Why TCP Timers Don't Work Well

Lixia Zhang
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Repeated observation of TCP retransmission timer problems stimulated investigation into the roles and limitations of timers. Timers are indispensable tools in building up reliable distributed systems. However, as the experience with the TCP retransmission timer has shown, timers have intrinsic limitations in offering optimal performance. Any timeout based action is a guess based on incomplete information, and as such is bound to be non-optimal. We conclude that, if we aim at high performance, we should use external events as a first line of defense against failures, and depend on timers only in cases where external notification has failed.

1. Overview

In computer communication networks a *timer* is a failure detection mechanism, normally used to decide when to retransmit a lost packet, or when to abandon a broken connection. Timers have been employed in all network protocols that offer reliable services. They seem to play an indispensable role. However, even with many years of experience, we are still not able to make timers work as well as we would like.

The Transmission Control Protocol (TCP) [8] is intended for use as a highly reliable host-to-host protocol in packet-switched computer networks, and in interconnected systems of such networks. TCP has been widely implemented and used over the years. Repeated observations of TCP timer problems stimulated our investigation into further understanding of the following questions:

- Is a timer really indispensable in network protocols?
- What roles should a timer play? What are its limitations?
- How should we use it?

The basic conclusions we draw are that timers are indispensable in building reliable distributed systems; yet their limitations need to be fully identified. In retrospect, we see that many of the problems encountered in using a timer are in fact due to misunderstanding of its limitations. Although the following discussions relate specifically to phenomena and problems occurring in TCP, the conclusions, we believe, apply to the roles of timers in similar protocols, such as the ISO Transport Protocol, and in all distributed systems.

The next section explains the necessity of timers in distributed systems in general, and in network protocols in particular. Section 3 is a review of previous work and experience with TCP timer. Section 4 explores the intrinsic limitations of a timer. With a better understanding of the limitations, section 5 suggests some heuristic rules in using timers, and uses the timing algorithm of NETBLT (NETwork BLock Transfer) [2], a bulk data transfer protocol, to give an example. The last section is a summary of the work.

2. Why a Timer?

A computer network is a distributed system. One of the advantages of distributed systems is that there is no *fate-sharing* among individual autonomous components in the system, i.e. they fail independently. This non-fate-sharing feature is achieved by coupling the components only through data communications channels. Consequently, individual components in a distributed system can only "hear" from each other through the communication channels, but cannot directly observe the existence or functioning of others and their running states. To coordinate with each other, they have two ways to detect external state changes or failures:

1. By external reports. For instances, upon the arrival of an acknowledgment, the data sender knows that the data sent have been successfully received; when an ARPANET host tries to communicate with another not-running host, the network will respond with a "remote host dead" message.

2. By local detection, e.g. using a retransmission timer to detect packet losses.

In this paper *failure* has a very general definition: it may simply refer to the failure of an intended function, as well as to a machine crash or the breaking of a communication channel. Later on we will see that external reports are a better way to do failure detection and recovery. For the following reasons, however, local detection is always needed:

- Not all external changes or failures can be reported. For example, if a receiver detects an incoming packet with a header checksum error, the source address part may have been damaged, hence the sender cannot be identified. The receiver will not be able to notify the sender to retransmit the packet.

- The reporting system may fail itself, e.g. an acknowledgment may get lost.

Therefore, to achieve sufficient self-protection in a distributed system, cautious users set up some form of local detection. So far, the only local detection tool available is a timer. This is not a coincidence. With no external information, *time* is the only tool that one can use to *estimate* external state changes. If one communicating end does not hear from the other end as it should within some reasonably long time period, it *assumes* that something must have gone wrong, either within the communication network or at the remote site. For example, the sending host of a TCP connection uses a timer to detect packet loss, so does an ARPANET IMP; during the absence of data traffic, ARPANET IMPs regularly talk to each other, and a neighbor IMP will be declared down if it has been silent for a certain time period. A timer is a *must* for any player in a distributed system.

