Diagnosing a TCP/ATM Performance Problem: A Case Study

Reid van Melle
Carey Williamson
Tim Harrison
Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, SK, CANADA
S7N 5A9
Phone: (306) 966-8656
FAX: (306) 966-4884
Email: {rav124,carey,harrison}@cs.usask.ca

October 2, 1996

Abstract

In May 1996, we obtained an SGI Indy workstation with OC-3 (155 Mbps) ATM connectivity to an experimental wide-area ATM network testbed. Unfortunately, our initial experiments with this network were disappointing: an abysmal end-to-end throughput of 4.3 Mbps for large file transfers between SGI Indy workstations at the University of Saskatchewan and the University of Calgary.

This paper describes our efforts in analyzing and diagnosing this performance problem, using cell-level measurements from the operational ATM network. Surprisingly, the performance problems are not due to send and receive socket buffer size problems, heterogenous ATM equipment, or cell loss within the ATM network. Rather, the poor performance is caused by an overrun problem at the sending workstation for the ftp transfer.

The paper presents the evidence for our diagnosis, including cell-level measurements, netperf tests, and a TCP/ATM simulation model that mimicks the observed network behaviour. The paper concludes with several proposed solutions to the TCP/ATM performance problem.

1 Introduction

In May 1996, we obtained an SGI Indy workstation at the University of Saskatchewan, complete with an OC-3 (155 Mbps) ATM (Asynchronous Transfer Mode) network interface card (NIC). This workstation is used to support collaborative research activities with other researchers on the CANARIE (Canadian Network for the Advancement of Research, Industry, and Education) National Test Network (NTN). This network provides a coast-to-coast experimental ATM network across Canada, with a DS-3 (45 Mbps) national backbone connecting several regional ATM testbeds.

Unfortunately, our initial experiments using this workstation and the NTN were disappointing: an abysmal end-to-end throughput of 4.3 Mbps for large (e.g., 8 Megabyte) file transfers between SGI Indy workstations at the University of Saskatchewan and the University of Calgary (approximately 400 miles apart, via the NTN). This poor performance was surprising, given that the NTN was practically idle during our tests, and given that each workstation had a dedicated Permanent Virtual Channel (PVC) allowing it to use up to 25 Mbps of bandwidth on the NTN backbone. The low throughput results were very repeatable, regardless of which direction the file transfer was attempted.

Several hypotheses immediately came to mind as possible explanations for the poor performance. First, the poor performance could be due to a poor choice of send and receive socket buffer sizes for the communicating TCP's. This phenomenon, often referred to as TCP deadlock, has been well documented in the literature in recent years [1, 3, 11]. However, our SGI Indy workstation was running Irix 5.3 with the TCP high performance extensions [9] enabled (i.e., very large send and receive socket buffer sizes), which ruled out this problem. Second, the poor performance could be due to cell loss in the network. Again, this effect has been well-documented in the literature [5, 12]: even a low ATM cell loss ratio can result in extremely poor end-to-end TCP performance. This cause seemed particularly likely given the heterogenous ATM equipment in our experimental network: Newbridge switches, Fore switches, Cisco routers, Fore NIC's, and HP NIC's. However, a detailed check of switch statistics showed no cell loss in the network. The third hypothesis was that performance was being constrained by the Network File System (NFS), since both workstations had to access a local file server. However, network tests with `netperf`, a software tool for testing end-to-end TCP performance using memory to memory transfers, revealed similarly low throughput.

Since none of these hypotheses were able to explain our observed performance problem, we took another approach entirely: ATM network traffic measurement. Fortunately, through joint research activities, we were able to borrow an ATM test set from TR*Labs* in Winnipeg. This test set provides the capability to monitor and capture complete ATM cell traces at OC-3 speeds from an operational ATM network, in a completely unobtrusive fashion. Traces of the ftp transfer, once properly decoded up the protocol stack from the ATM cell level to the TCP level, were crucial in figuring out what was causing our performance problem.

The purpose of this paper, then, is to describe our efforts in analyzing and diagnosing this TCP/ATM performance problem, using cell-level measurements from the operational ATM network. Surprisingly, the performance problems are not due to send and receive socket buffer size problems, heterogenous ATM equipment, or cell loss within the ATM network. Rather, the problem stems from overrun at the sending workstation for the ftp transfer. Whether this problem occurs in the ATM NIC or the device driver on the workstation is as yet unclear, but the problem definitely exists. In particular, large numbers of TCP sequence numbers are missing from the original transmissions by the workstation, and must be recovered by retransmission after lengthy TCP timeouts. This behaviour, which is very deterministic, accounts for the extremely low ftp throughput.

The rest of the paper is organized as follows. Section 2 provides some background material on the NTN and the environment used for our network traffic measurements. Section 3 presents the analysis of our TCP/ATM measurements, and our hypothesis for the cause of the performance problem. Section 4 presents additional experimental results in support of our hypothesis. These results come from a TCP/ATM simulation model that can recreate the observed TCP behaviour, and from some netperf tests designed to pinpoint the conditions under which the performance problem appears. Finally, Section 5 provides a summary of our observations to date, and suggests several ways to work around the TCP/ATM performance problem that we encountered.

2 Background

2.1 The CANARIE National Test Network

The National Test Network (NTN) is a coast-to-coast experimental ATM network, sponsored in part by the Canadian government CANARIE program. The network, which consists primarily of an east-west backbone network, connects together regional ATM testbed networks from several Canadian provinces. The slowest links in the NTN backbone are DS-3 (45 Mbps). The highest speed links, primarily within some of the regional testbeds, are OC-3 (155 Mbps).

The NTN backbone is currently configured as follows: 10 Mbps are reserved for carrying CA*net Internet traffic, 10 Mbps are reserved for distance education course offerings using Motion JPEG compressed video (or other technologies), and 25 Mbps are for use by Canadian university researchers in high performance computing and high performance networking.

2.2 Workstation Configuration

The workstation at the University of Saskatchewan that is connected to the NTN is an SGI Indy, running Irix 5.3. This workstation, named `joe90.usask.ca`, has a 174 MHz MIPS R4400/4010 CPU/FPU with a 1 MB secondary cache and 64 MB RAM. This workstation also has an OC-3 (155 Mbps) ATM network interface card (NIC). The precise NIC being used is a Fore Systems GIA-200e, with firmware version 4.0.0, and ForeThought driver version 4.0.1 (1.14).

In order to avoid exceeding the 25 Mbps bandwidth restriction, the ATM NIC on the SGI Indy is configured for a maximum output of 25 Mbps. This translates to a peak cell rate of approximately 58,962 cells per second and means that back-to-back cells must be separated by at least 17 μs of delay.

The SGI Indy workstation at the University of Calgary is similarly configured. Though the WAN traffic travels over a DS-3 line for most of its journey, all ATM cell traffic is delivered locally in both Saskatoon and Calgary over an OC-3 line at 155 Mbps. The round-trip time between the two workstations, over the NTN, is 14 milliseconds.

