# Transport Layer: Correctness Principles[*]

A. Udaya Shankar
Computer Science Department
University of Maryland, College Park

October 25, 2002

## 1 Introduction

The **transport layer** of a TCP/IP computer network is situated above the **network layer** and below the **applications layer**, as shown in Figure 1. The network layer provides unreliable packet transfer service between any two hosts. The transport layer uses this network service and provides transport services between any two applications in the network. Applications include email (SMTP), remote login (TELNET, SSH), file transfer (FTP), web browsers (HTTP), remote file systems (NFS), name-to-address translation (DNS), voice and video streaming (e.g. Real Networks), internet telephony (e.g. Vocaltec), etc. We refer to applications using the transport service as **transport users**, or **users** for short. The transport services are provided by transport protocols, which are distributed algorithms running on the hosts. There are different transport services, and hence different transport protocols. We refer to the components of the protocols running at the hosts as **transport entities**, or **entities** for short.

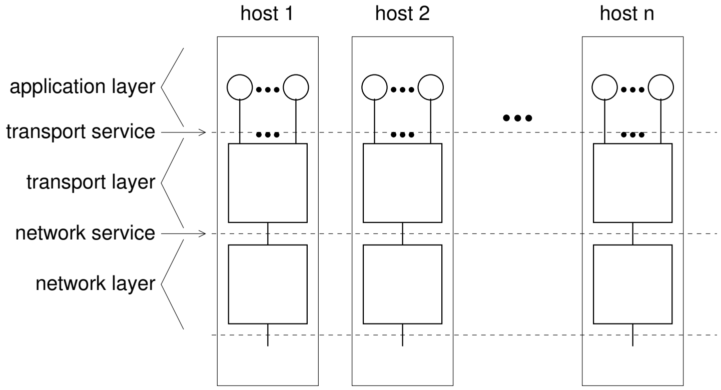

Figure 1: The application, transport and network layers.

Historically, transport protocol design has been driven by the need to operate correctly inspite of unreliable network service and failure-prone networks and hosts. In particular, the following **failure assumptions**, which are still valid in today's TCP/IP networks, were made:

- Hosts can fail and recover. A host failure is "fail-stop", that is, a failed entity performs no actions and retains no state information except for stable storage.

[*] An earlier version of this, titled "Transport Layer Principles", appears in The Communication Handbook, CRC Press, 1996

- The channels, i.e., packet transfer service, provided by the network layer between any two hosts can lose, duplicate, and reorder messages in transit. The channels impose (usually implicitly) a "maximum lifetime" on all messages, which is typically much larger than the expected end-to-end delay (e.g., an hour versus seconds).
- Channels can fail and recover. A channel failure means that the probability of message delivery becomes negligible, that is, even with retransmissions a message is not delivered within a specified time.

**User multiplexing and demulitplexing:** Providing user-to-user service implies that the transport layer has to do user multiplexing and demulitplexing at each host. The TCP/IP architecture uses **IP addresses** (32 bits in IPv4) to identify hosts, **port numbers** (16 bits in IPv4) to identify users, and **transport protocol numbers** (8 bits in IPv4) to identify transport protocols. Every transport user has a **local port number** (its port number) and a **local IP address** (the IP address of its host). Every IP packet has (in its header) the following attributes: **sender port number**, **sender host IP address**, **destination port number**, **destination host IP address**, and **transport protocol number**. A packet's sender attributes are the local attributes of the user that sent the packet. A packet's destination attributes are the local attributes of the user that is supposed to get the packet.

**User-user transport service:** The "ideal" transport service between two users is one that transfers data packets reliably and with low-delay and low-jitter. **Reliable data transfer** means that data is delivered in the same sequence it was sent and without loss. **Low-delay** means that data sent is delivered within a specified (usually small) time bound of it being sent. **Low-jitter** means that the time intervals between data sends is preserved at delivery within specified (usually small) time bounds. Achieving such ideal service requires the network to be capable of handling the worst-case load at any time, which, if the network is not to be incredibly expensive, means imposing severe restrictions on the network access and the data rates available to users (as in telephony networks).