3. Previous Experience and Work with TCP Timer

A timer is an alarm clock which goes off after a specified *timeout* period. The usual goal of a timer algorithm is to dynamically adjust the timeout value to approach an ideal where the timer is triggered *immediately* and *only* upon a real failure. In their desire to achieve good performance, all timer algorithms try to balance between two conflicting goals:

1. speeding up failure detection, and

2. minimizing false alarms, i.e. minimizing the incidents of the timer going off prematurely when no real failure has occurred.

TCP uses timers to detect packet losses (the retransmission timer) and connection breaks (the death timer). Since connection breaks happen rarely, and hosts usually are willing to try for a long time before finally giving up, the death timer is often set to a large value. This is not the case, however, for TCP retransmission timers. In the middle of a session, it is undesirable for a client to wait for a few minutes to recover a transmission error. TCP took the approach of setting the retransmission timer by dynamically estimating the *Round Trip Time* (RTT) between the two communicating entities. In this section we first introduce the TCP's adaptive retransmission timer algorithm, then discuss its problems.

3.1. TCP Retransmission Timer Algorithm

Due to the variability of the networks that compose an internetwork system, the TCP retransmission timer (TCP timer for short) is determined dynamically for each connection. TCP measures the RTT for each data segment transfer, and computes a *Smoothed Round Trip Time* (SRTT):

$$SRTT = \alpha \times SRTT + (1 - \alpha) \times RTT$$

Based on SRTT, it then computes the *Retransmission TimeOut* value (RTO):

$$RTO = min\,\{\,Ubound,\ max\,(Lbound,\ \beta \times SRTT)\,\}$$

Where *Ubound* and *Lbound* are the upper and lower bounds on the timeout value; α is a smoothing factor, and β is a variance factor. In real implementations, Ubound and Lbound values are assigned empirically as a loose limitation on the timer's value. Recommended values of α and β are $0.8 \sim 0.9$, and $1.3 \sim 2$, respectively. Different α and β values have been experimented with, as described below.

3.2. Problems with TCP timer

Over the years of running TCP in the ARPA Internet, many problems associated with the TCP timer have been encountered. Understanding them requires that we understand the running environment of TCP. The ARPA Internet is a heterogeneous network complex which connects together a large number of diverse networks: high speed LANs, narrow bandwidth dialup lines, long delay satellite channels, reliable long haul networks, etc., with the communication bandwidths and delays varying between networks by orders of magnitude. The data carrier over this complex is IP [9], a datagram protocol offering a "best effort", but not reliable, delivery service. Packet loss is not uncommon, especially when the network gets heavily loaded, because IP's only defensive tool is dropping packets, relying on the end-to-end transport protocols to recover the loss when necessary. TCP runs on top of IP. TCP does not have a negative-acknowledgment mechanism to report transmission errors; all data errors, including losses, rely on the sender's retransmission timer to trigger the recovery. Such an environment makes an accurate setting of the TCP timer necessary for good performance.

The first difficulty in using the TCP timer is to choose an initial value for the SRTT. Before the first data exchange between the two communicating entities, there is no information available to the sender as to how long the round trip time will be, assuming the destination address does not convey network topological implications. The current approach is to pick an arbitrary value, say 3 seconds, in the hope that it will quickly converge to the right value through the adaptive algorithm. It is often the case that this arbitrarily chosen value is too small, or too large, compared to the round trip time of the intended connection; so will be the initial RTO value. As a result, TCP will either retransmit superfluously, or wait for a long time before retransmitting if the first packet is lost. Also, the convergence is slow[1]. When the initial value is too small, excessive retransmissions may cause a temporary network congestion before the timer gets a chance to converge to the correct value. This problem has been observed many times in the ARPA Internet. On the other hand, a large initial value means a possible slow start to the client, but does no damage to the network as a whole otherwise.

A second problem is how to measure the round trip time. This measurement is, of course, trivial when there is no packet loss. When packet losses occur, however, getting correct RTT measurements is impossible, because when an acknowledgment is received after n retransmissions, the data sender cannot tell which of the n+1 copies sent is being acknowledged. This problem directly affects the computation result of the SRTT value. A case analysis due to Dave Clark (see Appendix-I) shows that TCP cannot compute the SRTT value correctly when packet retransmission occurs. Since the SRTT is used solely for packet loss recovery purposes, this problem is particularly unfortunate: the SRTT is not used when there is no loss; when it is to be used, it cannot be correct.