2.3 ATM Measurement Device

The network traffic measurements described in this paper were made using a GN Nettest NavTel InterWatch 95000 ATM test set (referred to as the IW95000 in the rest of the document). This test set was borrowed from TR*Labs* in Winnipeg for one month during the summer of 1996, and used for data collection at TR*Labs* in Saskatoon as well as at the University of Saskatchewan.

The IW95000 allows a user to capture up to 64 Megabytes of complete ATM cell information at OC-3 line rate. This capture buffer size works out to be approximately 1 million ATM cells. At 155 Mbps, this buffer size represents approximately 3 seconds of traffic. For traffic streams with lower bit rates the capture time can be significantly longer (minutes or hours). There is a timestamp recorded with each captured cell. Timestamps have a 1 microsecond resolution.

Cell-level traces can be written to disk and post-processed offline. In our case, we used perl scripts to analyze cell traces, and decode protocol information at the ATM, AAL-5, IP, and TCP levels.

2.4 Cell-Level Trace Captured

The trace analyzed in this paper is for an 8 Megabyte file transfer from the SGI Indy at the University of Saskatchewan to the SGI Indy at the University of Calgary. In particular, the ftp was done between `joe90.usask.ca` (206.75.91.131) and `mistaya.ffa.ucalgary.ca` (205.189.33.142). The file transfer was initiated from the University of Saskatchewan with the command `put xemacs-19.13-mips-sgi-irix5.3.tar`. This file is a 7,937,024 byte binary file.

The ftp application reported a transfer time of 14.22 seconds for this file, which gives a transfer rate of 4.5 Mbps. The ATM cell capture indicates a transfer time of 14.76 seconds, for a transfer rate of 4.3 Mbps. A total of 205,950 cells were sent between the two hosts.

For all of our tests, the IW95000 test set was situated physically beside the SGI Indy in Saskatoon (i.e., between `joe90` and the first ATM switch leading to the NTN). The IW95000 was capturing the complete cell streams from both `joe90` and `mistaya` in real-time in Saskatoon.

Because of the relatively long round-trip delays involved in this transfer, the placement of this measurement hardware is crucial in understanding the results. The IW95000 sees all cells sent by `joe90` approximately 7 ms before `mistaya` receives them, and sees all cells sent by `mistaya` approximately 7 ms after they were sent.

2.5 Relevant TCP Parameters

During the ftp connection, both hosts must agree on a maximum segment size to be used. Both hosts suggest a MSS and the smaller of the two suggestions must prevail. The standard Maximum Transmission Unit (MTU) for ATM is 9188 bytes, which leads to an MSS between 9140 and 9148 bytes depending on the amount of overhead at intermediate protocol layers.

During the connection setup phase in our experiments, `mistaya` first suggested a MSS of 512 bytes, which `joe90` agreed to in the next packet. All the TCP/IP packets in the transfer included 12 bytes worth of TCP options, which meant that only 500 bytes was available for user data in each segment. The 12 bytes of TCP options consisted of 10 bytes of *timestamp* information allowing the sender to calculate the RTT for each `ACK`, and 2 bytes of `NOP` for padding. A MSS of 512 bytes is the default size if no other information is available and is commonly used over long distances where packets may have to traverse several different networks, each with a different MTU.

While negotiating the MSS to be used during the connection, both hosts also use the TCP *window scale option* defined in RFC 1323 [9]. This option allows the hosts to avoid the 16-bit limitation on window size for *long fat pipes* [1]. For this connection, both hosts advertise a window of 64,000 bytes in the TCP header and then use the TCP window scale option to shift this value three positions for a maximum usable window of 512,000 bytes. This choice is an even multiple of the MSS and suggests that both hosts recognize the type of connection they are negotiating. This could also be related to the maximum socket sizes allowed on the two hosts since the usable window is often directly based on this value.

2.6 Summary

This section has presented background material on the measurement environment used for our experiments, and the experimental setup for collecting data on TCP/ATM traffic. The remainder of the paper describes the analysis of our ftp trace.

3 Analysis of the FTP Trace

The cell-level trace analyzed in this paper is for an 8 Megabyte file transfer from the SGI Indy at the University of Saskatchewan to the SGI Indy at the University of Calgary. This section presents our analysis of this file transfer trace.

3.1 General Traffic Characteristics

Figure 1 shows the overall traffic profile for our measured file transfer activity. The plot shows the number of ATM cells observed (transmitted or received by joe90) per 1 millisecond interval, for the full duration of the trace. It is immediately evident that something is wrong with the TCP/ATM performance for our file transfer: there are distinct alternating periods of cell transmissions and idle periods, with each such period lasting approximately one second.

Off-line analysis of the trace shows that the traffic is dominated by the send traffic from `joe90`. An analysis of the cells sent by `joe90` showed a total of 16,069 packets and 192,493 ATM cells with 8,017,436 bytes of data[2]. This gives a throughput figure of 4.345 Mbps but a goodput figure of 4.30 Mbps. The receiving host `mistaya` sent a total of 13,499 cells containing 6722 acknowledgement packets, and 132 bytes of data. This means that, on average, an acknowledgement was sent for every 1193 bytes of data. The data sent by `mistaya` consists almost entirely of connection setup and release commands.

Most of the large spikes in Figure 1 contain around 35,000 cells, which works out to approximately 1.68 Mbytes of data. These spikes are separated by idle periods that range from 1.1 to 1.4 seconds in duration.

[1] Networks with high bandwidth and/or long delay.

[2] For `joe90`, the difference $8,017,436 - 7,937,024 = 80,412$ bytes consists mainly of retransmitted data.

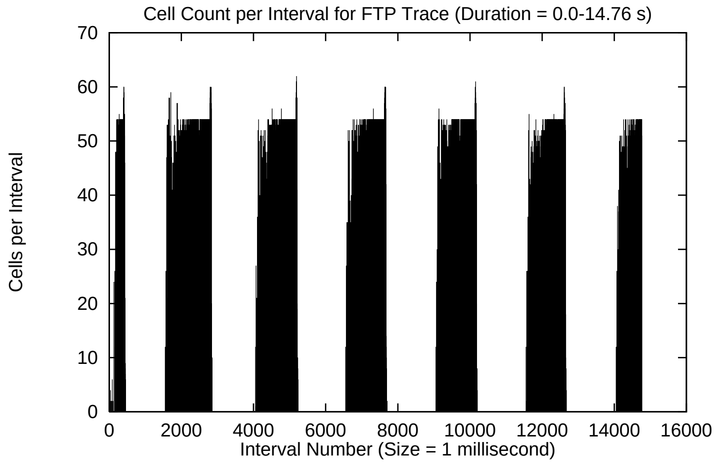

Figure 1: Cell Count per Interval for FTP Transfer over a WAN (Entire Trace)

Figure 2 zooms in on one of the spikes of activity, focusing on the time period from 4.0 to 5.5 seconds. Figure 2 shows that the sustained "bursts" of activity in Figure 1 actually have a much more interesting finer-grain structure.

