Introduction to TCP, in particular TCP-Reno

Teunis J. Ott

June 18, 1998

Abstract

This note is a short tutorial on TCP (Transport Control Protocol), the reliable transport protocol in the Internet. This note is meant as a quick and easy and incomplete introduction to the main aspects of TCP, in particular those aspects of most importance to the throughput performance of Internets. For a more complete treatment, see e.g. [25] and [26].

This note is one of the deliverables for the "TCP–ATM Interactions" project done at Bellcore in contract for Rome laboratories., Sept 1996 – Sept 1998. Other deliverables are a simulation tool with documentation, see [4], a tutorial on ATM with ABR, see [3], and a final report, see [5].

1 Introduction

This tutorial gives an overview of the most important aspects of the Internet TCP (Transmission Control Protocol). Currently, there are three flavors of TCP: TCP-Tahoe [10], (1990)), and TCP-Vegas [6]). Most implementations seem to be a mixture of Tahoe and Reno. TCP-Vegas is still under research. In our simulation we incorporated what we think

is a faithful implementation of a "typical" version of TCP/IP-Reno. However, TCP/IP tends to be different from manufacturer to manufacturer. We therefore do not expect that any version of TCP/IP commercially available will be identical to the version in our simulation.

This note is one of the deliverables for the "TCP–ATM Interactions" project done at Bellcore in contract for Rome laboratories., Sept 1996 – Sept 1998. Other deliverables are a simulation tool with documentation, see [4], a tutorial on ATM with ABR, see [3], and a final report, see [5]. This note is a tutorial on TCP as it is supposed to be according to IETF standards, but occasional references will be made to TCP as it is implemented in our simulation, or to simulation runs reported in [5].

In all forms of TCP, the destination sends acknowledgements for packets that arrive successfully. It is not the packet successfully received, but the packet (actually: byte) next expected that is announced. The bytes in the stream transported are numbered. If a packet arrives at the destination and (after the arrival) all bytes up to byte arr have arrived, but byte $arr + 1$ has not arrived yet, the destination will send the ACK "expect byte $arr + 1$". The acknowledgements may piggyback on top of data packets, and a single message may (implicitly) acknowledge bytes from a number of packets. In the simulation runs reported in our final report [5] there are no data packets in the opposite direction to piggyback on. "Unexpected" packets (packets "out of sequence") are individually acknowledged, while "Expected" packets (packets "in sequence") are either acknowledged at the next 200 msec timer (this is the "delayed acknowledgement" feature), or as soon as there are two packets to be acknowledged. As a result, there is usually one ACK packet for every two data packets. In TCP-Tahoe, packet loss is detected by maintaining a timer based on an estimate of the round-trip time RTT. TCP-Reno also maintains a timer, but attempts to primarily detect packet loss *before* timer expiration. Our simulation mimics TCP-Reno. To explain Reno, it is convenient to

first explain Tahoe. Both Tahoe and Reno have a dynamically changing "Window" W, roughly the maximum number of unacknowledged packets a source can have outstanding. The following simplified description of TCP Tahoe and TCP Reno has been adapted from Lakshman and Madhow [16].

2 Description of TCP-Tahoe:

The algorithm followed by each connection has two parameters, current window size W and threshold Θ, which are updated as follows:

1. After most acknowledgements (but not after duplicate acknowledgements, see next section):
   if $W < \Theta$, set $W = W + 1$; **Slow-Start Phase**
   else set $W = W + 1/\lfloor W \rfloor$. **Congestion-Avoidance Phase**
   ($\lfloor x \rfloor$ is the integer part of x)

2. After a packet loss is detected (via timer expiration):
   first set $\Theta = W/2$, then set $W = 1$.

The algorithm typically evolves as follows (although, as described in [16] the evolution can be somewhat different if the bottleneck buffer is small): when packet loss is detected, the window is reduced to one. In the slow start phase that follows, the window grows rapidly for every successfully acknowledged packet until it reaches half the window size attained before the last packet loss. The algorithm then switches to the congestion avoidance phase, probing for extra bandwidth by incrementing the window size by one for every window's worth of of acknowledged packets. This growth continues until another packet loss is detected, at which time another cycle begins.