The next problem in using the TCP timer is how to set RTO values. Wrong SRTT values lead to wrong RTO values. When the RTO is too small, the effective network throughput is reduced by too many duplicate packets. When the RTO is too large, network clients suffer from needless long waits before retransmitting lost packets. Most TCP implementations, as well as the TCP experiment discussed shortly, measure the RTT from the first sending. When the network is lightly loaded, packet loss is random and negligible, occasional inaccurate RTT measurements do not cause a big problem, because the SRTT value gradually approaches the true round trip time despite some momentary fluctuation, and because retransmissions are rare, so using a larger than needed RTO value does not degrade performance noticeably. However, when network

[1] Here are examples showing the rate of convergence when the initial SRTT (S_0) is improperly chosen. We choose α=0.85, $\beta = 2$, and assume $S_0 = 3\ \mathit{seconds}$, all measured $\mathit{RTT\ values} = 1\ \mathit{second}$, no packet loss:

$$SRTT_n = \alpha^n \times S_0 + (1-\alpha) \times RTT \times (1-\alpha^n)/(1-\alpha)$$
$$SRTT_{19} = 1.1\ \mathit{sec}$$

Obviously, if we turn the values of S_0 and RTT's around so that $S_0 = 1\mathit{sec}$, all measured $\mathit{RTT\ values} = 3\ \mathit{sec}$, no packet loss.

$$SRTT_n = 2.9\ \mathit{seconds\ \ when\ \ } n = 19$$

This convergence speed is slow for short data sessions.

congestion occurs, packet losses tend to be frequent, which in turn causes the SRTT and hence the RTO to grow rapidly. This phenomenon was observed in a network test conducted at MIT-LCS: data packets were sent from one host on a 10 Mbps ringnet to another host on a 10 Mbps Ethernet through a gateway. The packet flood congested the gateway, causing many packets to be dropped. The RTO value grew quickly from several hundred milli-seconds to more than 2 minutes, causing the sender to wait too long before initiating the recovery. The same phenomenon was also observed in a network experiment at Digital Equipment Corporation [5].

Even assuming the SRTT value is a correct average of the round trip time, setting an accurate RTO value based on the SRTT is still difficult, due to the potentially large variance of the RTT. One source of the variance comes from the packet length effect. Whenever there are one or more narrow bandwidth channels on the route of a connection, the dominant component in the RTT will be the bit transfer delay over that line, which is proportional to the packet length. Packet lengths can easily vary by a factor larger than two, causing false timeouts. Another source is dynamic network routing: since IP is a datagram protocol, packets may theoretically be routed through different paths with different delays. Still another one is the delay at the receiving host: besides its packet processing delay, the host, for performance considerations, may prefer not to respond immediately after every packet arrival [1], contributing another factor to the RTT variance.

The above arguments show that the variance of network delay can easily go above the recommended value, $1.3 \sim 2$, of the variance factor β, even without considering the effect of the network traffic fluctuations[2]. Still another difficulty in setting an accurate RTO value is the inevitable phase delay between the measured RTT values and the current round trip time inside the network. A sudden change in path or network condition, say at time T_0, can result in a sudden increase in the round trip time. Packets sent after T_0 will bear a longer delay, say of D seconds. The measured RTT value, however, does not reflect this change until time $T_0 + D$. The value of D can be several seconds, or even tens of seconds, on a long path. Reflecting the change to the RTO setting takes even longer when a big α value is used. It was observed in a network simulation that, during this time period, the timer frequently went off and triggered superfluous retransmissions [10].