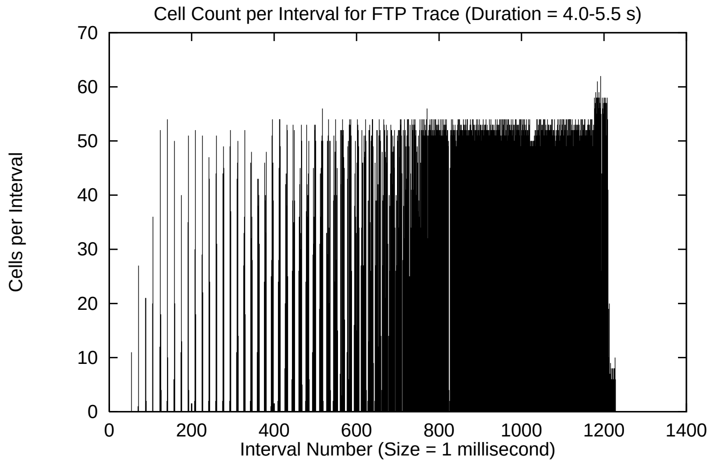

Figure 2: Cell Count per Interval for FTP Transfer over a WAN (Selected Interval of Trace from 4.0 to 5.5 seconds)

To the trained eye, this is exactly the behaviour expected from the TCP *slow start*[3] algorithm. The traffic source starts by sending a small amount of data into the network. As each small burst of data is acknowledged, the source is allowed to send larger and larger data bursts (the left half of Figure 2) into the network, until at last the traffic source is transmitting data at "full speed", in a sustained fashion (the black region of Figure 2).

Unfortunately, the "full speed" data transmissions appear to end abruptly (e.g., at time 5.2 seconds). Each blast is then followed by a one second idle period, before starting the next burst, again using slow start. This behaviour is perfectly consistent with the timeout and retransmission mechanisms in TCP to recover from lost packets.

[3]The TCP mechanisms mentioned in this paper are described briefly in Appendix A.

3.2 TCP Sequence Number Analysis

A different way to look at the same trace data is using TCP sequence numbers. Figure 3 shows two plots that clearly illustrate operation of the TCP congestion control algorithm and its method of slow start. Both of these plots show the current sequence number (in bytes) used by `joe90` on the vertical axis and the current time on the horizontal axis. The plot on the left shows the complete transfer, with the final sequence number equal to the size of the file. Every "j"-shaped line corresponds to one of the bursts seen in Figure 1. The throughput of the transfer is the slope of the line from the origin to the last point on the graph, and this gives a value of 4.3 Mbps.

We can also see the exponential shape of each line produced by the slow start algorithm, which allows the transfer rate to rapidly increase. The maximum slope of the spikes is approximately 18 Mbps. Looking closely, we can also see that all bursts (with the exception of the first one) start with a period of retransmissions where the sequence numbers overlap some of those sent during the last burst.

The plot on the right of Figure 3 zooms in on a portion of the first burst from 0.1 to 0.35 seconds. Here we can see slow start even more clearly with longer and longer sequences being sent. Every point on the graph represents the first sequence number of a packet that is transmitted. This means that there is a one-to-one correspondence between the number of points in a burst and the number of TCP segments transmitted.

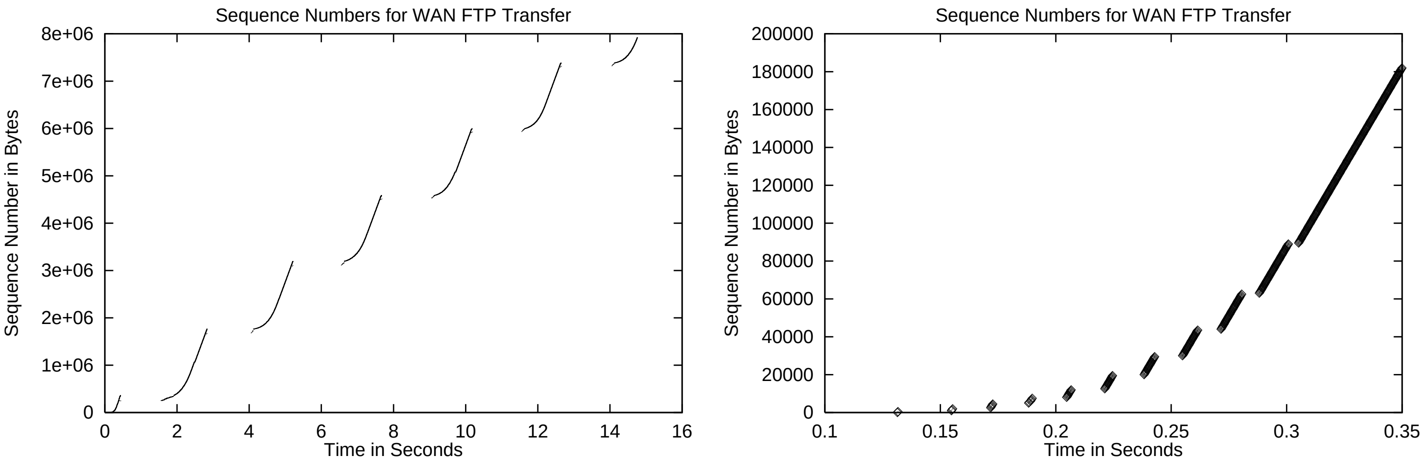

Figure 3: TCP Sequence Numbers vs Time Illustrating Slow Start During FTP Transfer

3.3 Our Hypothesis

We now turn our attention to the "burst and idle" pattern of Figure 1 and Figure 3. Clearly, packets are getting "lost" during the transfer, which causes the repeated occurrences of timeouts, retransmissions, and slow start.

What is interesting to note, however, is that the packets are not getting lost in the network (i.e., at the switches). Rather, the packets are getting "lost" before they even leave the ATM NIC on `joe90`. Manual analysis of the trace confirms that there are indeed patterns of missing TCP sequence numbers, and ensuing retransmissions of those sequence numbers at later points in time. Since the IW95000 is connected directly to `joe90` (i.e., between `joe90` and the first ATM switch traversed in the network), this can only mean that the packets are being lost before they leave the NIC. This is probably due to some sort of buffer overflow, but whether this happens in the device driver of Irix 5.3 or in the NIC is unclear.

