

TCP Buffering And Performance Over An ATM Network

Purdue Technical Report CSD-TR 94-026 (Revision)

Douglas E. Comer and John C. Lin*
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, U.S.A.

March 16, 1994

(Also appeared in *Journal of Internetworking: Research and Experience*, Vol. 6 (1), Pages 1-13, March 1995.)

Abstract

This paper reports a series of experiments to measure TCP performance when transferring data through an Asynchronous Transfer Mode (ATM) switch. The results show that TCP buffer sizes and the ATM interface maximum transmission unit have a dramatic impact on throughput. We observe a throughput anomaly in which an increase in the receiver's buffer size decreases throughput substantially. For example, when using a 16K octet send buffer and ATM Adaptation Layer 5 on a 100 megabit per second (Mb/s) ATM path, the mean throughput for a bulk transfer drops from 15.05 Mb/s to 0.322 Mb/s if the receiver's buffer size is increased from 16K octets to 24K octets. This paper analyzes the performance, explains the anomalous behavior, and describes a solution that prevent the anomaly from occurring.

*This work was supported in part by a fellowship from UniForum Association.

1 Introduction

Asynchronous Transfer Mode (or ATM) is a connection-oriented data communication technology that switches 53-octet data units called *cells* [8, 9]. ATM's fixed-size cell and early binding of routing information during connection setup make ATM suitable for high-speed data communication. Thus, standards committees (e.g., ANSI T1, ITU Study Group XVIII) have chosen ATM as an underlying transport technology for many Broadband Integrated Services Digital Network (B-ISDN) protocol stacks [7].

Although the standards committees are still working to refine ATM standards, network equipment manufacturers have developed ATM Local Area Network (LAN) equipment that provides gigabit aggregate bandwidth with connections to desktop computers. Many ATM LAN switches support the widely used TCP/IP Internet proto-

col suite by allowing the Internet Protocol (IP) [16] to operate over ATM. A user who connects to an ATM network can run existing applications that use the Transmission Control Protocol (TCP) [17] or the User Datagram Protocol (UDP) [15] without modification.

Users who share a conventional network (e.g., a 10 Mb/s Ethernet) expect dramatic increases in performance from a dedicated ATM connection that operates an order of magnitude faster. However, measurement of file transfers using FTP [19] showed a surprising result: the same *ftp* program that performs well over a 10 Mb/sec Ethernet can perform worse over an 100 Mb/s ATM path. For example, when transferring a 4.4 megabyte data file between two hosts connected to the same Ethernet, *ftp* reports a mean throughput of 1.313 Mb/s. However, using the same software and computers to transfer the file across a 100 Mb/s ATM path produce a mean throughput of only 0.366 Mb/s. Furthermore, the ATM network management software reports no cell lost. The low throughput prompted us to investigate the effects of TCP buffering on its performance [5]. Experiments revealed the sizes of the sender's and receiver's buffers have a dramatic effect on performance.

The remainder of this paper is organized as follows. Section 2 describes the ATM network configuration used to conduct the experiments,

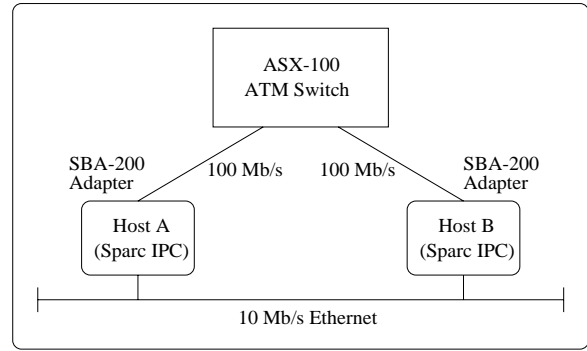


Figure 1: Configuration of an ATM network used to measure TCP throughput

the tool used to measure TCP performance, and the experimental procedures. Section 3 presents the results of the experiments and identifies a throughput anomaly in which an increase in the receiver's buffer size decreases TCP throughput significantly. Section 4 explains the cause of the throughput anomaly and describes a solution that prevents the anomaly from occurring. Section 5 summarizes the paper.

2 Measuring TCP Performance Over ATM

Figure 1 illustrates the network configuration for conducting the experiments. Two multi-homed Sun Microsystems' SPARCstation IPCs, *A* and *B*, running SunOS 4.1.1¹ are used to measure TCP performance over ATM. Each host has two network interfaces: one connects to an Ethernet and the other connects to a Fore Systems' ASX-100 ATM switch via a 100 Mb/s multi-mode fiber link.

¹SunOS 4.1.1 TCP is a version of 4.3BSD-Tahoe TCP.

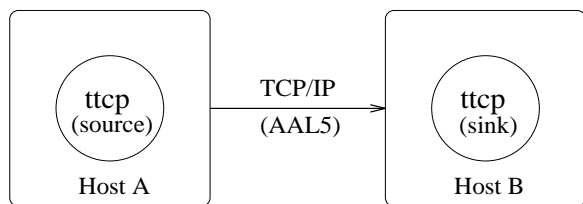


Figure 2: Using *ttcp* to measure TCP throughput

Each host uses a Fore SBA-200 ATM adapter card² to interface with the ATM switch. The adapter card embeds a dedicated processor and special purpose hardware to handle ATM Adaptation Layer 5 (AAL5) [6, 10]. The Maximum Transmission Unit (MTU) of the ATM interface and the Ethernet interface is 9188 octets and 1500 octets, respectively.