The last problem in using TCP timer is how to handle a timeout. If the TCP timer on an unacknowledged data segment S goes off, TCP implementations were recommended to retransmit only the packet containing S, not any subsequent packets that may be awaiting acknowledgment. In the real world, many possible events may result in a TCP timer going off when retransmission is unnecessary or infeasible. Consider the following cases:

1. S may not have left the host yet because of some lockup at lower layer. For example, the interface to the attached network is blocked.
2. The current timer value may be shorter than the fluctuating round trip time at that moment, causing a false alarm, e.g. a packet surge at some gateway made S have a much longer round trip time.
3. S was received correctly but its acknowledgment was damaged or lost.
4. S was dropped by some gateway due to congestion.
5. S was dropped due to transmission channel error.
6. The network partitioned or the destination host crashed.

In the first three cases, retransmission is unnecessary. Even for the next one, which does require retransmission, the existence of congestion implies care should be taken not to worsen the situation further. An immediate

[2] Here is an example showing the RTT variation due to traffic interference: assume a 9.6 Kbps link connects two LANs through two gateways, and a remote login session between two hosts on the two LANs. When there is no other traffic, the round trip time is small since telnet packets are usually small in size. Let us plug in some numbers and compute:

$$\begin{aligned}&\textit{If IP packet size (both directions)} = 50\,\textit{bytes},\\ &\quad \textit{gateway processing delay} = 5\,\textit{msec, and}\\ &\quad \textit{remote host processing delay} = 15\,\textit{msec},\\ &\textit{then } RTT = 5 \times 2 \times 2 + 15 + 50 \times 8 \times 2/9.6 < 120\,\textit{msec}\end{aligned}$$

The transfer delay on the LANs is small enough to ignore in the above computation. If now a full-size IP packet from another connection arrives at one of the gateways, it will take $576 \times 8/9.6 = 460$ msec to drain out the slow channel. The telnet packets following it will bear an RTT of $120 + 460 = 580\,msec$, almost 5 times as long as before.

If the packet size is sufficiently large to require fragmentation at gateways, the delay increase would even be worse than linear.

retransmission upon timeout is desirable only in case (5), but it is now being done in all the cases listed. On the other hand, more than one packet may be lost at once. The current strategy, which recovers only one packet per round trip time, may result in poor performance if the connection is over a channel with long delay such as a satellite link.

Too quick and too many retransmissions are often seen by people watching the daily network traffic. These retransmissions are considered one of the causes for network congestion. From above we see that there are several possible reasons for this phenomenon: SRTT initial values that are too small, delay variance estimation that is too low, poor RTT measurements that lead the SRTT to converge to wrong values, etc. On the other hand, complaints are often heard from the user side about slow network responses, which may well be due to RTO divergence. The two phenomena can even be related: if the initial RTO values of a few new connections are set too small, superfluous retransmissions can congest a gateway and cause packet losses on the new, as well as other established, connections. The losses on those running connections will in turn make their RTOs grow large and therefore the loss recovery will take a long time. If the congestion is severe, it may also cause TCP connections to appear to break, even there is no broken part in the network but merely a traffic jam due to IP's lack of control.

The above discussion might incorrectly be taken to mean that the robustness mechanism in TCP is not valid. In fact, the above problems are largely due to IP's deficiency in congestion control. TCP is intended to be a reliable end-to-end transport protocol; the TCP timer is designed to guarantee this reliability, under the assumption that data losses are random and rare, say with a $1 \sim 2\%$ loss ratio. As mentioned earlier, however, this assumption is invalidated by the fact that dropping packets is IP's primary way of handling congestion. TCP cannot help IP solve network congestion problems while still keeping good performance. Unfortunately, when the performance becomes too poor, it is no longer distinguishable from failure.

3.3. Previous Work with TCP Timer

In [6], Mills suggested that two values of α be used, with $\alpha_1 = 15/16$ when RTT < SRTT and $\alpha_2 = 3/4$ when RTT > SRTT. The effect is to make the algorithm more responsive to upward-going trends in packet round trip time and less responsive to downward-going trends. He then did some test runs by measuring the delay of ICMP echo/reply messages between many hosts, and concluded that, using the new α values instead of the recommended one, the results were better by several percent in most cases, and worse by several percent in a few cases. Notice, however, that the test was a lock/step process with at most one packet in the fly, and therefore did not suffer from SRTT divergence caused by consecutive losses. A simulation with the suggested α values showed that the SRTT goes up substantially in the case of consecutive losses [10], since in this case the measured RTT includes not only the round trip time but also the loss detection time[3]. Trying to adapt quickly to this wrong value simply makes the RTO diverge much faster.