Our own hypothesis for what is happening concerns the ATM NIC on `joe90`. The NIC provides a buffering mechanism so that IP packets delivered for transmission are buffered and then sent out at the maximum bandwidth allowed over the link. As the congestion window grows, the sending host (`joe90`) dumps an increasing number of TCP/IP packets to be delivered to `mistaya`. At the beginning of this cycle, the number of segments delivered to the NIC is small, so the NIC has no trouble buffering and sending them away. Meanwhile, `joe90` must wait for an acknowledgement from `mistaya` before delivering its next burst of segments to the NIC for delivery. It is this acknowledgement delay that allows the NIC on `joe90` to successfully deliver larger and larger numbers of TCP/IP packets without getting swamped by the sending host, which is certainly capable of delivering packets at a rate faster than 25 Mbps.

At some point, however, the congestion window grows large enough so that `joe90` is delivering packets faster than the NIC can transmit them at 25 Mbps. As a result, a backlog starts to build. Depending on the size of the buffer on the NIC, cells will continue to be transmitted in a 25 Mbps constant bit rate fashion. Successful acknowledgements will continue to arrive from `mistaya`, and `joe90` will continue to deliver new packets to the NIC. This behaviour continues until the buffer on the NIC overflows and segments are lost[4]. When the two hosts become aware of this (through *duplicate acknowledgements* or *retransmission timeouts*), they interpret it as network congestion, and react accordingly with standard TCP congestion algorithms.

The next two subsections provide further evidence for our diagnosis of this problem.

3.4 Analysis of Packet Loss

Figure 4 shows two plots illustrating the pattern of packet loss that occurs during the first spike of activity. The plot on the left shows the sequence numbers of packets within the congestion window that are *not* physically transmitted in the opening burst. Because the IW95000 is right beside `joe90`, we can be certain that these missing segments never left the NIC. The plot on the right shows the cell counts per 10 ms interval over the time from 0.10 to 0.45 seconds. In this plot, we can see that the output from the card is an approximately 20 Mbps CBR stream of cells starting at time 0.30 seconds. The first missing segments appear at time 0.38 seconds. This means that following time 0.30, `joe90` delivered enough segments to overflow the buffer so that at least one of the packets was discarded. This gap in the sequence numbers worked its way to the front of the buffer and appeared at time 0.38 seconds.

Unfortunately, we cannot calculate the NIC buffer size directly from this information because we don't know the time at which IP tried to deliver the datagram to the NIC, though we could make a reasonable guess. Slow start behaviour is not observed immediately after this packet loss so it is obvious that either:

- the NIC card did not alert IP to the overfull interface queue
- IP received an error from the NIC card but did not call the `tcp_quench` function

[4]Under normal circumstances, a full interface queue on the transmission medium should result in a `tcp_quench` order, which forces the sending host to reduce the congestion window to one MSS and begin slow start. This is described in more detail in Appendix A.

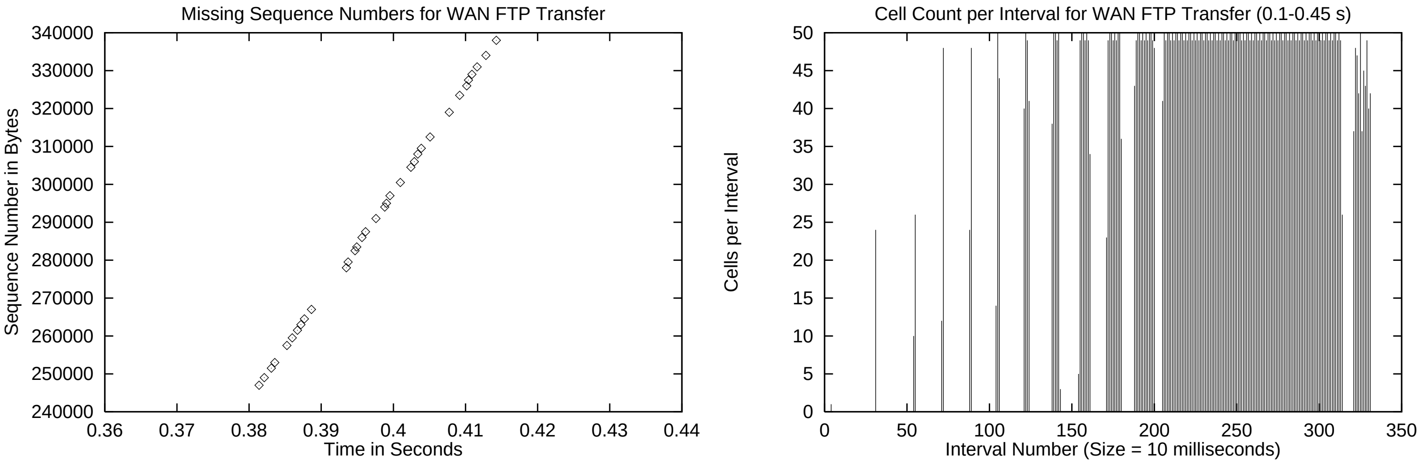

Figure 4: TCP Segment Loss Due to Buffer Overflow on the ATM NIC

- the packet was lost by some other mechanism in the operating system, and did not generate an error

3.5 Detailed Analysis of Packet Loss Behaviour

The size of the NIC buffer can be estimated by a more careful analysis of our TCP/ATM trace. In particular, we focus on the timing structure of the sequence numbers and acknowledgement numbers around the time of segment loss. Figure 5 shows a detailed plot of this information during one of the periods of interest.

Figure 5 clearly illustrates several features of TCP. These include:

- *slow-start* (the ramp up behaviour starting from the bottom left of the plot);
- the *congestion window* (the vertical distance between acknowledgement numbers and sequence numbers);
- round-trip delay (the horizontal distance between sequence numbers and the corresponding acknowledgement);
- *duplicate acknowledgements* (the horizontal portion of the acknowledgement number plot, starting at time 0.4);
- *fast retransmit* (hard to see, but its effect is indicated by a vertical increase in the acknowledgement number at time 0.43); and
- *fast recovery* (the topmost spurt at time 0.42 on the sequence number plot).

Recovery from missing segments (for a different portion of the trace) is shown in more detail in Figure 6. Several of the events depicted in Figure 6 also illustrate *false fast retransmits* [6].

Most importantly in Figure 5, we can see the round-trip delay as the gap between the sequence numbers being transmitted by `joe90` and the acknowledgement numbers coming back from `mistaya`. This time gap widens a bit over the course of the plot as the volume of data in the buffer on the NIC increases.

