Management of Protocol State

Ibrahim Matta

December 2012

1 Introduction

These notes highlight the main issues related to synchronizing the data at both sender and receiver of a protocol. For example, in a P2P file sharing application protocol, one peer (sender) may inform the file directory server (receiver) about the set of files it has and it can serve to other peers. In this case, we say that the state of such protocol is the set of files a peer is willing to share with other peers. Keeping the states at the sender and receiver consistent ensures that a query for a file does not get directed to a peer who no longer is serving the file, or the query does not get falsely rejected as the directory server is unaware of a peer who is currently serving that file.

As another example, a transport protocol that provides reliable service would need to synchronize the sequence numbers of those packets sent and received successfully so that the receiver can distinguish "new" data/ACK packets from "old" ones (i.e. duplicates). In this case, the state of such protocol is the range of valid sequence numbers, in addition to other transfer information such as RTT estimate for the connection, RTO timers, etc. Keeping the states at the sender and receiver consistent ensures that the receiver does not accept duplicate data, or the sender falsely assumes that new data has been delivered because of an old duplicate ACK.

There are two main approaches to synchronizing the state: hard state (HS) and soft state (SS). HS employs reliable explicit signaling for establishing the state when the session starts and for deleting the state when the session ends. On the other hand, SS typically employs best-effort signaling for establishing the state and henceforth the state at the receiver is deleted after a timeout if not refreshed by the sender.

We have also discussed in class that there is a spectrum of schemes that lie between SS and HS, e.g. SS+ER (soft state with explicit reliable state removal).

We first start with reliable transport as a case study, and then we consider general signaling protocol design and performance.

2 Connection Management for Reliable Transport

TCP is an example of HS, though the sender deletes its state after a timeout (i.e. SS timer). So we classify TCP as hybrid HS+SS. Delta-t is an example of pure SS.

To provide transport reliability, the network must provide a Maximum Packet Lifetime (MPL) guarantee. So, in some sense, there is no way around the need for timers. Delta-t relies only on this MPL knowledge to set timers at the sender and receiver to make sure the connection state at both sender and receiver is not deleted until all packets from this connection instance (incarnation) have died out.

On the other hand, TCP relies on explicit handshaking to distinguish a new connection incarnation by way of a "new initial sequence number" – the sender chooses this new initial sequence number to ensure no packets inside the network have that sequence number, which implicitly assumes knowledge of MPL. We have seen in class, considering reliable one-data message delivery, how we can evolve the design of a reliable transport protocol from a minimal two-message exchange (DATA, and ACK) to five-message exchange (SYN, SYN-ACK, DATA, ACK, CLOSE), which degenerates to TCP if we additionally had a state timer at the sender, initialized to 2MPL, to ensure the last CLOSE from the sender has been received successfully at the receiver before the sender deletes the connection state.

We have also seen how Delta-t just augments the two-message exchange with two SS timers: (1) an SS timer at the receiver, initialized to 2MPL whenever a new in-sequence packet is received, to ensure that the receiver does not delete its state as long as the sender is still trying to deliver a message, and (2) an SS timer at the sender, initialized to 3MPL whenever it transmits any packet, to ensure that the sender does not delete its state before the receiver does. Note that the SS timer at the receiver is important to ensure that the receiver does not "forget" this connection unless the sender has given up and all packets from that connection have died out. This way, the receiver is sure that no duplicates will ever be received even if it starts a new connection incarnation with the same sender and the sender uses any initial sequence number for its first packet. The key principle here is that a Delta-t receiver, by keeping (caching) a connection state until all its packets have died out, does not need to verify with the sender whether the first data packet belongs to a new or old connection. If the Delta-t receiver does not have a state for the connection, then it must be a new connection.