2.1 Measurement Tool

We used a public domain C program called *ttcp*³ to measure TCP throughput. A *ttcp* running on host *A* uses the BSD socket interface provided by SunOS to communicate with another *ttcp* on host *B*. As Figure 2 illustrates, we configured one *ttcp* as a source and the other as a sink. Once a user has specified the amount of data to transmit, the source *ttcp* continuously transmits the data to the destination *ttcp* until all the data are transmitted; the destination *ttcp* simply discards the data it receives. Program *ttcp* also provides an option for users to specify the sending TCP's send buffer size

²The SBA-200 cards use firmware version 1.8.1 and software version 2.2.6.

³The program *ttcp* is available for anonymous ftp on host gwen.cs.purdue.edu in directory `/pub/lin`.

and the receiving TCP's receive buffer size.

The experiments use two timestamps at the sending *ttcp* to calculate throughput: one taken at the instant before it calls the *write* system call to start transmitting data, and the other taken at the instant after it finishes the transmission. System call *gettimeofday* provides the timestamps. On a Sun IPC running SunOS 4.1.1, the timestamps have a granularity of one microsecond. TCP throughput is calculated as the total number of application data transmitted divided by the interval between the two timestamps.

Because ATM is connection-oriented, a connection must be established between the sender and the receiver before IP datagrams can be transmitted. Once an ATM connection between two IP hosts has been established, Fore Systems' software provides a caching mechanism such that an ATM connection is only closed when the connection is quiet for approximately 15 minutes [2]. We artificially established an ATM connection before each experiment. Thus, the reported throughput does not include connection setup time.

2.2 Experimental Procedures

We conducted 100 experiments to investigate the effect of send and receive buffer sizes on TCP throughput when transferring bulk data over a 100 Mb/s ATM path. The send and receive buffer sizes range from 16K octets to 51K octets in a 4K

		Receive Buffer Size (octet)									
		16K	20K	24K	28K	32K	36K	40K	44K	48K	51K
Send Buffer Size (octet)	16K	15.05	13.60	0.322	0.319	0.319	0.467	0.469	0.466	0.469	0.469
	20K	15.99	14.60	15.07	14.87	15.40	14.24	1.095	1.095	0.548	0.549
	24K	17.71	16.79	16.74	16.32	17.40	17.31	17.42	17.12	0.760	0.740
	28K	16.57	17.69	17.93	18.13	18.36	19.20	19.74	19.78	18.38	18.20
	32K	14.63	18.96	18.42	19.23	19.14	19.74	19.96	20.31	19.69	19.17
	36K	14.33	19.22	18.12	19.82	19.77	19.92	20.56	20.49	20.13	20.20
	40K	15.16	19.34	18.85	19.73	20.11	20.41	20.81	20.74	20.69	20.57
	44K	14.80	19.40	18.27	20.39	20.16	20.74	20.99	20.87	20.89	20.70
	48K	14.62	19.46	18.34	20.48	20.26	20.41	20.85	20.83	20.93	20.83
	51K	13.92	19.41	18.26	20.50	20.06	20.21	20.88	20.91	21.21	21.06

Note 1: Throughput numbers are in megabits per second (Mb/s).
 2: Shaded area indicates abnormal TCP throughput.

Table 1: TCP buffer sizes and mean throughput

increments. The minimum buffer size of 16K was selected because the SunOS kernel is configured to use a default send and receive buffer size of 16K when installing the driver software for the ATM adapter cards. The maximum buffer size of 51K was selected because SunOS 4.1.1 TCP restricts the buffer size to 52428 octets.

Each experiment consists of 50 independent throughput measurements. In each measurement, the source transmits 32 megabytes of data to the sink. There is a delay of 5 seconds between measurements. All the experiments use AAL5 to encapsulate IP datagrams.

3 Results

Table 1 shows the mean throughput measured for each experiment. As Table 1 shows, in general,

TCP throughput increases as the sender's and receiver's buffer sizes increase. Some experiments, however, show a decrease in mean throughput when send and/or receive buffer sizes increase. For example, when a sending TCP uses a 16K buffer, the mean throughput *decreases* about 10% when the receiving TCP *increases* the receive buffer size from 16K to 20K (see subsection 4.2 for an explanation). Also, as Figure 3 shows, when the sender and receiver use the same buffer size, TCP performs better with a 16K buffer than with a 20K buffer.