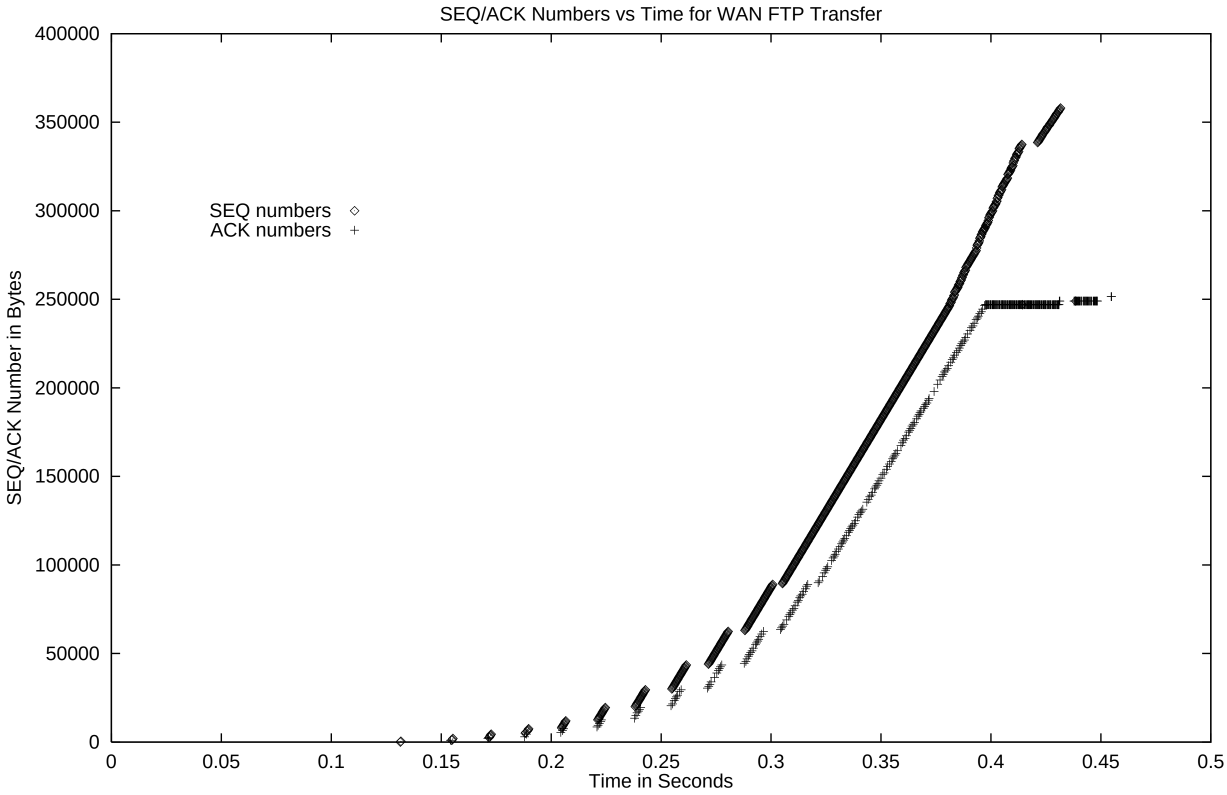

Figure 5: SEQ/ACK Numbers Before and After Loss of a TCP Segment

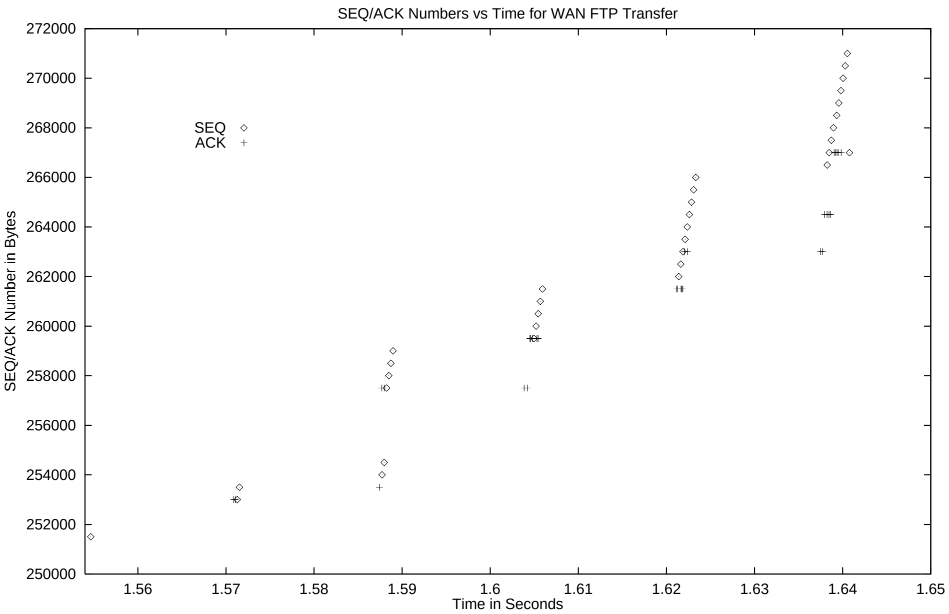

Figure 6: Recovery from Missing Segments

At some unknown time, `mistaya` detects the first missing packet, and is not permitted to acknowledge the packets that arrive after it. For this reason, `mistaya` begins sending duplicate acknowledgements to get `joe90` to retransmit the missing packet. These duplicate acknowledgements arrive at time $t \approx 0.4$ as shown on the plot but were sent at least 7 ms earlier by `mistaya`. Based on the fast retransmit and fast recovery algorithms, we expect that once `joe90` receives the third duplicate acknowledgement, the missing packet will immediately be retransmitted and the congestion window will be halved. Instead of an immediate retransmission, however, we see `joe90` continue to send a steady stream of "out of order" segments. These segments, which appear before the retransmission, represent the amount of data contained in the buffer on the NIC.

Table 1 summarizes the significant TCP/ATM events that are represented in Figure 5. The times listed for each event are in seconds from the start of the ftp transfer.

From the events detailed in Table 1, we can make several observations:

- pipeline effects dominate the behaviour of the two hosts
- the fast recovery and fast retransmit algorithms only work well when single isolated packets have been lost[5]
- the first missing segment caused `joe90` to resend the segment immediately, stop transmitting due to the congestion window being cut in half, and begin sending packets again as many duplicate ACK's inflated the congestion window to a point that allowed further transmission
- the second missing segment caused `joe90` to resend the segment[6] immediately, and stop transmitting as the congestion window is cut in half with a large amount of unacknowledged data
- the third missing segment caused `joe90` to take a 1.1 second break before responding with a retransmission triggered by a retransmission timeout.
- assuming that segment 247,001 is retransmitted by `joe90` immediately upon receiving the duplicate ACK, the transmit buffer on the ATM NIC is $\approx$ 38,500 bytes in size.

3.6 Summary

Our observed TCP/ATM performance problem stems from overrun at the sending workstation for the ftp transfer. The problem arises for several reasons. First, the flow control mechanism in TCP is solely an end-to-end flow control mechanism. There is no mechanism to do flow control between a sending TCP and an attached NIC. Second, our workstations are configured with large send and receive socket buffer sizes to better utilize our particular network, which has a

[5]The same observation is made by Hoe [6].