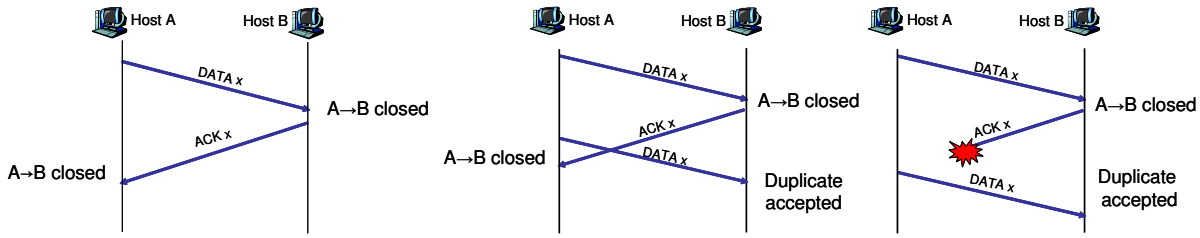
In what follows, we describe the basic operation of the different reliable transport approaches mentioned above, for the worst-case scenario of reliably sending a single message per conversation between a single sender and a single receiver, over a channel that may lose or re-order (delay) messages.[1] We say "worst case" since information from successive packets in a stream can only help the transport protocol, *e.g.*, to identify a missing packet in the stream sequence or to keep the connection state alive (refreshed).

[1] We use the terms "message" and "packet" interchangeably. When we refer to "single-message" or "multi-message" conversation/transfer/communication scenario, then we mean *data* messages.

We review four approaches to reliable transport [Belsnes'76]. They represent a spectrum of solutions where the amount of explicit connection-management messages and the use of connection-state timers vary: (1) the *two-packet* (DATA and its ACK) protocol has no connection-state timers nor explicit connection-management messages, (2) the *three-packet* protocol augments the two-packet protocol with an explicit connection-management CLOSE message, (3) the *five-packet* (TCP) protocol augments the three-packet protocol with explicit connection-management (SYN and SYN+ACK) messages and a connection-state timer at the sender, and (4) the *Delta-t* protocol augments two-packet using only connection-state timers at both the sender and receiver. Delta-t and its predecessor (two-packet) represent soft-state protocols, three-packet represents a hard-state protocol, whereas five-packet represents a hybrid hard-/soft-state protocol[2].

Note that although, from a correctness standpoint, we note below that two-packet and three-packet may result in duplicate connections being accepted, we include them to understand the progression to five-packet and ultimately to TCP.

2.1 Two-Packet Protocol



(a) Normal case (b) Premature retransmission (c) Acknowledgment loss

Figure 1: Two-Packet Protocol

To detect data (packet) loss, this protocol uses positive acknowledgments. When there is data to send, the sender opens a connection to the receiver and transmits the data message. Opening a connection means that control information is kept about the connection, which we refer to as *state information*. When the receiver receives the data message, it opens a connection, delivers the data message to the application, sends an acknowledgment message back to the sender, and immediately closes the connection. Closing the connection means removing the state information of the connection. A normal conversation is illustrated in Figure 1(a).

If the sender does not receive the acknowledgment within an estimated retransmission timeout (RTO) duration, then it retransmits the data message. Figure 1(b) illustrates the

[2]We use "five-packet" and "TCP" interchangeably as we augment the basic five-packet with TCP's connection-state timer at the sender.

case where the retransmission timeout value is underestimated, thus the sender prematurely retransmits the data message. Since the receiver closes the connection right after it sends the acknowledgment, it can not distinguish a premature retransmission (duplicate) from new data (new connection). Thus, the receiver accepts and delivers a duplicate to the application.

Another scenario that causes data duplication is when the network (channel) loses the acknowledgment. Figure 1(c) illustrates this case. If the acknowledgment is lost, the sender retransmits the data message after RTO.

In [Belsnes'76], the correctness of the two-packet protocol is studied in detail, including the case of data messages falsely acknowledged (*i.e.*, without being actually delivered) and hence lost. This latter problem is solved by introducing sequence numbers [Tomlinson'75]. The sender appends to each new data message a new sequence number that has not been recently used in its communication with the receiver.[3] A sequence number should not re-used until all messages with that sequence number (including duplicates) have left the network. Thus, the two-packet protocol (augmented with such initial sequence numbers) does not lose data but may accept duplicates.

2.2 Three-Packet Protocol

To solve the duplication problem due to acknowledgment loss, this protocol augments the two-packet protocol with an acknowledgment for the ACK, which can be thought of as an explicit CLOSE connection-management message sent by the sender. When there is data to send, the sender opens a connection to the receiver and transmits the data message. When the receiver receives the data message, it opens a connection, delivers the data message to the application, sends an acknowledgment message back to the sender, and waits for the CLOSE message from the sender before clearing the connection-state. When the sender gets the acknowledgment, it transmits the CLOSE message to the receiver and closes the connection. The receiver in turn closes the connection once it gets the CLOSE message.

