport connection.

- **End-to-End Flow Control:** For multimedia applications which can require large amounts of buffering, end-to-end flow control is required to prevent buffer overload at the receiver. This means that the windowing flow control mechanisms at the transport level must take into account packets that have been processed by the protocol but have not been consumed by the receiver.

- **Support for Multicast:** The ability to multicast data to several endpoints is becoming a much desired feature in transport protocols. Where possible, transport protocols should take advantage of multicast facilities offered by the underlying network layer. However it was also felt that there should be support for multicast at the transport level where data is simply replicated on individual connections which make up the multicast group.

QoS Support in A1

A1 provides a number of QoS parameters including the average and burst rates for sending and receiving, the priority of the connection and whether checksumming and/or retransmission is required. An application can also specify the size of the largest message it will send and the size of the largest message it is willing to receive as well as the maximum number of messages that it would like to have outstanding at any time. This is used to calculate the size of the flow control window.

The QoS support that a transport layer can offer depends to a large extent on the characteristics of the underlying networks. It is essential therefore to be able to obtain the QoS being offered by the network and to be able to translate these into transport parameters and vice-versa. Presently some QoS parameters are defined for ATM and are slowly being implemented. RSVP [**Zhang93**] is also being deployed by the IP community and though it allows dynamic changes to the quality of service, the actual time taken to effect these changes may be too slow for multimedia applications.

In view of these uncertainties, we have tried to minimize the need to involve the network layer when dynamically changing the QoS on a stream. When the connection is being set up the maximum QoS parameters that the endpoints will use are calculated. These parameters are submitted to the underlying network. Once these parameters have been agreed then the QoS may be dynamically changed at the transport layer without going to the network once the change is within the agreed bounds. If the maximum parameters are exceeded the network layer must be explicitly consulted or the transport layer could close the connection and reopen a new connection with a higher quality of service.

To keep the protocol simple, it was decided not to support end-to-end QoS negotiation at connection setup. The receiver can change the maximum QoS parameters in a limited way but must abandon the connection if the QoS parameters are very different between the sender and the receiver.

The User Interface

The user interface provides applications with a socket-like interface to **A1**. An application first asks for a socket and then binds the socket to an address and a maximum quality of service. Addresses in A1 are 64-bit entities and at the moment comprise a 32-bit host number and a 32-bit port number. An application can connect to an endpoint by invoking the **add_sink** call specifying the address to which it wants to be connected. When adding the first sink, applications can specify the initial quality of service that will be used which might be different from the maximum QoS specified in the bind call. One can multicast a connection by simply invoking another **add_sink** call. Applications can then send data or receive data. When sending data applications can specify whether they wish to take their own action when the flow control window is full or whether they would like to be blocked until the window is opened again. Applications can dynamically change the quality of service by issuing a **_setqos** call. When the applications are finished they close the socket which then shuts down the connection.

Most user interface calls have a 32-bit status field which is used to indicate several pieces of information to the application including the start and end of messages, whether the other side has closed, whether the network has aborted the connection and whether the application has exceeded the agreed flow control window size and/or rate. There is also a *QoS pending* indicator which indicates that a QoS change is about to occur and there is another indicator which informs the application when the change has actually occurred.

Implementation

Every connection in the **A1** protocol has an associated thread called a **satisfy** thread which is created when the application adds the first sink or does a listen. Each connection is represented by a **context structure** which contains a pthread mutex and a condition variable to ensure proper interaction between the application and the **satisfy** thread.

The overall architecture for our implementation is given in Figure 1. There is no thread switching involved in sending data. The application thread simply grabs the mutex, does the relevant processing and then invokes the send call in the network interface. For receiving, the **satisfy** thread is notified by the OS-dependent layer when packets have arrived or when a timeout has occurred. If packets have arrived,

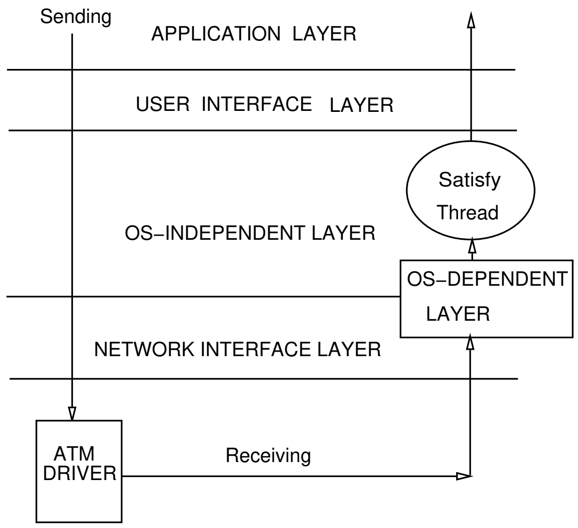

Figure 1: Implementation Architecture

it processes them and unblocks the application if it is waiting to receive data.