[6]The Hoe paper [6] states that only a single lost packet can be recovered by the fast retransmit algorithm. However, this is not always the case: in our situation, there were two successfully recovered packets, with only the third missing packet requiring a timeout. We believe that the success of the fast retransmit/recovery algorithm is a function of the delay-bandwidth product, the current size of the congestion window, and how close together the lost packets are in the sequence number space.

Table 1: Timeline of Significant TCP/ATM Events in Figure 5

Start Time	End Time	Event Description
0.130000	-	joe90 begins sending packets using slow start
0.310000	-	ATM NIC on joe90 begins to send a constant CBR stream, indicating that joe90 has built up a backlog of waiting packets
0.381200	-	joe90 misses sending packet with sequence number 247,001
0.382000	-	joe90 misses sending packet with sequence number 249,001
0.382107	-	joe90 continues normal transmission with segment 249,501
0.397681	-	first duplicate ACK for 247,001 arrives at joe90 from mistaya
0.397700	-	third duplicate ACK for 247,001 arrives at joe90 from mistaya We assume that joe90 immediately retransmits segment 247,001 (fast retransmit) to the ATM NIC and enters a congestion avoidance state (fast recovery).
0.397700	0.414280	38,500 bytes are transmitted from the NIC on joe90 with sequence numbers 292,001 to 337,501 (and 14 missing segments in that range)
0.414280	-	packet with sequence number 247,001 is physically retransmitted (though it is difficult to see in the plot)
0.414280	0.421248	joe90 does not send any packets, but duplicate ACK's continue to arrive from mistaya for segment 247,001. After the retransmission, joe90's congestion window was halved, so the amount of unack'ed data must have exceeded the (new) congestion window size.
0.421248	0.431140	The NIC on joe90 sends cells at near maximum rate with sequence numbers 338,501 to 357,001. This is made possible by a steady stream of duplicate ACK's, which continually inflate the congestion window, enabling new packet transmissions (fast recovery).
0.431315	-	ACK for missing segment 249,001 arrives at joe90 from mistaya, indicating that the retransmitted segment with number 247,001 arrived successfully at mistaya. The congestion window should now be reduced to the slow start threshold, with unacknowledged data again exceeding the size of the congestion window. Joe 90 should now know that segment 249,001 requires retransmission.
0.431380	0.431621	joe90 sends two more packets with sequence numbers 357,501 and 358,001 which may have been left in the interface queue on the NIC
0.431621	0.437770	No traffic is detected from either host. joe90 cannot send until more data has been acknowledged, and mistaya is not receiving packets to trigger duplicate ACK's, due to the earlier pause by joe90.
0.437770	-	Duplicate ACK's for missing segment 249,001 begin to arrive again at joe90. These were triggered by the burst starting at time 0.421248.
0.438518	-	Segment with sequence number 249,001 is retransmitted by joe90
0.438521	0.454638	Approximately 42 more duplicate ACK's arrive for segment 249,001 while joe90 sends nothing. The congestion window grows as these ACK's arrive, but not enough to allow more segments to be sent.
0.454856	-	Duplicate ACK for missing segment 251,501 arrives at joe90, indicating that mistaya finally received segment 249,001.
0.454856	1.554656	No traffic is sent from either host. The retransmission timer on joe90 expires. We again have a deadlock situation where joe90 cannot send segments because of unacknowledged data, and mistaya will not send duplicate ACK's because no new packets are being received.

large delay-bandwidth product. Third, our NIC card is configured to transmit traffic into the ATM network at a maximum rate of 25 Mbps. Finally, our SGI Indy is itself fast enough to inject TCP/IP packets into the NIC much faster than the draining rate of 25 Mbps. As a result, the NIC buffer eventually fills and overflows when a large enough window size is used. This behaviour is completely deterministic and repeatable.

4 Additional Experimental Results

4.1 Simulation Results

In an attempt to verify our hypothesis regarding the NIC buffer overflow, we attempted to recreate our file transfer performance results using a TCP/ATM simulation model [4, 14]. This simulation model, based on the NetBSD implementation of TCP, has detailed modeling of the socket layer, the TCP layer, and the AAL-5 layer [4].

Our initial runs with the simulation model were not promising: our simulation model showed a sustained throughput of 18 Mbps throughout the course of the simulated file transfer.

However, by making two small modifications to the simulation model, we were able to recreate the TCP/ATM behaviour observed in our real network. The first change involved reducing the maximum queue size allowed for our simulated "interface queue", which handles data passed from TCP to the AAL-5 layer. By reducing this queue from size 50 to size 5, we made the possibility of overrunning the queue much more likely. The second change involved removing the line of code that signalled a full interface queue back to the sending TCP. This change allows the TCP layer to blindly send packets to the AAL-5 layer, as fast as it can. Figure 7 illustrates the exact source code changes made in our simulation model.

```
if (qfull())
  { // Send queue is full.
    // Drop packet.
    ++aals.sqpktdrop;
    aals.sqbytedrop += pkt->pkt_size;
    return (1); // Signal dropped packet.
  }
```

(a)

```
if (q_size >= 5)
  { // Send queue is full.
    // Drop packet.
    ++aals.sqpktdrop;
    aals.sqbytedrop += pkt->pkt_size;
    return (0); // Don't signal it!
  }
```

(b)

Figure 7: Relevant Source Code in TCP/ATM Simulation Model: (a) Original Source Code; (b) Modified Source Code

Figure 8 shows the results for the modified TCP/ATM simulation model. With the two changes described above, the modified TCP/ATM simulation model behaves almost exactly like the TCP in our traces. This provides further evidence for our hypothesis concerning the overrun problem.

4.2 Netperf Results

We have conducted several other experiments in an attempt to identify the exact conditions under which our observed TCP/ATM performance problem occurs. These additional tests have

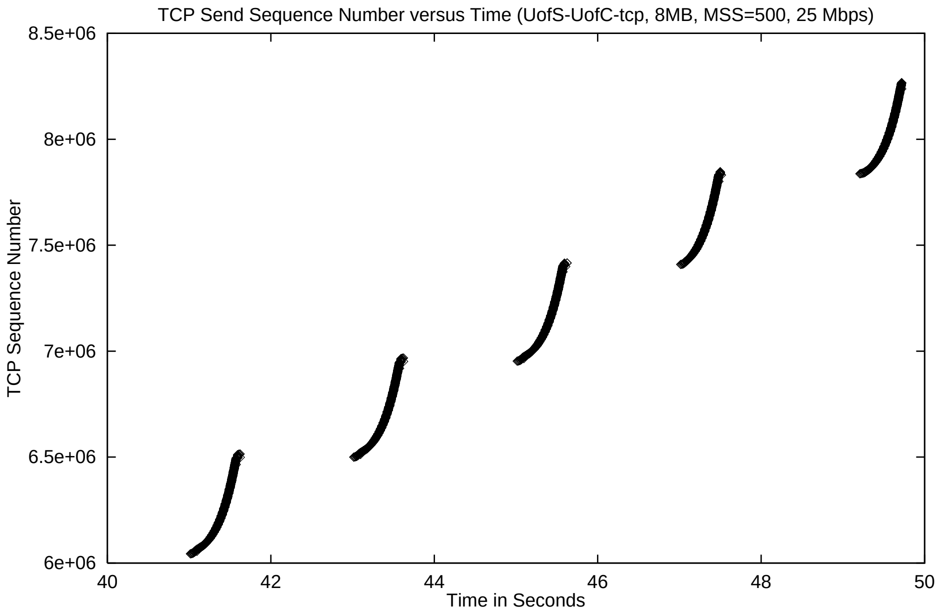

Figure 8: TCP Sequence Number Plot for Modified TCP/ATM Simulation Model

studied the effect of send socket buffer size, and the effect of workstation speed.