Surprisingly, as the shaded entries in Table 1 shows, certain combinations of unequal send and receiver buffer sizes cause exceptional low throughput. Furthermore, the exceptional low throughput occurs when a receiving TCP *increases* its receive buffer size. For example, when

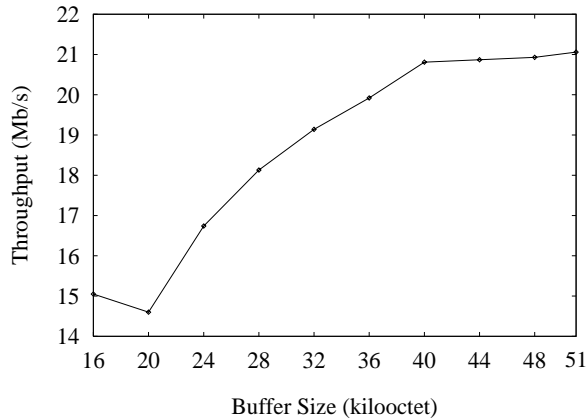


Figure 3: TCP mean throughput when sender and receiver use the same buffer size

using a 16K send buffer, TCP mean throughput decreases from 15.05 Mb/s to 0.322 Mb/s if the receive buffer size is increased from 16K to 24K. The next section explains the anomalous behavior and describes a solution to prevent it from occurring.

4 Analysis of the Results

Although network analyzers can be used to capture data on a shared access network (e.g., an Ethernet), the technique does not work well in a point-to-point, non-shared access ATM network. Thus, we use *kernel probing* [14] to study the observed TCP throughput anomaly over ATM. The technique uses a data structure in the kernel to store information gathered by inserted probing code at various kernel modules. An application program reads the gathered data from the kernel for off-line analysis.

4.1 Analysis of the Throughput Anomaly

By analyzing the gathered data, we conclude that the interaction of the following items causes the dramatic throughput decrease:

1. Sender's send buffer size
2. Receiver's receive buffer size
3. The MTU of the ATM interface
4. The TCP maximum segment size (MSS)
5. The way user data is added to the TCP send buffer
6. Sender side Silly Window Syndrome avoidance
7. Receiver side delayed acknowledgment algorithm

The first two items are configurable by applications in SunOS by using *setsockopt* system call. The MTU of Fore Systems' ATM interface card is 9188 octets for IP over ATM when using AAL5⁴. SunOS 4.1.1 calculates TCP MSS as 9148 octets (i.e., 9188 minus the default TCP and IP header sizes) in our ATM network configuration. Items 5 to 7 are implementation related; we describe how SunOS 4.1.1 implements them below.

⁴RFC-1626 [1] proposes that the default IP MTU for use over ATM AAL5 is 9180 octets.

4.1.1 Adding Data to TCP Buffer

The SunOS 4.1.1 implements TCP buffers as a list of *mbufs* [12]. Each *mbuf* can store up to 112 octets of data or contain a pointer to a 1K octet memory block for storing large messages. During bulk data transfer, if the send buffer is larger than 4K and the user has more than 4K data to send, SunOS adds user data in blocks of 4K octets to *mbufs*, then invokes a TCP routine to transmit the data; if the available space in the send buffer is smaller than 4K, SunOS adds data in multiples of 1K octet block.

4.1.2 Sender Side SWS Avoidance (Nagle's Algorithm)

Silly Window Syndrome (SWS) [4] is characterized as a situation in which a steady pattern of small TCP window increments results in small data segments being transmitted. Sending small data segments lowers TCP throughput because TCP and IP headers consume network bandwidth. To avoid SWS, both sender and receiver must implement SWS avoidance algorithm [3]. On the receiver side, it must avoid advancing the right window edge in small increments when it receives small data segments. On the sender side, it must avoid sending small data segments to the receiver even if the receiver has space available to accept them.

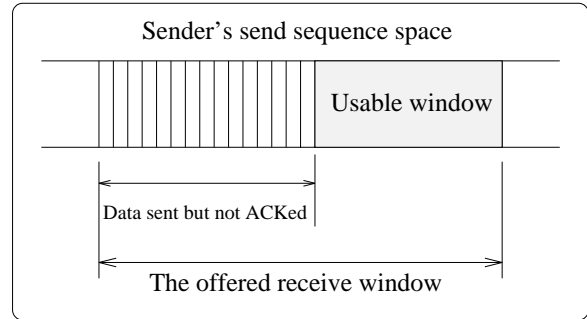


Figure 4: Illustration of a sender's usable window

For applications that are character-oriented (e.g., remote login, TELNET [18]), every character generated by the applications must be *pushed* explicitly by the application or TCP to avoid deadlocks. If a TCP transmits every pushed data, the result is a stream of one octet data segments. To better utilize network resources, a TCP tries to buffer segments that are small compared to the size of TCP and IP headers. However, to avoid deadlock, TCP must not buffer a data segment that needs immediate delivery. Nagle's algorithm [13] provides a simple solution to the dilemma: if there is unacknowledged data, TCP buffers all data (even if the PUSH bit is set) until TCP can send a maximum-sized segment or until all the outstanding data has been acknowledged [3, 13]. SunOS 4.1.1 TCP implements Nagle's algorithm as follows⁵:

S1: Let D be the amount of data to be transmitted, U be the *usable window*⁶ [4] as illustrated in

⁵Another condition *S3* which is irrelevant in explaining the observed exceptional low throughput will be discussed in subsection 4.2.