The context structure also contains two packet queues for receiving packets. ACKs, NACKs and retransmitted packets are placed on a high priority queue and are processed first. Data packets and SYNCs packet are queued on the other queue and are processed after the packets in the high priority queue have been dealt with.

Buffers are provided by the network interface and are represented by a structure which indicates the actual size of the buffer and where it is located in memory. There is also a pointer to the start of the data of interest within the buffer and its length. Therefore by adjusting these two fields it is possible to point different layers of the protocol stack to their relevant area of processing as packets are moved up and down the stack. There is also a pointer to a free function which is called when all the layers are finished with the buffer. This returns buffers to the relevant buffer queue.

Preliminary Performance

We report on some preliminary performance tests for A1. The tests were done using two Pentium Pros (199 bogomips/256M) connected via an ATML Virata Switch using Efficient Networks, ENI-155 Mbits/s PCI adaptor. Both machines were running Linux 2.0.25 with Werner Almesberger's Linux-ATM release 0.26. For this implementation, 40-byte E164 addressing was used. The first test consists of sending a large amount of data but using block sizes up to 16K in length and measuring the throughput of the system.

Figure 2 shows the results for raw AAL5 SVC, classical TCP/IP or **CLIP** over ATM and A1 with and without retransmission. For the AAL5 measurements, a switched virtual circuit was set up, using the ATM_UBR QoS traffic class. A 1Gbyte burst was sent and the throughput was measured form a 100 Kbyte sample. The plot shows a rather linear rise in throughput to 135 Mbits/s with the knee of the curve appearing at a block size of 1.7Kbyte. For CLIP a 10 Gbyte stream was used with

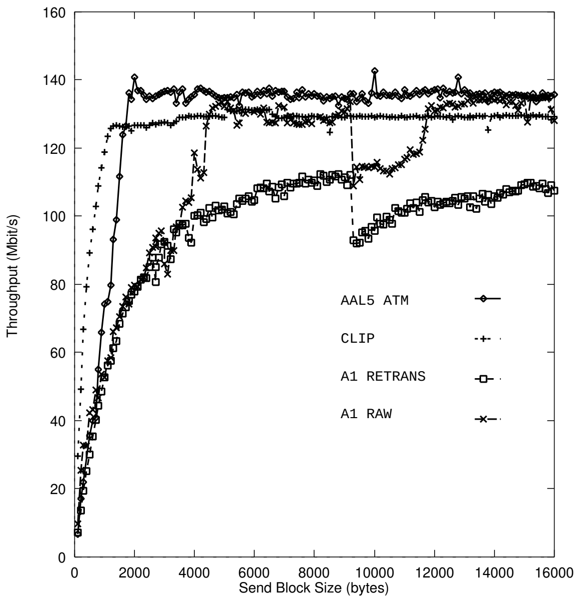

Figure 2: Throughput Measurements

the average throughput measured over the whole stream. The CLIP plot shows the throughput for small block sizes which exceed those for raw ATM because of the effect of TCP batching up small writes. The knee of the throughput curve occurs at 1.2Kbyte blocks and 130 Mbits/s.

The same test was done for A1 with retransmission (A1 RETRANS), and the throughput was measured over a 1Gbyte stream. A 9Kbyte network block size was selected to make a valid comparison with CLIP. A flow control window size of 160 Kbytes was used. The graph slowly climbs to about 111Mbits/s with a fall as the send block size exceeds the network block size. This is due to the increased work of segmenting the payload into two segments as the network block size is passed. CPU utilization was observed to vary from 17% to a maximum of 45% at the network block size.

With A1 without retransmission (A1 RAW), the throughput climbs on a similar curve to A1 RETRANS but continues upward to about 130 Mbits/s before falling once the send block size reaches goes past the network blocksize and then rising again to a level close to the raw ATM throughput. Some instability was observed and the window size had to be varied between 160 - 196 Kbytes.