Despite the extra CLOSE message, this protocol does not solve the duplication problem. If a delayed retransmission of a data message arrives at the receiver right after the receiver closes the connection, the receiver wrongly opens a new connection and accepts a duplicate.

2.3 Five-Packet Protocol

To avoid data duplication, two additional explicit connection-management messages are introduced to open a connection. Figure 2 illustrates a normal conversation of the protocol (ala TCP). The sender transmits a synchronization SYN message to initiate the connection.

[3] In practice, since the sender does not keep state once the connection ends, a sender would probably pick a random initial sequence number in the hope that it indeed does not exist in the network! Otherwise, the sender would use for all its connections a large sequence number space that does not wrap around for at least 2MPL.

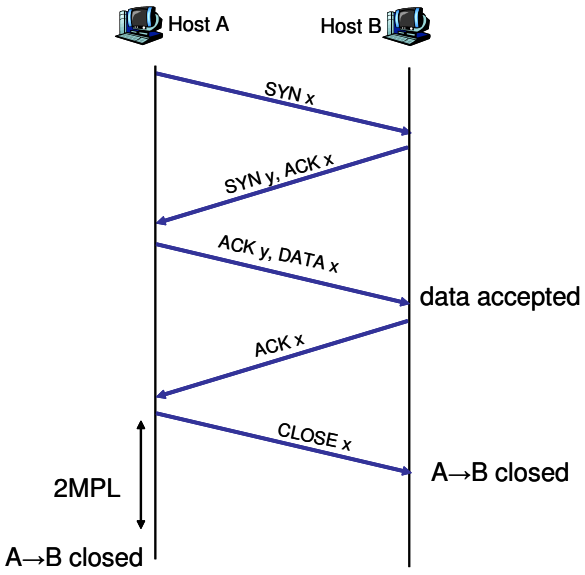


Figure 2: Five-Packet Protocol

The receiver responds to the SYN message with a SYN+ACK message. The sender then transmits the data message, which also acknowledges the receiver's SYN, thus synchronizing the sender and receiver, ensuring that the initial SYN message is not a duplicate (from an old connection). Upon receiving the acknowledgment for its data, the sender transmits an explicit CLOSE message and closes the connection. Upon receiving the CLOSE message, the receiver closes its end of the connection.

TCP follows this five-packet protocol. However, in TCP, after the sender sends the CLOSE message, it does not immediately close the connection, rather it waits for at least 2×MPL. This is the case because the last CLOSE message can get lost, which would lead to inconsistent states when the state is closed (removed) at the sender and still open at the receiver. This timed wait of 2×MPL at the sender ensures that the last ACK-CLOSE exchange is completed, otherwise both ends will terminate the connection and remove its state. Note that at this point, no packets from this connection exits in the network.