⁶A sending TCP's congestion avoidance scheme may re-

Figure 4. If $\min(D, U) \geq 1 * MSS$, then transmit a segment with $1 * MSS$ octet of data.

S2: If an ACK from the peer acknowledges all the outstanding data and there are X octets of data waiting in the send buffer, then transmit a segment with $\min(X, U)$ octets of data.

In condition *S1*, if a sender has at least $1 * MSS$ octets of data to be transmitted and the receiver has space to receive a maximum-sized segment, TCP transmits a maximum-sized segment. Condition *S2* requires TCP to transmit buffered data when the peer acknowledges all the outstanding data. When there is unacknowledged data, conditions *S1* and *S2* allow TCP to buffer small data segments until it can send a maximum-sized segment or all the unacknowledged data have been acknowledged. When a connection is idle (i.e., there is no unacknowledged data), TCP immediately transmits data added to the send buffer even if the amount of data is less than $1 * MSS$.

Figure 5 illustrates how a sending TCP with 16K send buffer uses conditions *S1* and *S2* to decide when to transmit data segments over an ATM path to a receiver with 24K receive buffer. TCP transmits 4K octets in the first segment because the connection was idle. Because the MSS is 9148 octets, after SunOS finishes adding the fourth 4K octet block, TCP transmits a second data segment with 9148 octet of data leaving 3140 (12K - 9148) octets of data waiting to be transmitted. When the peer acknowledges segments 1 and 2, TCP transmits a 3140 octet data segment (condition *S2*).

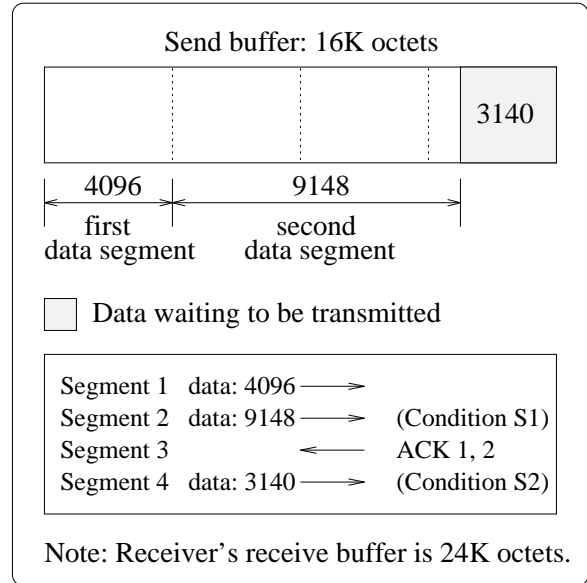
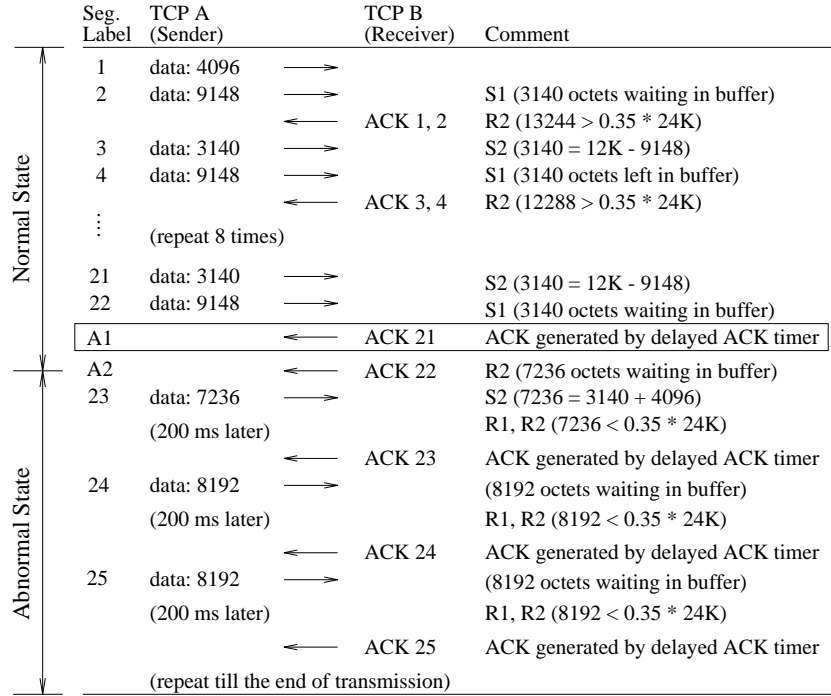


Figure 5: Illustration of how a TCP determines when to send data over an ATM path

octets of data queued in the send buffer (condition *S1*). When the peer acknowledges segments 1 and 2, TCP transmits a 3140 octet data segment (condition *S2*).

4.1.3 Receiver Side Delayed ACKs

A receiving TCP can reduce protocol processing at both ends and generate less traffic by using *delayed ACKs* [4]. A receiving TCP implements delayed ACKs by generating fewer than one ACK per data segment received. A TCP *should* implement a delayed ACK, but should not excessively delay an ACK because TCP uses ACKs to estimate packet round-trip time and determine how much more data to transmit [3, 11]. RFC-1122 recommends that in a stream of maximum-sized segments there should be an ACK for at least every



Note: 1. A's send buffer size is 16K, and B's receive buffer size is 24K.
 2. S1, S2, R1, and R2 are condition labels.