2.1 Description of TCP-Reno:

Once a packet has been lost, until the loss has been repaired, all later packets arriving at the destination are "unexpected" packets and cause an ACK packet "expect x", where x is the sequence number of the first data byte in the missing packet. Hence, a single packet loss can be detected at the source by consecutive acknowledgements having the same "next expected number". Such acknowledgement packets are called "duplicate acknowledgement". In Reno, the source retransmits the apparently lost packet after the number of such repeated acknowledgements exceeds a threshold. This is called the "Fast-Retransmit Feature." At the same time the algorithm halves the window W. Also, in order to prevent a burst of packets from being transmitted when the retransmission is finally acknowledged, as soon as sufficiently many more duplicate ACKs for the same packet have been received the algorithm temporarily permits the source to transmit a new packet for every new duplicate acknowledgement until the "next expected" number in the acknowledgements advances. While these subtleties are essential for the performance of the algorithm, (see [11]) and have been implemented in our simulation, the following simplified description is adequate for conveying and understanding of the algorithm's behavior.

1. After every non-duplicate acknowledgement, the algorithm works as before:
   if $W < \Theta$, set $W = W + 1$; **Slow Start Phase**
   else set $W = 1 + 1/\lfloor W \rfloor$. **Congestion-Avoidance Phase**

2. When the number of duplicated acknowledgements reaches a threshold (in our simulation that threshold is three),
   retransmit "next expected packet," i.e., the apparently lost packet,
   set $\Theta = W/2$, then set $W = \Theta$ (i.e. halve the window);
   resume congestion avoidance using the new window once retransmission is acknowledged.

3. Upon timer expiration, the algorithm goes into slow start as in Tahoe: first set $\Theta = W/2$, then set $W = 1$.

The implementation in our simulation is equivalent to the explanation above but follows [25], in that after the third duplicate acknowledgement it enters the Fast-Recovery Stage during which the variable *cwnd* temporarily has a different interpretation.

In Reno, loss is usually detected early through duplicated acknowledgements, and instead of dropping the window to one, the window is halved.

It is worth relating our nomenclature to that used in standard TCP code (see for example [19]): W is usually referred to as the congestion window *cwnd*, and Θ is denoted as *ssthresh*. There also is an upper limit on the window size which is dictated by the receiver and which is denoted by *maxwnd*. *cwnd*, *ssthresh* and *maxwnd* are actually expressed in bytes, not packets, and through most of the runs we are reporting on in the final report [5] we chose *maxwnd* equal to 512Kbytes (524,288 bytes).

Many researchers have observed that both Tahoe and Reno (without ABR) have the potential for "Phasing": as long as no congestion is present, all TCP connections passing through a potential bottleneck buffer keep increasing their Window sizes (unless they reach the *maxwnd* level). At some point, the sum of the rates gets high enough to cause backup of the traffic in an output port, and some time later again this output port buffer fills and packets or cells are lost. Since the sources of the traffic do not notice the congestion until either duplicate acknowledgments start coming back or time-out occurs, they in fact keep sending at the high rate for some time after loss starts occurring. The result is that a single congestion episode leads to loss for many sources, who then start throttling their output all at the same time. In addition, the fact that the sources keep sending at the original high rate for some time after the congestion started increases the seriousness of the congestion episode, and increases the risk that some sources will lose multiple packets.

Depending on the type and specific implementation of TCP in that source, this may lead to much stronger throttling, thus increasing the "phasing" effect.

A possible solution to this problem is "RED" (Random Early Detection, often called Random Early Discard or Random Early Drop) [9]. In RED the bottleneck switch or router preemptively drops a carefully chosen streams of packets (or cells), in such a way that sources never see loss of multiple packets in quick succession, and the loss-rate is carefully tuned to keep the sources sending packets at (jointly) the right rate to prevent both buffer underflow and overflow.

A different but related method for improving TCP performance by improving the discard policy in routers and switches is the "Drop from Front" policy described in [17].

When congestion occurs in the segmentation buffer, the current simulation drops the entire frame, which is equivalent to a packet. Dropping of cells in the switch buffer is also from the tail. In 1996, we hope to experiment with "Random Early Detection" and "Drop from Front".

With ABR, if ABR indeed prevents cell loss in the shared bottleneck, it is likely that "phasing" will be greatly reduced or eliminated. In its place, we have the problem of overflow of the segmentation buffer.

As briefly discussed in Section 3 of this paper, TCP Reno runs into potential problems when a sizable number of packets from the window gets lost. TCP Vegas attempts to alleviate this by (among other techniques) attempting not to halve the *cwnd* more than once during one congestion episode. "Selective Acknowledgement" is a new feature of Reno currently being discussed in the IETF and has the same goal of providing robustness under loss of multiple packets.