Fortunately, this ideal service is hardly ever required by applications. The transport layer in TCP/IP networks does not strive for it. Instead it provides two separate services: a **reliable service** which can suffer high delays and jitter, and an **unreliable service** which does no better than the network-layer service. The reliable service, implemented by a transport protocol known as TCP, is used by applications where data integrity is essential, such as file transfer, email, remote login, etc. The unreliable service, implemented by a transport protocol known as UDP, is used by applications where data loss can be tolerated but low-delay or low-jitter is desired, such as internet telephony, voice/video streaming, and periodic control updates (e.g., DNS, routing). A TCP/IP network can also have other transport protocols (e.g., IPX).

To a first approximation, reliable transport service is nothing but reliable data transfer between any two users. But reliable data transfer requires resources at the entities, such as buffers and processes for retransmitting data, reassembling data, etc. These resources typically cannot be maintained across failures. Furthermore, maintaining the resources continuously for every pair of users would be prohibitively inefficient, because only a very small fraction of user pairs in a network exchange data with any regularity, especially in a large network such as the Internet. Therefore, a reliable transport service involves **connection management** and **data transfer**. Data transfer provides for the reliable exchange of data between connected users. Connection management provides for the establishment and termination of connections between users.

In summary, reliable transport service (e.g., TCP service) involves three aspects: user multiplexing, reliable connection management between users, and reliable data transfer between connected users. Unreliable transport service (e.g., UDP service), on the other hand, involves two aspects: user multiplexing and unreliable data transfer between users.

**Application-level architecture:** Currently, applications follow the so-called **client-server** architecture. Here, the users of the transport layer are partitioned into clients and servers and connections are initiated only by clients and only to servers. Traditionally, servers are on prespecified machines and use prespecified ports. A recent trend is **peer-to-peer** architectures, where servers are use arbitrary machines and ports and are typically on the same machines as clients.

However, the transport and lower layers are not concerned about whether its users follow the traditional client-server model or a peer-to-peer model. There is nothing in the TCP/IP protocols and message headers that identifies these aspects of the sender or destination.

**Organization of this note:** Section 2 describes UDP, in particular, user multiplexing in UDP. Section 3 describes the basic features of reliable transport services. Section 4 describes user multiplexing in TCP. Section 5 describes a protocol that provides reliable data transfer. Section 6 describes a protocol that provides connection management. Section 7 combines the previous protocols to obtain a transport protocol that provides the same service as TCP and is simpler. Section 8 describes so-called minimum latency transport protocols. Section 9 concludes.

**Note:** This report deals with correctness issues in the transport layer. It does not address the equally important performance issues, which deal with how to specialize, or "tune", a correct system to achieve good performance for the situations usually encountered.

## 2 Unreliable Transport Service

UDP, the TCP/IP transport protocol that provides unreliable service, is essentially user multiplexing added on top of the unreliable host-to-host message transfer service provided by the network layer. Hence the rest of this section describes user multiplexing in UDP.

As mentioned earlier, TCP/IP associates a local port number to each user, and TCP/IP packets identify the end-users by sender port number, sender IP address, destination port number, destination IP address, and transport protocol number.

The application layer invariably has other forms of identifiers for users, depending on the host operating system. Thus users need to be mapped to port and protocol numbers. Furthermore, the mapping to port numbers should be dynamic because the set of users of a transport service at a host is not static.

UDP uses the obvious approach to mapping. Each UDP user is identified network-wide by its local port number, its IP address, and its protocol number (17 for UDP). The local port number is assigned when it starts UDP service.

- When a user requests access to the UDP service, it gets a local port number, which is either one that is specified by the user (e.g., if it is a server) or any free port number. If no appropriate port number is free, the user is denied access to the UDP service.

- Once a user has access to the UDP service, it can send UDP packets with packet sender attributes set to the user's local attributes, transport protocol number set to UDP, and any destination attributes. [More precisely, the UDP entity adds the sender port number and destination port number and IP address. The IP entity attaches the sender IP address and protocol number.]

- Once a user has access to the UDP service, it can receive any UDP packet with the packet destination attributes equal to the user's local attributes. [More precisely, the local IP entity at a host screens a received packet based on the packet's destination IP address and protocol number, either discarding it or passing it up to the appropriate transport entity. The UDP entity at the host screens a received packet based on the packet's destination port, either discarding it or passing it up to the appropriate user.]