Figure 6: Illustration of the segment exchange between two TCPs that leads to a circular-wait state

second segment. SunOS 4.1.1 TCP implements delayed ACKs and uses it by default. The following two conditions determine when a SunOS 4.1.1 TCP should transmit an ACK if delayed ACKs is used:

R1: If the receive sequence space has advanced at least 2*MSS octets and receive buffer is empty, then transmit an ACK.

R2: If the receive sequence space has advanced at least 35% of the total receive buffer space, then transmit an ACK.

Note that the receive sequence space advances as an application extracts data from the receive

buffer, and TCP checks condition *R1* before condition *R2*. Condition *R1* allows TCP to acknowledge the peer after every two maximum-sized segments received by an application. For a receiving TCP with a small receive buffer, as compared to TCP MSS, condition *R2* generates ACKs to allow the sending TCP to transmit more data.

RFC-1122 mandates that a TCP must ACK the peer within 500 milliseconds (ms) after receiving data. Observe that the above two conditions do not guarantee that a receiving TCP will meet the requirement. For example, if an application extracts data from the receive buffer too slowly, TCP can delay sending an ACK for more than 500 ms. Therefore, SunOS 4.1.1 TCP schedules a *de-*

layed ACK timer event every 200 ms to check for possible delayed ACKs [12]; an ACK is transmitted when the timer expires and ACKs have been delayed.

4.1.4 Anomalous Behavior

TCP experiences low throughput when the sending TCP (*A*) has a buffer of 16K octets and the receiving TCP (*B*) has a buffer of 24K octets. As Figure 6 illustrates, after *B* acknowledges segments 1 and 2, a repeated pattern of two data segments from *A* and an immediate ACK from *B* lasts till segment *A1*. The first data segment contains 3140 octets of data because the ACK from *B* acknowledges all the outstanding data sent by *A*. The second data segment is a maximum-sized segment because *A* adds three blocks of 4K octets data in the send buffer, then applies condition *S1* to send an 1*MSS segment leaving 3140 octets of data queued in the send buffer. At this point, the available space in the send buffer is 956 octets (16K-12K-3140). TCP *A* does not fill up the available space because the space is less than 1K and the application has more than 1K of data to send. Thus, after sending the two data segments, *A* has 3140 octets of data queued in its send buffer and waits for an ACK from *B*. When an ACK from *B* acknowledges the two data segments, the communication pattern repeats.

Segment *A1*, which is generated by *B*'s delayed

ACK timer, breaks the repeated pattern. Segment *A1* allows SunOS to add 4K octets of data to the send buffer. After *A* receives segment *A2* that acknowledges all the outstanding data, by condition *S2*, it immediately sends a data segment with 7236 (3140+4096) octets to *B*. Because 7236 is less than 2*MSS and also less than 35% of 24K, *B* delays acknowledging *A* and expects *A* to send more data. Meanwhile, *A* adds only 8K octets of data to the send buffer even though the send buffer has 9148 octet space available. Because 8K is less than 1*MSS and does not satisfy condition *S2*, *A* waits for an ACK from *B* before transmitting the 8K data.

A *circular-wait* situation has occurred: *A* is waiting for an ACK from *B* before it sends more data, and *B* is waiting for more data from *A* before it sends an ACK. Finally, *B*'s delayed ACK timer expires and causes *B* to send an ACK that breaks the circular-wait. After *A* responds to the ACK from *B* by sending an 8K data segment, the circular-wait situation occurs again. Thus, a lock-step interaction in which a circular-wait followed by an ACK that breaks the circular-wait has established. Because the delayed ACK timer generates one ACK per 200 ms, TCP throughput decreases dramatically.

4.1.5 Discussion

As the previous subsection describes, an ACK segment (segment *A1* in Figure 6) causes a TCP connection to transit from a normal state to a repeated circular-wait state that lowers the throughput significantly. Because the delayed ACK timer generates the ACK, the time at which the first circular-wait occurs depends on when the delayed ACK timer will generate such an ACK. The longer the data transfer takes, the more likely a transition to the repeated circular-wait state becomes. Once in the repeated circular-wait state, TCP throughput is approximately 8K octets per 200 ms (or 0.3125 Mb/s).

Certain combinations of send and receive buffer sizes cause the repeated circular-wait state to occur shortly after connection establishment. For example, when a sender with 16K buffer communicates with a receiver with a 36K buffer, the first circular-wait occurs after the sender transmits the fourth data segment (see Figure 6) because 12288 (3140+9148) is less than 18296 ($2 \cdot \text{MSS}$) and 12903 (35% of 36K). Then, the followed repeated circular-wait yields a throughput of 12288 octets per 200 ms (or 0.46875 Mb/s). However, if the delayed ACK timer generates an ACK for segment 3, a different pattern of repeated circular-wait that yields a throughput of 8K octets per 200 ms (or 0.3125 Mb/s) occurs.