3 Timers in TCP

As mentioned above, TCP uses timers to determine that a packet has timed out. Following the lead of (among others) [25] and [26] we will explain timers by a two-pass procedure. The first pass gives a approximate and easy to understand description. The second pass is correct.

3.1 Fine Grained Timers

The timing in TCP has two components. The first component produces estimates for the current average of the round trip time (RTT_{est}) and the current standard deviation of the round trip time (SD_{est}). The second component uses these estimates for average and standard deviation to determine when a sent but unacknowledged packet has been underway long enough that it must be assumed lost, and must be re-transmitted.

For the first mechanism, there are so called *timed* packets. There is at most one timed packet at a time. When there is no timed packet, under most circumstances the first packet to be sent becomes a timed packet. This means a record of its departure time is kept. When the packet is acknowledged the arrival time of the acknowledgement packet is compared with the departure time of the packet, and the difference is the measured round trip time (RTT_{meas}). RTT_{est} and SD_{est} are now updated, approximately using the rules

$$delta = RTT_{meas} - RTT_{est}, \tag{3.1}$$

$$RTT_{est} = \frac{7}{8}RTT_{est} + \frac{1}{8}RTT_{meas}, \tag{3.2}$$

$$SD_{est} = \frac{3}{4}SD_{est} + \frac{1}{4}|delta|. \tag{3.3}$$

At this point there no longer is a timed packet, so (under most ciscumstances) the next packet to be sent becomes the new timed packet.

The TCP source also maintains and entity called RTO (Retransmit Time Out). when a "good" acknowledgement packet (i.e. non–duplicate acknowledgement) arrives at the source, a time-out event is set for RTO later. In this situation RTO is computed as

$$RTO = RTT_{est} + 4 \times SD_{est}. \tag{3.4}$$

Below we will see that actually RTO has a minimum value of 500 msec. (TCP in our simulation tool actually has a minimal RTO value of 750 msec, this because we use fine–grained timers but still want to emulate the behavior of coarse grained timers as used in the standards, see below.) If by the time the "RTO" timer expires no other "good" acknowledgement has arrived a time-out event occurs. This means that the oldest unacknowledged packet (the one of which the first byte is the first unacknowledged byte) is retransmitted, and a new RTO is computed equal to twice the old one (with a maximum of 64 seconds). This new RTO is used to set the next time-out event. At a time-out event also the *cwnd* is reset to 1 MSS (Maximum Segment Size), and the source enters "slow start", etc.

3.2 Coarse Grained Timers

The floating point arithmetic featured in (3.1) - (3.4) would, certainly in 1980's computers, be a performance drain, and setting timers and waiting for them to expire would probably be even worse. The actual mechanism uses only integer arithmetic and does not set timers. Also, the divisions and multiplications that occur are actually shift operations.

In the first place, the system has 500 msec "clock" which once every 500 msec causes a clock-event.

The system maintains a variable called t_rtt (int) which is zero as long as there is no timed packet. When a timed packet is created, t_rtt is set to 1, and as long as $t\_rtt > 1$, t_rtt is increased by one at every clock event of the 500 msec clock. Thus, when a timed packet is acknowledged, $(t\_rtt - 1) \times 500$ msec is a plausible (though very coarse) estimate for its round trip time.

The system also maintains a variable t_srtt (int) which represents the estimated round trip time. t_srtt counts in units of $500/8 = 62.5$ msec, therefore one unit in t_rtt corresponds with 8 units in t_srtt. The system also maintains a variable t_rttvar (int) which represents the estimated standard deviation. t_rttvar counts in units of $500/4 = 125$ msec, therefore one unit in t_rtt corresponds with 4 units in t_rttvar.

When a timed packet is acknowledged, t_srtt and t_rttvar are updated. There is an auxiliary variable rtt which is set equal to the "returning" value of t_rtt:

$$rtt = t\_rtt;$$

$$delta = rtt - 1 - \frac{t\_srtt}{8};$$

$$t\_srtt = t\_srtt + delta;$$

$$\text{if } t\_srtt \leq 0 \; \{t\_srtt = 1;\}$$

$$delta = |delta|;$$

$$delta = delta - \frac{t\_rttvar}{4};$$

$$t\_rttvar = t\_rttvar + delta;$$

$$\text{if } t\_rttvar \leq 0 \; \{t\_rttvar = 1;\}$$

This completes the update. The reader can verify it somewhat corresponds with (3.1) – (3.3). Note that the estimated round trip time is always at least 62.5 msec, and the estimated standard deviation always is at least 125 msec.