The second test measured the latency of the AAL5, CLIP and A1 RETRANS. It consisted of measuring half the round trip time of 1000 user level "pings" with varying send block sizes. The results are shown in Figure 3. The graph shows that latency of A1 is greater than TCP and RAW AAL5 for small packets but this decreases as the send block size increases.

Conclusions

The results are very interesting for several reasons. Firstly the results for TCP/IP are impressive especially for small send block sizes. This implementation benefits

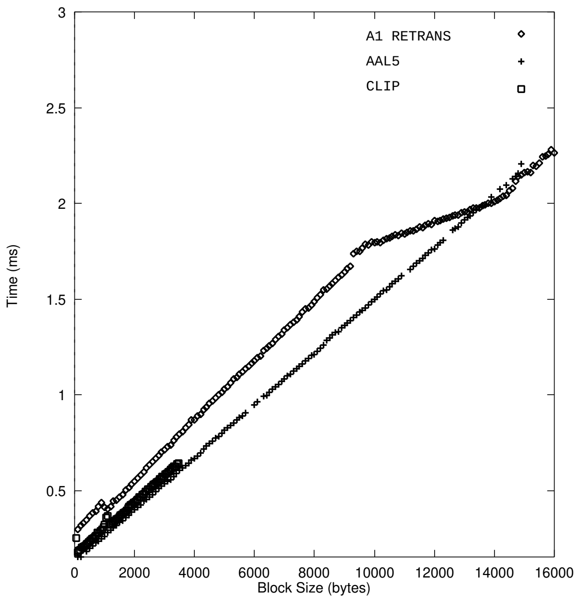

Figure 3: Latency Measurements

from recent extensions to the protocol [**Stevens94**] as well as from running in the Linux kernel. **A1** results are very good because **A1** is running in user-space and yet manages to achieve significant throughput and latencies well below 1 millisecond. The protocol has not been tuned or optimised in any significant way. No assembler routines were used in this implementation. These results clearly show that a user-space protocol is able to achieve significant performance and should help to debunk the idea that tranport protocols must be run in the kernel in order to achieve reasonable performance!

We are investigating the interaction between the **satisfy** thread and the application to reduce the batching of data at very high throughput rates. We are also interested in studying the behaviour of transport protocols on very fast (i.e. 1-100 Gbits/s) links [**Sterbenz95**].

In closing we note that there has been an emphasis on the performance of transport protocols [**Jacobson93**]. This is good but we believe that the issue of functionality will in the future be as important as performance.

Acknowledgements

We would like to thank Steve Hodges, Brendan Murphy, Martin Brown and Andy Harter at ORL for many useful discussions and Tim Strayer for his work on Sandi-aXTP which triggered this effort.

References

[**Hopper90**] A. Hopper. Pandora – an experimental system for multimedia applications. *ACM Operating System Review*, April 1990.

[Jacobson93] V. Jacobson. A High-Performance TCP/IP Implementation. In *Gigabit-per-Second TCP Wksp.* Ctr. for Nat'l Res. Initiatives, March 1993.

[Mapp95] G.E. Mapp. Preliminary Performance Evaluation of SandiaXTP on ATM at ORL. In *PROMS '95, Second Workshop on Protocols for Multimedia Systems*, October 1995.

[Sterbenz95] J.P.G. Sterbenz. Protocols for High Speed Networks: Life After ATM? In *Protocol for High Speed Networks IV*, pages 3–18. Chapman & Hall, 1995.

[Stevens94] W.R Stevens. *TCP/IP, Illustrated, Volume 1.* Addison and Wesley, 1994.

[Strayer92] W.T. Strayer, B.J. Dempsey, and A. C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison and Wesley, 1992.

[Strayer94] T. Strayer, G. Simon, and R. E. Cline Jr. An Object-Oriented Implementation of the Xpress Transfer Protocol. *XTP Forum Research Affiliate Annual Report*, pages 53–66, 1994.

[Wray94] S. Wray, T. Glauert, and A. Hopper. The Medusa Applications Environment. In *International Conference on Multimedia Computing and Systems*, May 1994.

[Zhang93] Zhang, Lixia, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, pages 8–18, September 1993.