Figure 7 summarizes the patterns of repeated

circular-wait for various combinations of send and receive buffer sizes. It shows whether the delayed timer causes the transition to the repeated circular-wait or not, and the throughput achieved when such repeated circular-wait occurs.

4.1.6 Preventing Anomalous Behavior

We have discovered a way to avoid the anomalous behavior: the anomaly will not occur if the sender uses a buffer that is at least $3 \cdot \text{MSS}$ octets. The method is particularly elegant because it allows a sending TCP to prevent anomalous behavior regardless of the receiver's buffer size.

To understand our solution, observe that even if it is using delayed ACKs, an implementation of TCP that follows RFC-1122 must generate an ACK after receiving at least two maximum-sized segments, and consider the two cases of a receive buffer. In case the receiver's buffer is smaller than $3 \cdot \text{MSS}$ octets, the buffer will be more than 35% full, and the receiver will send an ACK according to *R2*. In case the receiver's buffer is larger than $3 \cdot \text{MSS}$ octets, the sender will have at least $2 \cdot \text{MSS}$ octets of data outstanding at any time, and the receiver will send an ACK according to *R1*. Thus, the receiver will always acknowledge segments promptly.

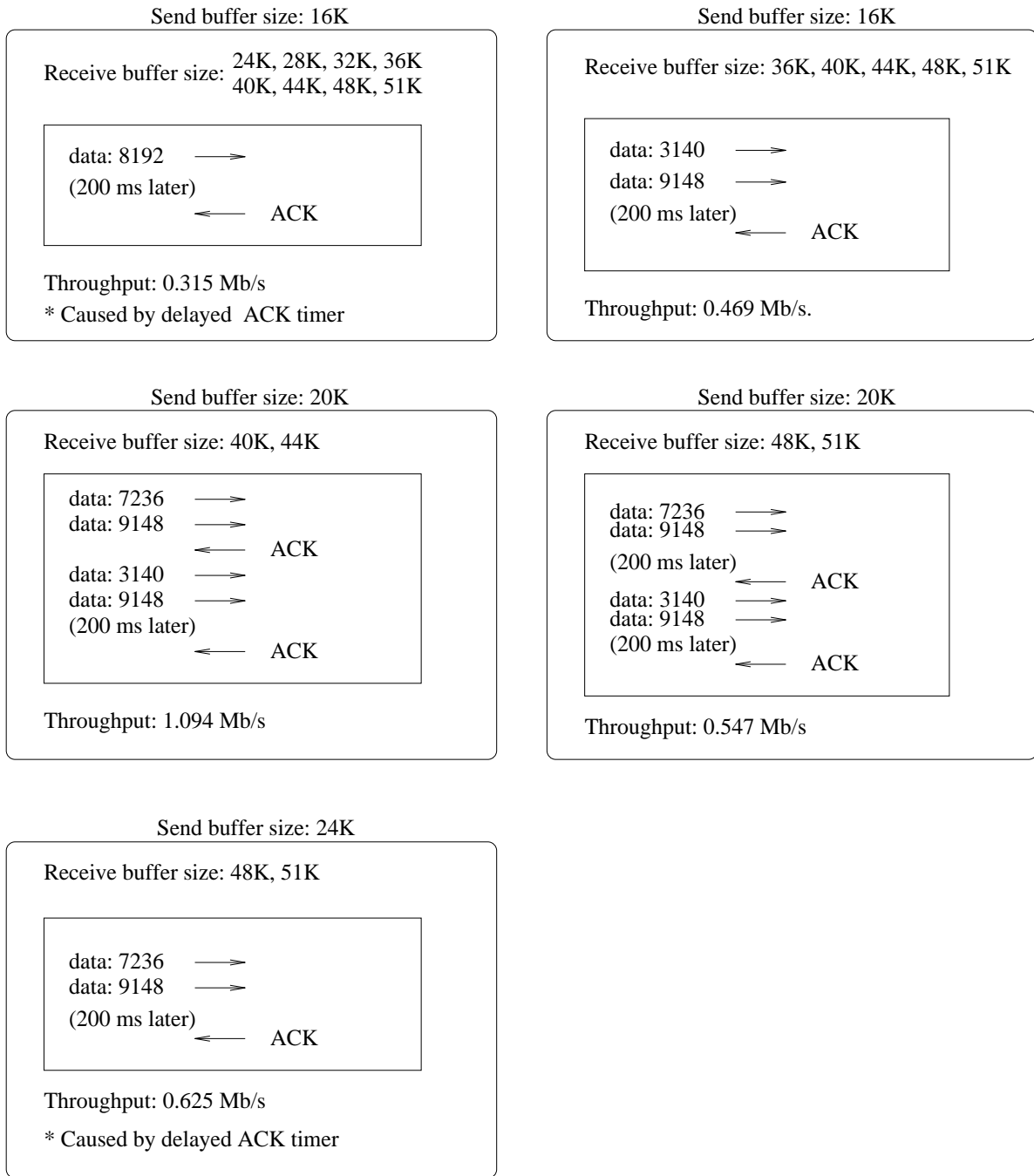


Figure 7: Summary of the repeated circular-wait behavior that decreases TCP throughput dramatically