An experiment was performed at MIT-LCS to try out another way to estimate RTO values without using the RTT, thereby sidestepping the problem with the RTT measurement. This experiment assumes that the delay distribution is a bell-shaped curve, treating lost packets as having infinite delay. The percentage of packet retransmissions, P, is used to adjust the RTO value. If P becomes bigger than some threshold, the RTO is increased, otherwise it is decreased. The original RTO value is chosen large, and intended to decrease gradually. This scheme did not work out well. The RTO value diverged for the same reason as in the TCP timer case: consecutive packet losses due to congestion resulted in a continuous growth of P, hence a continuous growth of the RTO, followed by a very slow recovery of the lost packets.

There are other studies on the timeout algorithm worth mentioning. Cooper, in designing a new retransmission timer algorithm for TFTP [3], pointed out that "the probability of a single packet being lost may be some constant P_1, but the probability that a second packet will be lost once a packet has already been lost is $P_2 \gg P_1$." He suggested that in case of retransmissions, the SRTT value should be increased by some empirical value (e.g. 2 seconds), instead of as a function of the RTT measurement. This approach may slow down, but does not prevent, RTO divergence. Work done by Morris [7] is similar to the MIT-LCS

[3] The simulation followed Mills approach of measuring the RTT from sending the first copy of data to receiving the acknowledgment.

experiment: under the assumption that the packet round trip time is a random variable of some known distribution function, optimal timeout values, in terms of minimizing the sum of the performance loss of false alarm and unnecessary waits, can be computed. In [4] Edge assumes that packet delays are random variables forming certain stochastic processes, and he determines the timeout value by estimating the mean and the variance of the measured delays.

These studies have their great merits. We also believe that TCP timer algorithm can be further improved, as a couple of suggestions will be made in the next section. However, we consider that several problems in using the TCP timer are more due to some intrinsic limitations of using timers than due to the specific algorithm used. The next section explains these limitations.

4. Intrinsic Limitations of Timers

4.1. TCP Timer Problem Revisited

In the previous section, we found two basic problems in using TCP timers: choosing a timeout value (RTO) and handling a timeout. Let us now look at each of the two again from a more general viewpoint.

4.1.1. Choosing a Value

No algorithm can magically compute an accurate RTO for each packet transfer. As mentioned above, the difficulties in choosing the initial SRTT value, in measuring the round trip time, and in setting the RTO value are all due to the same reason: lack of adequate information about the network topology and dynamics. Many factors involved in the round trip time are not currently known by the data sender. Given this problem, it seems that directly providing the needed information may help more in choosing a correct value than cleverly tuning an algorithm based on inadequate information, and that the network should make an effort to reduce the variance of the round trip time. An example of the former is to let the TCP timer stamp a unique ID number on every packet sent, and let the acknowledgment, which is triggered by receiving packet P, carry back the ID of P. This will fix the RTT measurement problem in packet retransmission cases. An example of the latter is to add an effective congestion control mechanism to IP; It will improve TCP performance more effectively than any tuning on the timer algorithm.

As shown in the previous section, the network delay is a random variable involved with many uncontrollable factors. Therefore even with further improvements, the TCP timer still should be set with a sufficient variance margin. This will have little effect on the performance, if the network does not drop many packets. On the other hand, we should not expect an optimal performance by using a timer. A timeout is a guess based on incomplete information, and as such is bound to be non-optimal. That a timer is triggered only and immediately upon a real failure, unfortunately, can only be an unachievable ideal by any timeout algorithms.