4.2.1 Effect of Send Socket Buffer Size

Our experiments with send socket buffer sizes were done using netperf. Netperf allows the user to set socket buffer sizes, message sizes, and other parameters in an attempt to evaluate end-to-end TCP performance.

Our tests were conducted between `joe90` and `mistaya`. For all of our tests, the receive socket buffer size was set to 512,000 bytes, and the maximum segment size (MSS) was set to 500 bytes[7]. The send socket buffer size was then varied from 1 kilobyte to 128 kilobytes in steps of 1 kilobyte.

The results from this experiment are illustrated in Figure 9. The results show that increasing the send socket buffer size first improves performance, but a plateau of approximately 16 Mbps occurs at 34 kilobytes, as the network "pipe" (i.e., delay-bandwidth product) fills. Beyond a send socket buffer size of 63 kilobytes, there is a sharp drop in throughput to the 4 Mbps range. Clearly, the overrun problem is present in this range.

Assuming that the standard IP interface queue size on Irix 5.3 is 50 packets (each of size 512 bytes in this case), then the maximum transmit buffer space on the NIC is estimated to be 38 kilobytes (63 kilobytes − 25 kilobytes).

4.2.2 Effect of Workstation Speed

Netperf tests were run between pairs of three different SGI Indy workstations on the NTN. One of these workstations is an older SGI Indy with a GIA-200 ATM NIC (not a GIA-200e), at the University of Alberta. The test results showed that ftp throughput to and from the older SGI

[7]We have also conducted experiments with the MSS set to 9140 bytes, based on the recommended ATM MTU of 9180 bytes. The same behaviour occurs.

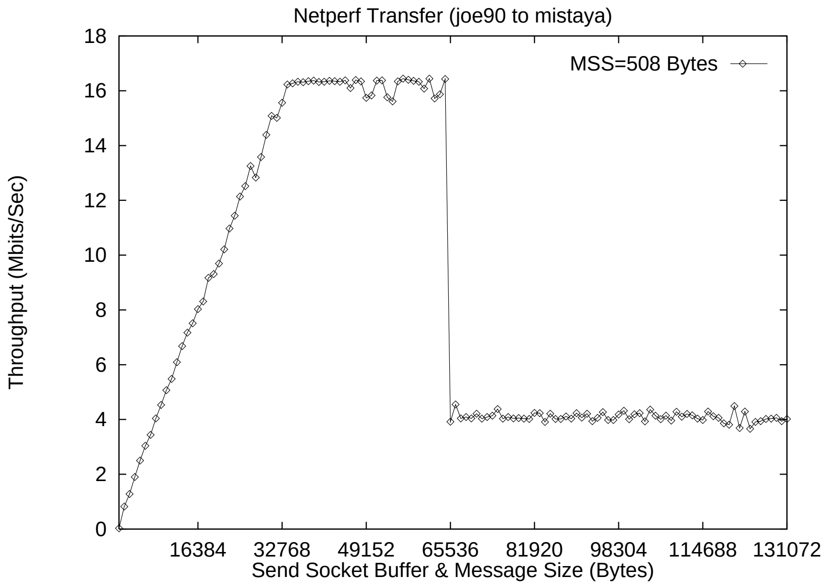

Figure 9: Netperf Results Illustrating Overrun Problem for Large Send Socket Buffer Sizes

Indy workstation (with an older NIC) resulted in significantly *higher* end-to-end throughput (as high as 14 Mbps), for otherwise similar configurations. Clearly, the overrun problem is less likely with slower workstations (or the older NIC).

4.3 Summary

Additional experiments have provided further evidence to support our diagnosis of the TCP/ATM performance problem. The overrun problem seems to be dependent on fast workstations, large send and receive socket buffer sizes, and a high delay-bandwidth network.

5 Summary and Conclusions

This paper presented our experiences in diagnosing a TCP/ATM performance problem, using cell-level measurements from an operational ATM network. The performance problem manifests itself in extremely low throughput for large file transfers between two SGI Indy workstations, connected by an experimental ATM wide area network.

Detailed analysis of our cell-level ATM traces shows that our observed TCP/ATM performance problem stems from overrun at the sending workstation for the ftp transfer.

The problem arises for several reasons. First, our workstations are configured with large send and receive socket buffer sizes to better utilize our particular network, which has a large delay-bandwidth product. Second, our NIC card is configured to transmit traffic into the ATM network at a maximum rate of 25 Mbps. Third, there is no apparent flow control mechanism between the sending TCP on the SGI Indy, and the attached ATM NIC. As a result, the NIC buffer eventually fills and overflows when a large enough window size is used. This behaviour is completely deterministic and repeatable, and causes repeated loss and retransmission of TCP segments. The ensuing timeouts to recover from these losses accounts for the low end-to-end throughput achieved.

Several solutions for the performance problem are possible. One solution is to constrain the maximum send socket buffer size to be less than 64 kilobytes (i.e., to disable the high performance TCP extensions available in Irix 5.3). Additional possibilities include modifying the start-up behaviour of TCP [6], modifying the fast recovery algorithm to better recover from multiple segment loss [10], or using TCP's proposed selective acknowledgement (SACK) feature [8].

Acknowledgements

Diagnosing this TCP/ATM performance problem would not have been possible without the NavTel IW95000 ATM test set borrowed from TR*Labs* in Winnipeg. The authors gratefully acknowledge the cooperation and assistance provided by Clint Gibler, Len Dacombe, Dave Blight, Jeff Diamond, and others at TR*Labs* Winnipeg. The authors would also like to thank Lawrence Brakmo, Jeff Mogul, and Vern Paxson for their insight into this performance problem.

Financial support for this research was provided by TR*Labs* in Saskatoon and by the Natural Sciences and Engineering Research Council (NSERC) of Canada, through research grants OGP0120969, IOR152668, and through an Industrial NSERC Summer Undergraduate Research Award.

Appendix A: TCP Details

This section provides some background details on the operation of TCP, particularly its algorithms for congestion avoidance and control. The algorithms of interest are called slow start, congestion avoidance, fast retransmit, and fast recovery. For full details on the design and implementation of TCP, the reader is referred to Comer [2] and Stevens [13, 15].