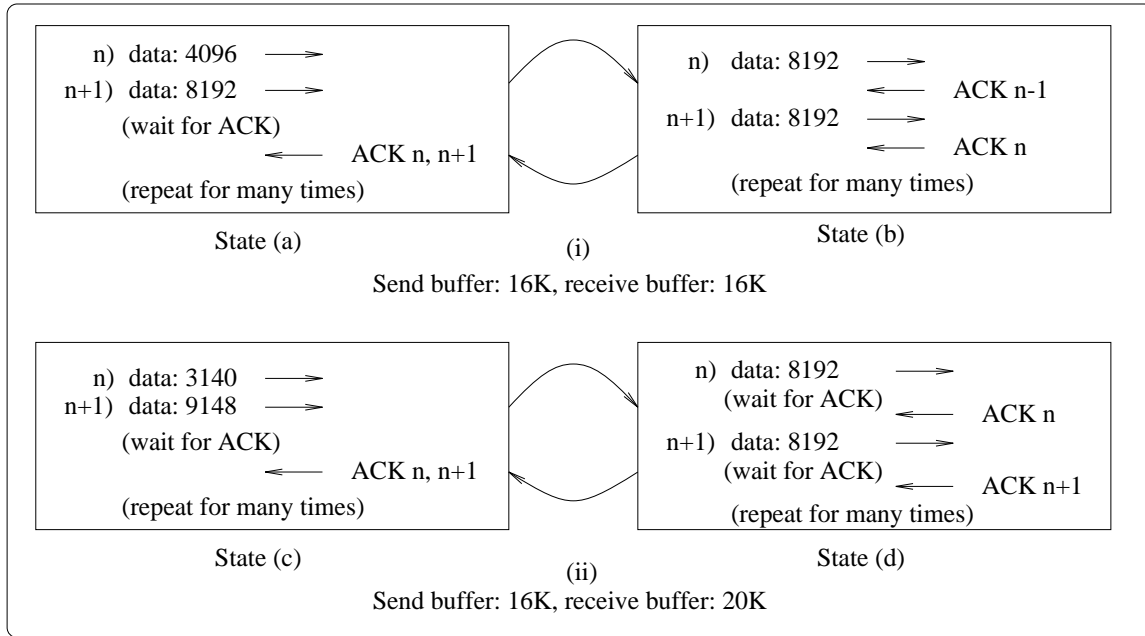


Figure 8: Illustration of the main states observed during a data transfer

4.2 Counter-intuitive Behavior

Table 1 shows that a sending SunOS 4.1.1 TCP with 16K buffer achieves better performance if a receiver *reduces* its buffer size from 20K to 16K. The main reason for improved throughput arises because SunOS 4.1.1 TCP uses the following condition, in addition to conditions *S1* and *S2*, to determine when to send a data segment:

S3: Let D be the amount of data to be transmitted, U be the usable window, and L be $\min(D, U)$. If $L \geq \text{max_sndwnd}/2$, then transmit a segment with L octets of data, where max_sndwnd is the largest receive window the peer has offered.

Because SunOS 4.1.1 TCP checks conditions *S1* and *S2* before condition *S3*, L is always less

than $1 * \text{MSS}$. When a receiver's buffer space is small (e.g., less than $2 * \text{MSS}$), condition *S3* allows TCP to send data segments that are smaller than $1 * \text{MSS}$. To see how SunOS 4.1.1 TCP uses condition *S3* to send data, consider a sending TCP with 16K buffer communicating with a receiving TCP with 16K buffer over an ATM path. Because the receiving TCP uses a 16K buffer, the largest receive window it offers to the peer is 16K (i.e., sender's max_sndwnd is 16K). Assuming that the receiver has buffer space at least 8K to accept the incoming data, and the sender's buffer contains more than 8K data ready to be transmitted, by applying condition *S3*, the sender can transmit an 8K data segment to the receiver because $8K \geq \text{max_sndwnd}/2$.

Figure 8 (i) illustrates the two main states observed during a data transfer between a sender with 16K buffer and a receiver with 16K buffer; Figure 8 (ii) illustrates the two main states observed during a data transfer between the same sender and a receiver with 20K buffer. States (a) and (c) show a repeated pattern in which the sender transmits two data segments, waits for an ACK, then receives an immediate ACK from the receiver. Notice that two segments in state (a) carry the same amount of data as the two segments in state (c). State (b) shows a steady flow of 8K data segments from the sender mixed with immediate ACKs from the receiver. State (d) shows a repeated pattern in which the sender transmits an 8K data segment, waits for an ACK, then receives an immediate ACK from the receiver.

Comparing states (b) and (d) reveals the cause for the observed throughput difference. State (b) occurs because sender and receiver both use a 16K buffer, hence the sender can apply condition *S3* to transmit two consecutive 8K data segments before waiting for an ACK. Furthermore, because the receiver consumes the incoming 8K data segments fast enough, it generates ACKs in time for the sender to send additional 8K segments. Thus, State (b) shows a continuous flow of 8K data segments from the sender. In State (d), because receiver's buffer space is larger than $2 * MSS$, the sender cannot apply condition *S3* to send data.