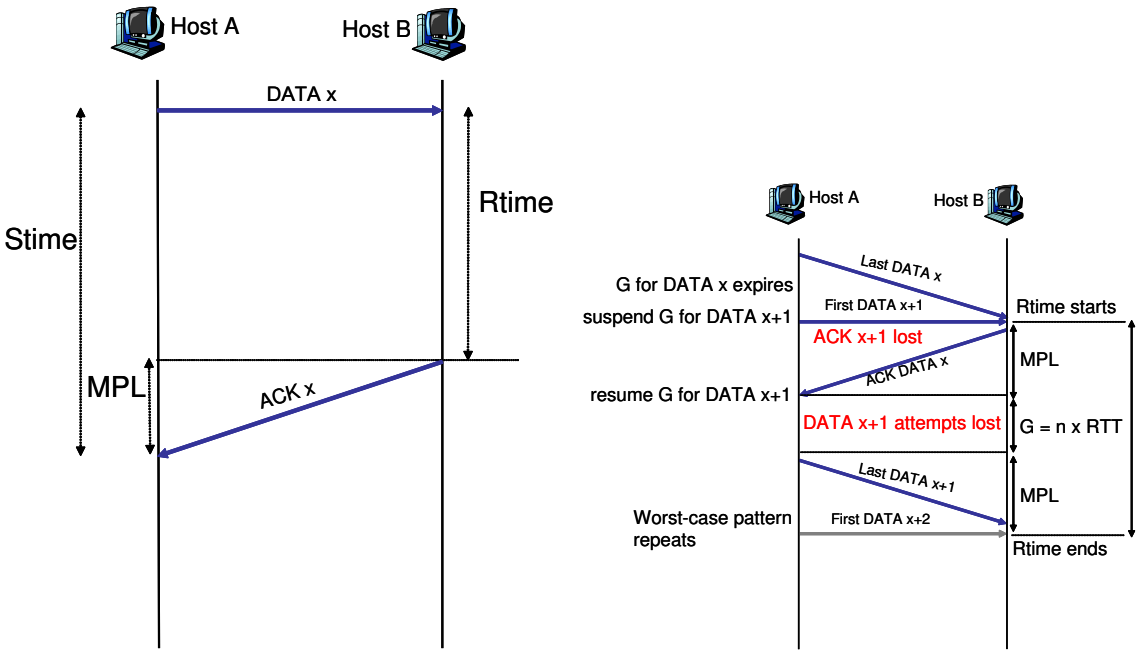
2.4 Delta-t Protocol

As noted above, the transport protocol inevitably assumes, either implicitly or explicitly, that the underlying network (channel) provides a guarantee on the Maximum Packet Lifetime (MPL). The Delta-t protocol [Watson'81] thus exclusively relies on connection-management (state) timers that are bounded by MPL. Delta-t is basically a two-packet protocol, augmented by state timers at both the sender and receiver to solve the problem of data duplication. Unlike the five-packet protocol, there are no explicit (separate) messages

to open and close the connection.

The sender and the receiver state timers are set to guarantee that none of the messages (including duplicates) of the active connection will arrive to the ends after they close the connection. Figure 3(a) illustrates the connection state lifetime at the sender and the receiver. The sender starts its state timer whenever it sends a data message (new or retransmission). The connection at the sender should be open long enough—denoted by *Stime*—to receive the acknowledgment, which could be transmitted in the worst-case right before the receiver state lifetime—denoted by *Rtime*—expires. Since the lifetime of a packet is bounded by MPL, we have the following relationship:

$$Stime = Rtime + MPL \tag{1}$$



(a) State Timers

(b) Setting Timers

Figure 3: Delta-t Protocol

The receiver starts its connection-state timer whenever it receives (and accepts) a new data message. The receiver state timer should be running long enough to receive all possible retransmissions of the data message in the presence of an unreliable (lossy) channel. This allows the receiver to catch (recognize) duplicates of the data message. The connection is closed at the receiver after the last possible acknowledgment for the connection is sent. Figure 3(b), reproduced from [Fletcher'78], illustrates the worst-case multi-message

conversation between the sender and receiver[4]. Denote by G, the maximum time a sender keeps retransmitting a data message before it gives up and aborts the connection. If n is the maximum number of retransmissions for each data message, then $G = n \times RTO \approx n \times RTT$. Associate with each data packet a timer initialized to G when it is first transmitted. Whenever a data packet's G-timer expires, the G-timers of all other data packets are frozen hoping to successfully get the acknowledgment, otherwise the connection is aborted and the application is informed.

Figure 3(b) shows the multi-message scenario when a new data packet (whose sequence number is $x + 1$) is received instantly, so in the worst case, *Rtime* is started as early as possible. Due to consecutive losses, the G-timer of the previous data packet (whose sequence number is x) *expires* while waiting for the acknowledgment ACK x for its last retransmission attempt, which in the worst case, will take MPL to arrive. At this time instant, the G-timers of all oustanding packets are frozen, thus data packet $x+1$ has not yet used up its maximum delivery time G. Now when ACK x arrives, in the worst case, due to ACK losses, data packet $x+1$ keeps getting retransmitted until all its G is consumed by the time its last retransmission is sent, which in the worst case, takes another MPL to arrive at the receiver. This worst-case pattern repeats with data packet $x + 2$, which causes the receiver's state timer to be re-started (refreshed). Given this worst-case scenario, a Delta-t receiver sets its *Rtime* as follows:

$$Rtime = 2 \times MPL + G \approx 2MPL \tag{2}$$

Thus, substituting *Rtime* in Equation (1), we have:

$$Stime = 3 \times MPL + G \approx 3MPL \tag{3}$$

3 General Signaling

Inconsistent states at the sender and receiver may cause incorrect behavior or bad performance. In our reliable transport case study above, we have ensured that the design of either TCP (exemplifying mostly HS design) or Delta-t (exemplifying SS) is correct, i.e. no data loss or duplication.