4.1.2. Handling the Timeout

The difficulty in setting the timer value is only half of the story. Since the timer is a failure detection tool, following a timeout there has to be a failure recovery. As discussed above, there are two ways to detect failures, external reports and timers. The two have different impacts on the recovery. Failure reports, assuming they carry correct messages, bring in explicit information of what went wrong. But a timeout, by itself, is merely a symptom which can have any of a large number of causes. For instance, upon receiving a "remote host dead" message, the local client can be informed to close the connection: while if a packet transfer has timed out five times, it is not clear whether this is caused by a temporary network congestion or a remote host crash. The fundamental problem in using the timer is that a timeout does not tell precisely what went wrong (or even whether there is anything gone wrong), so we cannot know with certainty what should be done in response. Assumptions have to be made when attempting to recover the unknown failure. The price to pay for this uncertainty is, again, non-optimal performance.

4.2. Timer's Roles and Limitations

The above shows that the TCP timer has intrinsic limitations, i.e. it does not have all the information available to achieve the good performance as we would desire. We consider this a common feature of any algorithms based on timeout. As we discussed in section 2, a timer is a local tool mandatory for achieving reliability in distributed systems. However, the necessity of a timer does not imply that we should, or have to, rely on it for everything. Timers should be used only when all other means fail.

In general distributed systems or applications, the problem of using timers to achieve good performance seems even more difficult than in TCP. For long TCP sessions, though the initial value of RTO may not be right it can be tuned by continuous measuring on the RTT. But for distributed applications running transactions with a few packet exchanges each, the real round trip time is not known to start with, and not enough data exchanges will occur for an arbitrarily chosen initial value to converge. For this kind of interactions, a timer alone cannot insure a good performance.

5. A Better Way to use Timers

5.1. Heuristic Rules

Since we have to pay the price of non-optimal performance whenever using a timer, the first advice in using timers is to rely on them as little as possible. This means that any abnormal situation should be resolved if possible, rather than turning it to a failure too easily and relying on timers to recover, and that any failure should be explicitly reported if possible. External reports are both faster and more accurate than using a timer in failure detection.

Secondly, try to get more information to help set a proper timeout value, and do not attempt to tighten the timer for a "better performance", unless it is based on the knowledge of the underlying system, because the gain in occasional faster detection by a tight timer may well be smaller than the loss due to false alarms.

Accepting the fact that the timer should be set loosely, if it is not feasible to waste the time when waiting for either a confirmation or a timeout, one way to improve the performance is to explore more concurrency by applying the knowledge of the specific applications.

5.2. An Example

Here we use NETBLT [2] as an example to show a better way to use timers. NETBLT was designed as a bulk data transfer protocol at MIT-LCS. It has achieved very good performance during the preliminary implementation test. More tests are yet to be performed over a wide range of network conditions, however. The reader should be warned, therefore, that the following discussions are more based on sound arguments than on actual experience.

NETBLT is another transport level protocol designed for transferring large quantities of data across the internet. Like TCP, it uses a timer to detect packet loss, but its data transmission timing scheme is drastically different from TCP's. The four major differences are described below.

First, NETBLT sets the retransmission timer at the receiving end, rather than at the sender as TCP does. When considering the state of a data transfer, it is the receiver that is more concerned with the transfer results, and that knows the state changes (correct reception of new data) first.

Second, NETBLT sets a retransmission timer on each block of data, which contains a large number of packets, instead of timing each packet. This allows the timeout value to be set more loosely to avoid false alarms, and still saves a lot of waiting time, because at worst the receiver waits only once to initiate the recovery cycle for all packet losses in a block of data. Additionally, from an implementation point of view, setting and canceling of timers are expensive operations in all systems; setting fewer timers certainly saves system overhead.

Thirdly, in case of packet loss, NETBLT does not wait for the timeout to trigger the recovery. Instead, as soon as the last packet in a block arrives, the receiver will check to see if any packets are missing; if so, it dallies for a short time period (to wait for possible out-of-order packets) and then informs the sender with a list of all missing packets in the block. The block retransmission timer is used to initiate the recovery only when the last packet in a block is lost.

Fourth, the retransmission timer value is computed from the transfer speed of the sender, rather than the measured network delay. Upon receiving the first packet of a block, the receiver starts the timer with the RTO value equal to the amount of time required to transfer the whole block of data (this time can be computed from the block length and the sender's speed), plus a variation margin. Therefore the timer does not suffer from the RTT measurement errors. Also, as a side effect of timing an entire block of data, delay variances on individual packets in the same block are likely to cancel out, hence a moderate variance value is expected to be sufficient.