After sending an 8K segment, the sender must wait for an ACK before sending a buffered 8K segment because 8K is less than $1 * MSS$. The repeated waiting for an ACK before sending another data segment causes TCP throughput to be lower than the throughput measured in a continuous flow of the same sized data segments.

5 Summary

The results of the performance measurements show that TCP protocol software that performs well in a conventional LAN environment may suffer poor performance in a high-speed ATM LAN environment. The large MTU used by ATM creates a circular-wait situation not previously observed on a conventional LAN. The circular-wait, which can only be broken by the receiver's delayed ACK timer, creates a lockstep interaction in which the sender repeatedly experiences unnecessarily long delay before sending additional data. Thus, the network remains idle while data is waiting to be transmitted. As a result, TCP throughput decreases dramatically.

We observed a throughput anomaly in which an increase in the receiver's buffer size reduces throughput substantially. The anomaly is particularly annoying and surprising to users of a high-speed ATM LAN when they discover that the same *ftp* program they use to transfer files on a 10 Mb/s Ethernet can perform much worse on an ATM

connection with 100 Mb/s interface hardware.

Large MTU size, as compared with the TCP buffer sizes, and mismatched TCP send and receive buffer sizes are the main cause of the anomalous behavior. We conclude that a TCP can prevent such a behavior from occurring, regardless of the receiver buffer size, by using a send buffer size no smaller than $3 * MSS$. The solution is especially effective in a heterogeneous distributed environment because a sending TCP does not have control over the receive buffer size chosen by the peer. However, a sending TCP knows the TCP MSS and has control over its send buffer size.

Finally, it is worth noting that the significant throughput decrease observed in this paper is not restricted to ATM. Any environment with similar combinations of TCP send and receive buffer sizes can experience poor throughput. For example, two hosts attached to the same Ethernet running SunOS 4.1.1 will experience poor performance when the sending TCP uses a 3K octet send buffer and the receiving TCP uses a 6K octet receive buffer.

6 Trademarks

Fore Systems and ForeRunner are trademarks of Fore Systems, Incorporated. Sun, Sun-4, SPARCstation, and SunOS are trademarks of Sun Microsystems, Incorporated. UniForum is a registered trademark of UniForum Association.

7 Note

Apparently, Kjersti Moldeklev of Norwegian Telecom Research and Per Gunningberg of Swedish Institute of Computer Science observed the same anomalous behavior described in this paper.

References

- [1] R. Atkinson. RFC-1626: Default IP MTU for use over ATM AAL5. *Request For Comments*, May 1994. Internet Network Information Center.
- [2] E. Biagioni, E. Cooper, and R. Sansom. Designing a Practical ATM LAN. *IEEE Network*, pages 32–39, March 1993.
- [3] R. Braden. RFC-1122: Requirements for Internet Hosts – Communication Layers. *Request For Comments*, October 1989. Internet Network Information Center.
- [4] David D. Clark. RFC-813: Window and Acknowledgement Strategy in TCP. *Request For Comments*, July 1982. Internet Network Information Center.
- [5] D. E. Comer and J. C. Lin. TCP Buffering And Performance Over An ATM Network. Technical Report CSD-TR 94-026, Purdue University, West Lafayette, Indiana, March 1994.
- [6] Fore Systems, Incorporated. *ForeRunner SBA-100/-200 ATM SBus Adapter User's manual*, 1993.
- [7] The ATM Forum. *ATM User-Network Interface Specification Version 3.0*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
- [8] ITU-TS. Draft Recommendation I.150: B-ISDN ATM Functional Characteristics. *ITU-TS Study Group XVIII*, June 1992.
- [9] ITU-TS. Draft Recommendation I.361: B-ISDN ATM Layer Specification. *ITU-TS Study Group XVIII*, June 1992.

- [10] ITU-TS. Draft Recommendation I.363: B-ISDN AAL Specification. *ITU-TS Study Group XVIII*, Jan. 1993.
- [11] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88*, pages 314–328, Aug. 1988.
- [12] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1990.
- [13] John Nagle. RFC-896: Congestion Control in IP/TCP Internetworks. *Request For Comments*, January 1984. Internet Network Information Center.
- [14] C. Papadopoulos and G. M. Parulkar. Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(2), April 1993.
- [15] J. Postel. RFC-768: User Datagram Protocol. *Request For Comments*, Aug. 1980. Internet Network Information Center.
- [16] J. Postel. RFC-791: Internet Protocol. *Request For Comments*, Sept. 1981. Internet Network Information Center.
- [17] J. Postel. RFC-793: Transmission Control Protocol. *Request For Comments*, September 1981. Internet Network Information Center.
- [18] J. Postel and J. Reynolds. RFC-854: Telnet Protocol specification. *Request For Comments*, May 1983. Internet Network Information Center.
- [19] J. Postel and J. Reynolds. RFC-959: File Transfer Protocol. *Request For Comments*, Oct. 1985. Internet Network Information Center.