In general, in terms of performance, the design of SS wins in terms of robustness. By *robustness* we mean that performance does not degrade precipitously as network conditions get worse than normal (e.g. as message losses and delays increase). Moreover, the sender and receiver are loosely coupled which makes the design simpler. We discussed an analytical cost model [Lui'04] that captures a spectrum of SS and HS behavior, and used it to show that SS is more robust than HS, i.e., SS has slightly higher cost under good/normal network conditions, but much lower cost than HS under bad network conditions. The derived cost expression is a function of the refresh period R; it has three terms:

[4]For simplicity, we assume that the receiver does not delay sending its acknowledgment.

1. a term that represents the overhead and cost of losing refresh messages present in SS protocols,

2. a term representing the inconsistency cost due to lost signaling messages, and

3. a term that represents the cost of "orphaned" state at the receiver, i.e. state not yet removed waiting for explicit removal or to be timed out.

The model attempts to calculate the optimal R that minimizes the total cost. It captures HS by giving higher relative weight to the first term since protocols that are more HS have minimal or no refresh cost. SS is captured by giving higher relative weight to the last term since protocols that are more SS should attempt to minimize the cost of orphaned state present until the SS state timer expires. SS is shown to have much lower cost under high message loss probability as more frequent refreshes, due to the small optimal R, are able to overcome the high loss conditions and thus reduce the cost of orphaned state at the receiver. An HS protocol, on the other hand, suffers from losing its signaling (state trigger or removal) messages under high losses, where each signaling message requires at least RTT to detect its loss and be retransmitted.[5]

On the other hand, in general, HS wins in terms of inconsistency ratio and signaling overhead. This is because state triggers (installs) and removals are sent reliably, and there is no need for periodic refresh messages. We discussed a continuous-time Markov model [Ji'03] developed to model SS and other signaling approaches, and used it to compute the inconsistency and signaling rate metrics. By *inconsistency ratio* we mean the fraction of time when the sender and receiver are not synchronized, i.e. they do not have the same exact views or values of the state variables. Under normal low loss conditions, reliable signaling for state triggers and removals in HS costs much less than periodic refreshes in SS. Note however, that this last statement is misleading: to be robust to extreme network conditions, e.g. sender crashes, in HS, an underlying keep-alive mechanism is usually employed to signal such error to the "orphaned" receiver so state can eventually be removed.

Then the main drawback of SS is typically higher inconsistency because of the delay in removing state at the receiver, waiting for the state to time out, called "orphaned" state. The price of this is higher memory usage, in addition to inconsistency. With an explicit state removal and/or setting the SS timer at the receiver to be the minimal possible (as allowed by correctness and performance goals), an SS design should be the right choice given its robustness and simplicity!

4 Exercises

1. For the following statements, circle your choice:

[5] In practice, there is a limit on the number of retransmission attempts, after which the session is aborted.

(a) In Delta-t, the sender's connection state timer is typically set to (2×MPL OR 3×MPL), where MPL is the Maximum Packet Lifetime. And the receiver's state timer is set to (2×MPL OR 3×MPL).

(b) Soft-state signaling protocols are generally found to be (more OR less) consistent, and (more OR less) robust, compared to hard-state protocols.

2. Argue for or against Delta-t compared to TCP. Consider signaling overhead, inconsistency ratio, and robustness metrics and how they manifest in the two protocols in practice. For example, consider that signaling information can be piggybacked onto data packets, connections could be for short or long data transfers, state inconsistencies may affect different aspects of correctness, performance, resource consumption, etc.

   Also, comment on the "complexity" of a Delta-t vs. TCP implementation, i.e. which one do you think is easier to code and maintain?