A.1 Sliding Window Flow Control

TCP is a transport layer protocol that operates using a sliding window flow control mechanism. Flow control is used to constrain the maximum number of data bytes that can be in transit at any time from the sender to the receiver.

In TCP, every sending host must maintain two windows: the usable window advertised by the receiver and another window called the congestion window. The sliding window of unacknowledged data used by the sending host is the minimum of the current congestion window and the advertised window size.

A.2 Slow Start Algorithm

When a new connection is established, the congestion window is initialized to one maximum size segment. A segment is sent, and if it is acknowledged before the retransmit timer goes off, one MSS worth of bytes is added to the congestion window. The sender may now transmit two segments without waiting for an acknowledgement. Every time a segment is acknowledged within the timeout period, the congestion window is expanded by one segment. This effectively doubles the congestion window after every successful burst of data. The algorithm that implements this is called *slow start*.

A.3 Congestion Avoidance Algorithm

The TCP congestion control algorithm maintains a third parameter called the *threshold*, which is initialized to 64 Kbytes. Once the congestion window is larger than the threshold, the doubling behaviour stops. Instead, one MSS worth of bytes is added for every successfully transmitted and acknowledged *window* of data. If a timeout occurs, the threshold is halved, and the congestion window is reset to one MSS. Of course, if the congestion window grows larger than the usable window advertised by the host, then the usable window will become the new limiting factor. This slower growth governed by the threshold value is known as "*congestion avoidance*".

A.4 Fast Retransmit Algorithm

If three (usually) consecutive duplicate ACK's arrive for the same sequence number, then TCP deduces that a segment with that sequence number has been lost. The *fast retransmit* algorithm allows the sender to immediately retransmit this missing segment, even if the timer has not yet expired.

A.5 Fast Recovery Algorithm

The *fast recovery* algorithm allows the sender to perform congestion avoidance rather than slow start after a segment has been retransmitted by the fast retransmit algorithm. This improves throughput under moderate congestion when large windows are being used.

The receiving host will send a duplicate ACK for every out-of-order packet it receives. When the number of duplicate ACK's equals three, the sender will retransmit the missing segment and perform congestion avoidance [8]. This involves cutting the slow start threshold to one half of its current congestion window and then setting the congestion window to the slow start threshold plus the number of out-of-order segments for which the receiving end has generated duplicate ACK's.

Every duplicate ACK after the third represents another packet that has successfully traversed the network and been received at the other end. Therefore, the congestion window is incremented by one segment for every duplicate ACK after the third. This aggressive window manipulation allows the sending host to keep the network "*pipe*" full when data has not been ACK'ed due to an out-of-order or missing segment. When the retransmitted segment is finally received and acknowledged, the temporary inflation of the congestion window is terminated by setting it back to the slow start threshold.

A.6 Retransmission Timeout

The retransmission timer is set whenever a segment is sent and an acknowledgement is expected from the other end. If a retransmission timeout (RTO) occurs before an ACK is received from the other end, the segment is retransmitted, the congestion window is reduced to one MSS, and a slow start phase is entered. The minimum value for the RTO is one second, which serves as the initial default value. The RTO is dynamically calculated over the connection between two hosts using a formula based on the smoothed round-trip time (RTT) estimator and its smoothed mean deviation estimator.

The RTT is only calculated for segments that were not retransmitted. The calculation is based on a timer that TCP keeps for every segment. Newer versions of TCP may use the timestamp option [9], which is an extra 12 bytes of information in the header of every segment that allows more accurate and robust RTT estimates.

[8] In older versions of TCP, the slow start phase was entered after the retransmission.

A.7 TCP Quench Function

TCP has a function called `tcp_quench` that may be called for several reasons:

- the interface queue on the transmission medium is full
- IP is unable to allocate a needed mbuf
- a source quench is received for the connection

When the `tcp_quench` function is called, the congestion window is set to one MSS causing slow start to take over. The threshold value, however, remains unchanged. The interface queue may become full, for example, if the network interface on a host is unable to access a busy network or is receiving packets for transmission faster than they can be sent. The hardware device will signal higher level protocols of this condition, eventually resulting in an aggressive TCP algorithm receiving the *quench* order.

A datagram is sent by passing data to the `tcp_output` function. If the `tcp_output` function is unable to send the datagram due to a full interface queue, the datagram will be discarded and the function will return zero kilobytes available in the send window. An error will not be generated, however, so it is up to the TCP algorithms and timers to recognize that a datagram was lost.

References

[1] A. Bianco, "Performance of the TCP Protocol over ATM Networks", *Proceedings of the 3rd International Conference on Computer Communications and Networks*, San Francisco, CA, pp. 170-177, September 1994.

[2] D. Comer, *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture*, Second Edition, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[3] D. Comer and J. Lin, "TCP Buffering and Performance over an ATM Network", *Internetworking: Research and Experience*, Vol. 6, No. 1, pp. 1-14, March 1995.

[4] R. Gurski and C. Williamson, "TCP over ATM: Simulation Model and Performance Results", *Proceedings of the 1996 IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, Phoenix, Arizona, pp. 328-335, March 1996.

[5] M. Hassan, "Impact of Cell Loss on the Efficiency of TCP/IP over ATM", *Proceedings of the 3rd International Conference on Computer Communications and Networks*, San Francisco, CA, pp. 165-169, September 1994.

[6] J. Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", *Proceedings of the 1996 ACM SIGCOMM Conference*, Stanford, CA, pp. 270-280, August 1996.

[7] V. Jacobson, "Congestion Avoidance and Control", *Proceedings of the 1988 ACM SIGCOMM Conference*, Stanford, CA, pp. 314-329, August 1988.

[8] V. Jacobson and R. Braden, "TCP Extensions for Long Delay Paths", RFC 1072, October 1988.

[9] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[10] M. Mathis and J. Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control", *Proceedings of the 1996 ACM SIGCOMM Conference*, Stanford, CA, pp. 281-291, August 1996.

[11] K. Moldeklev and P. Gunningberg, "How a Large ATM MTU Causes Deadlocks in TCP Data Transfers", *IEEE/ACM Transactions on Networking*, Vol. 3, No. 4, pp. 409-422, August 1995.

[12] A. Romanow and S. Floyd, "Dynamics of TCP Traffic over ATM Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 13, No. 4, pp. 633-641, May 1995.

[13] W. Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, Reading, Massachusetts, 1993.

[14] B. Unger, F. Gomes, Z. Xiao, P. Gburzynski, T. Ono-Tesfaye, S. Ramaswamy, C. Williamson, and A. Covington, "A High Fidelity ATM Traffic and Network Simulator", *Proceedings of the 1995 Winter Simulation Conference*, Arlington, VA, December 1995.

[15] G. Wright and W. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation.* Addison-Wesley, Reading, Massachusetts, 1995.