Additionally, NETBLT provides for multiple data blocks being transmitted concurrently. When one block finishes transmitting its data and is waiting for the receiver's reply, the next block can start sending immediately, keeping the communication channel busy. NETBLT uses a rate-based flow control to coordinate the host data transfer speed and the network speed. When properly supported by the network, it will smooth out the transfer and avoid data accumulation inside the network, hence reduce network delay, delay variance, and packet loss caused by congestion.

In short, the main theme in NETBLT timing is to reduce the dependency on the timer to failure detections that cannot be detected by other means. High performance is achieved through using more information about the data transfer, exploring concurrency, and avoiding congestion. Of course the NETBLT protocol has its limitations, i.e. it is mainly for bulk data shipment. The approaches it employs in its timing algorithm, however, are expected to be generally applicable to other network protocols and distributed applications.

6. Conclusion

The purpose of this paper is to identify the importance of timers and their roles in distributed systems. A timer is an indispensable tool in building up reliable distributed systems. However, as the experience with the TCP timer has shown, it has intrinsic limitations in offering optimal performance. We should bear in mind these limitations in future protocol design. If we aim at high performance, we should use external events as a first line of defense against failures, and depend on timers only in cases where external notification has failed.

Acknowledgment

I wish to thank to Dave Clark, J. Noel Chiappa, Mike Greenwald, and Larry Allen for numerous enlightening discussions over the years. I am also grateful for the help of Dave Clark, J. Noel Chiappa, Jim Gibson, and Bob Baldwin during the writing of this paper. Finally, thanks to Bob Braden of USC-ISI for his valuable comments on an earlier version of the paper.

Appendix-I: RTT Measurement in Packet Retransmission Case

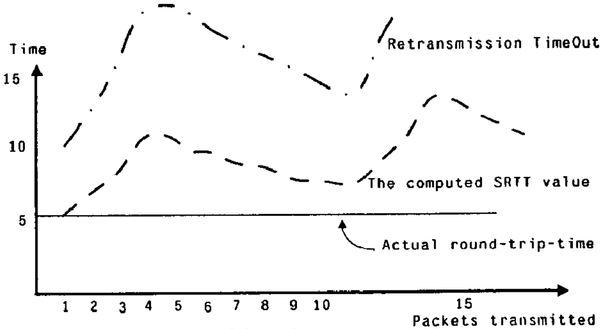
The following case analysis due to Dave Clark shows that, when packet retransmissions occur, there is an uncertainty in the RTT measurement, and therefore the SRTT value cannot be computed correctly. Assume an acknowledgment is received after retransmitting a packet n times,

1. If the RTT sample is taken as the elapsed time from sending the first copy of the packet to finally receiving the acknowledgment, the time period actually covers the loss detection time ($n \times RTO$) as well as the recovery time (the true round trip time) (assuming no false alarms, so the last retransmission is being acknowledged). Using this value to compute the SRTT and then the RTO, we see a loop in the computation:

$$\begin{aligned} RTO_i &= min\,\{Ubound, max\,(\beta \times SRTT_i,\ Lbound)\,\} \\ RTT_{i+1} &= n \times RTO_i + true\_RTT = n \times \beta \times SRTT_i^{*} + true\_RTT \\ SRTT_{i+1} &= \alpha \times SRTT_i + (1-\alpha) \times RTT_{i+1} \\ &= \underbrace{(\alpha \times SRTT_i + (1-\alpha) \times true\_RTT)}_{desired\ term} + \underbrace{(1-\alpha) \times n \times \beta \times SRTT_i}_{unwanted\ contributor} \end{aligned}$$

Therefore a single packet loss may cause a big jump in the SRTT value, and multiple losses in a row (a likely result of network congestion) will make the SRTT and RTO values grow until the RTO is bounded by Ubound.