There is a variable t_rxtcur (int) which (by first approximation) is the new value of the time-out period, in 500 msec units. It is computed at the end of the update above:

$$t\_rxtcur = \frac{t\_srtt}{8} + t\_rttvar;$$

The reader can verify this somewhat corresponds with (3.4).

Next, t_rtt is reset to zero (until the next packet is transmitted there is no timed packet).

There is a counter $t\_timer[tcp\_rexmt]$ (int, this is the tcp re-transmit timer). The reason it is part of an array is that there is a number of timers (the delayed ack timer, etc). When there is no unacknowledged packet outstanding, $t\_timer[tcp\_rexmt] = 0$. When a packet is sent while $t\_timer[tcp\_rexmt] = 0$, $t\_timer[tcp\_rexmt]$ is (usually) set to the new time-out period length:

$$t\_timer[tcp\_rexmt] = min(128, max(2, t\_rxtcur)); \tag{3.5}$$

When a good acknowledgement arrives at the source, $t\_timer[tcp\_rexmt]$ is either reset to zero (if there are no more unacknowledged packets outstanding) or reset to the same value as in (3.5), if there is still at least one unacknowledged packet outstanding.

At every tick of the 500 msec clock: If $t\_timer[tcp\_rexmt]$ is zero nothing is done. If $t\_timer[tcp\_rexmt] > 0$, $t\_timer[tcp\_rexmt]$ is decreased by one. If that makes $t\_timer[tcp\_rexmt]$ zero, there is a time-out: The oldest unacknowledged packet is retransmitted, t_rxtcur is doubled, $t\_timer[tcp\_rexmt]$ is set as in (3.5) (with the new value of t_rxtcur), etc. If the new value of $t\_timer[tcp\_rexmt]$ was still positive, no further action is taken.

We see that after a "good acknowledgement" at least two clockticks of the 500 msec clock must occur before a time-out can occur. In average this means a delay of 750 msec. So, as long as there is a "good acknowledgement" at least once every 500 msec, no time-out

will occur. If after a "good acknowledgement" 128 clockticks occur (64 seconds!) before the next "good acknowledgement", a time-out always occurs, independent of estimated round trip time or standard deviation.

References

[1] ANSI T1.646-1995 Broadband ISDN - Physical Layer Specification for User-Network Interfaces including DS1/ATM.

[2] ATM Forum Draft Standard on Traffic Management, Version 4.0, April 1996. ATM Forum/af-tm-0056.000.

[3] Ott, T.J. The Available Bit Rate Service Category in ATM. (Nov 1997). This is a tutorial on ABR written for Rome Laboratories, and delivered to Rome Laboratories in November 1997.

[4] Wong, Larry H. Software User Manual, Item No. A006. (Sept 1998) This is the Manual for the Simulation Tool delivered to Rome Laboratories. Drafts were delivered in July 1997, Dec 1997 and in Jan 1998. The Sept 1998 version will be virtually unchanged.

[5] Ott, T.J., Burns, J.E., and Wong, L.H. (1998) TCP over ATM: a Simulation Study. Final report for Rome Laboratories Contract F30602–96–C–0260. September 1998.

[6] Brakmo, L.S. O'Malley, S.O. and Peterson, L.L. (1994) TCP Vegas: New Techniques for Congestion Detection and Avoidance. Proc. ACM SIGCOMM'94.

[7] Floyd, S. (1991) Connections with Multiple Congested Gateways in Packet-Switched Networks, Part 1: One-way traffic. Computer Communications Review, vol 21 no 5 pp 30-47.

[8] Floyd, S. and Jacobson, V. (1991) On Traffic Phase Effects in Packet-Switched Gateways. Internetworking: Research and Experience, vol 3 no 3 pp 115-156. (An earlier version of this paper appeared in Computer Communications Review, vol 21 no 2, 1991)

[9] Floyd, S. and Jacobson, V. (1993) Random Early Detection gateways for congestion avoidance. IEEE/ACM Transactions of Networking, vol 1 no 4 pp 397 - 413.

[10] Jacobson, V. (1988) Congestion Avoidance and Control. Proc ACM SIGCOMM'88 pp 314-329.

[11] Jacobson,V. (1990a) Modified TCP Congestion Avoidance Algorithm. Message to end2end interest mailing list, April 1990.

[12] Jacobson, V. (1990b) Berkeley TCP Evolution from 4.3 Tahoe to 4.3 Reno. Proc. of the 18th Internet Engineering Task Force. Vancouver, Aug. 1990.

[13] Jacobson, V. Braden, R. and Borman, D. (1992) TCP extensions for High Performance, IETF RFC 1323.