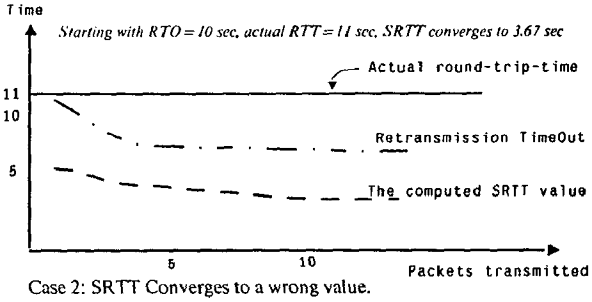
Consecutive 3 losses of packets 1-3 caused SRTT to double its original correct value of 5 sec; 10 successful transmissions brought it down to 7.3 sec; then another 3 losses soared it up to 13.6 sec.



Case 1: SRTT grows to Ubound.

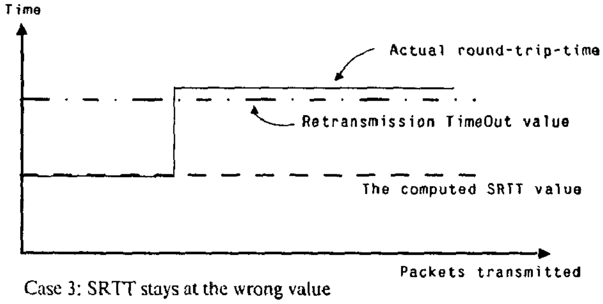
[*] Since Ubound and Lbound are wired-in constants in TCP implementations, they are usually chosen very loosely to suit order-of-magnitude differences of the RTTs in TCP's running environment. Therefore $RTO = \beta \times SRTT$, until SRTT soars too high or shrinks too small.

2. If the RTT is measured from sending the last copy to receiving the acknowledgment, the result will be a smaller value than the real round trip time, if an earlier than the last retransmitted copy triggered the acknowledgment. The SRTT will then converge to wrong values. Consider the following example: if the true RTT is 11 seconds, but the RTO was wrongly set to 10 seconds, the packet is then retransmitted after 10 seconds, and the RTT measurement returns 1 second when the acknowledgment to the first packet is received.



Case 2: SRTT Converges to a wrong value.

3. If the measured RTT is not used to adjust the SRTT when retransmissions occur, the SRTT will not change. If the original RTO is shorter than the real round trip time or the network delay has suddenly increased (e.g. because of route change), the RTO will stick at the small value, resulting in unnecessarily retransmitting every packet.



Case 3: SRTT stays at the wrong value

References

[1] David Clark.
Window and Acknowledgment Strategy in TCP.
ARPA RFC-813.
1982

[2] David Clark, Mark Lambert, & Lixia Zhang.
NETBLT: A Bulk Data Transfer Protocol.
ARPA RFC-969.
December, 1985

[3] Geoffrey Cooper.
A New Timing Algorithm for Transmission and Retransmission in TFTP.
A working paper draft written at Computer System Research group, MIT-LCS.
1983

[4] Stephen W. Edge.
An Adaptive Timeout Algorithm for Retransmission Across a Packet Switching Network.
ACM Computer Communication Review 14(2):248-255, June, 1984.

[5] Raj Jain.
Divergence of Timeout Algorithms for Packet Retransmissions.
Technical Report 329, Digital Equipment Corp., 1985.

[6] David L. Mills.
Internet Delay Experiments.
ARPA RFC-889.
December, 1983

[7] Robert J.T. Morris.
Fixing Timeout Intervals for Lost Packet Detection in Computer Communication Networks.
In *Proc. of National Computer Conference.* AFIPS, 1979.

[8] J. Postel.
DoD Standard Transmission Control Protocol.
ARPA RFC-793.
September, 1981

[9] J. Postel.
DoD Standard Internet Protocol.
ARPA RFC-791.
September, 1981

[10] Lixia Zhang.
Network Simulation Report.
Working paper in progress.
This report summarizes test results on IP source quench handling and TCP timer problems. The simulator was built by the author at MIT-LCS to study network congestion control problems. Its topology imitates the conditions in the current ARPA Internet, i.e. the delay and bandwidth characteristics of communication channels differ by orders of magnitude. The data traffic generator models two types of applications: file transfer and remote login.