[14] Kalampoukas, L., Varma,A. and Ramakrishnan, K.K. (1995) An efficient rate allocation algorithm for ATM networks providing max-min fairness. ATM Forum Contribution. Orlando, Fla, June 1995.

[15] Kalampoukas, L., Varma,A. and Ramakrishnan, K.K. (1995) Examination of the TM Source Behavior with an Efficient Switch Rate Allocation Algorithm (June 1995), ATM Forum/95-0767,

[16] Lakshman, T.V. and Madhow, U. (1994) Performance Analysis of widow-based flow control using TCP/IP: the effect of high bandwidth-delay products and random loss. IFIP Transactions C-26, High performance Networking V, pp 135-150. Also appeared as Bellcore TM-24168 under the title "Window-based Congestion Control for Networks with high badwidth-delay products and random loss: a study of TCP/IP performance.

[17] Lakshman, T.V., Neidhardt, A, and Ott, T.J. (1995) The Drop from Front Strategy in TCP and in TCP over ATM. Bellcore TM-25100. Also to appear in the Proceedings of Infocom '96.

[18] Romanow, A. and Floyd, S. (1994) Dynamics of TCP Traffic over ATM Networks. Proc. ACM SIGCOMM'94, pp 79-88.

[19] Shenker, S., Zhang, L. and Clark, D.D. (1990) Some observations on the dynamics of a Congestion Control Algorithm. Computer Communications Review, pp 30-39, Oct 1990.

[20] Zhang, L. (1989) A new architecture for packet switching network protocols. PhD dissertation, MIT.

[21] Zhang, L, Shenker, S. and Clark, D.D. (1991) Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic. Proc. ACM SIGCOMM'91, pp 133-147.

[22] Mathis, M., Semke, J. Mahdavi, J. and Ott, T.J. (1997) The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communications Review 27* (3), pp 67 - 82 (July 1997).

[23] Ott, T.J., Kemperman, J.H.B., and Mathis, M. (1996) The Stationary Behavior of Idealized TCP Congestion Behavior.
ftp://ftp.bellcore.com/pub/tjo/TCPwindow.ps

[24] Van Jacobson. Congestion avoidance and control, *Proceedings of ACM SIGCOMM '88*, August 1988.

[25] W. Stevens, *TCP/IP Illustrated*, volume 1, Addison-Wesley, Reading MA, 1994.

[26] G.R. Wright W.R. Stevens *TCP/IP Illustrated*, volume 2, Addison-Wesley, Reading MA, 1994.

[27] Kevin Fall and Sally Floyd, Simulations-based comparisons of tahoe, reno and SACK TCP, *Proceedings of ACM SIGCOMM '96*, May 1996.

[28] Janey C. Hoe, Improving the start-up behavior of a congestion control scheme for TCP, *Proceedings of ACM SIGCOMM '96*, August 1996.

[29] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson, TCP vegas: New techniques for congestion detection and avoidance, *Proceedings of ACM SIGCOMM '94*, August 1994.

[30] Lawrence S. Brakmo and Larry L. Peterson, Performance problems in BSD4.4 TCP, *Proceedings of ACM SIGCOMM '95*, October 1995.

[31] Sally Floyd, Connections with Multiple Congested Gateways in Packet-Switched Networks Part I: One Way Traffic. *CCR* **21** no 5 pp 30 - 47.

[32] Sally Floyd, TCP and successive fast retransmits, February 1995, Obtain via ftp://ftp.ee.lbl.gov/papers/fastretrans.ps.

[33] Sally Floyd and Van Jacobson, Random early detection gateways for congestion avoidance, *IEEE/ACM Transactions on Networking*, August 1993.

[34] Sally Floyd, TCP and explicit congestion notification, *ACM CCR*, 24(5), October 1994.

[35] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow, TCP selective acknowledgement options, May 1996, Internet Draft ("work in progress") draft-ietf-tcplw-sack-02.txt.

[36] Matthew Mathis, Jamshid Mahdavi, Forward Acknowledgment: Refining TCP Congestion Control, *Proceedings of ACM SIGCOMM '96*, August 1996.

[37] Lakshman, T.V., and Madhow, U. (1997) The Performance of TCP/IP for Networks with high Bandwidth-Delay products and random loss. *Trans of Netw* 1997.

[38] Lakshman, T.V., Madhow, U. and Suter, B. (1997) Window-based error recovery and flow control with a slow acknowledgement channel: a study of TCP/IP performance *Infocom '97*.