- When a user stops using the UDP service, its local port number becomes free.

## 3 Reliable Transport Service

Recall that reliable transport service has three aspects: user multiplexing, connection management, and data transfer. TCP, the TCP/IP transport protocol that provides reliable service, has sophisticated mechanisms for each of these aspects. User multiplexing in TCP uses a different mechanism from that in UDP. Data transfer provides for the reliable exchange of data between connected users. Connection management provides for the establishment and termination of connections between users. Users can open and close connections to other users, and can accept or reject incoming connection requests. Resources are acquired when a user enters a connection, and released when the user leaves the connection. An incoming connection request is rejected if the user has failed or its transport entity does not have adequate resources for new connections.

A key concern of transport protocols is to ensure that a connection is not infiltrated by old messages that may remain in the network from previous terminated connections. The standard techniques are to use the **3-way handshake** mechanism for connection management and the **sliding window** mechanism for data transfer within a connection. These mechanisms use **cyclic sequence numbers** to identify the connection attempts of a user and the data blocks within a connection. The protocol must ensure that received cyclic sequence numbers are correctly interpreted, and this invariably requires the network to enforce a **maximum message lifetime**.

To define the correctness properties of reliable transport service, we first define the notion of **incarnations**. Each user goes through a succession of incarnations. An incarnation of a client is started whenever the client requests a connection to any server. An incarnation of a server is started whenever the server accepts a (potentially new) connection request from any client. Every incarnation is assigned an **incarnation number** when it starts; the incarnation is uniquely distinguished by its incarnation number and user id. (In TCP, a user is identified by its local/remote port numbers and IP addresses and the appropriate protocol id.)

Once an incarnation x of a user c is started in an attempt to connect to a user s, it has one of two possible futures. The first possibility is that at some point x becomes open and acquires an incarnation number y of some incarnation of s — we refer to this as "x becomes open to incarnation y of s"; at some later point x becomes closed. The second possibility is that x becomes closed without ever becoming open. This can happen to a client incarnation either because its connection request was rejected by the server or because of failure (in the server, the client, the relevant transport entities, or the channels). It can happen to a server incarnation either because of failure or because it was started in response to a connection request that later turns out to be a duplicate request from some old, now closed, incarnation.

Because of failures, it is also possible that an incarnation x of c becomes open to incarnation y of s but y becomes closed without becoming open. This is referred to as a "half-open" connection.

A **connection** is an association between two open incarnations. Formally, a connection exists between incarnation x of user c and incarnation y of user s if y has become open to x and x has become open to y. The following properties are desired of connection management:

- Consistent connections: If an incarnation x of user c becomes open to an incarnation y of user s, then incarnation y is either open to x or will become open to x unless there are failures.
- Consistent data-transfer: If an incarnation x of user c becomes open to an incarnation y of user s, then x accepts received data only if sent by y.
- Progress: If an incarnation x of a client requests a connection to a server, then a connection is established between x and an incarnation of the server within some specified time, provided the server does not reject x's request and neither client, server nor channels fail within that time.
- Terminating handshakes: The transport entity (of either user) cannot stay indefinitely in a state (or set of states) where it is repeatedly sending messages expecting a response that never arrives. (Such "infinite chatter" is worse than deadlock because in addition to not making progress, the transport protocol is consuming precious network resources.)

Given a connection between incarnations x and y, the following properties are desired of the data transfer between x and y:

- In-sequence delivery: Data blocks are received at y (x) in the same order as they were sent by x (y).
- Progress: A data block sent by x (y) is received at y (x) within some specified time, provided the connection is not terminated (either intentionally or due to failures) within that time.

As you can see, reliable transport service is not easily defined. As you can guess, TCP uses sophisticated mechanisms to achieve it. To ease our understanding of these mechanisms, we shall first examine not TCP itself but some conceptually cleaner protocols that achieve the same services.

In the following sections, we develop in stages a transport protocol that achieves this reliable service. We first describe the user multiplexing mechanism of TCP. We next develop a protocol simpler than TCP that provides reliable one-way data transfer between permanently connected users. We next develop a protocol simpler than TCP that that provides reliable connection management between a client user and a server user. We then combine the components to form a protocol that provides reliable transport service (minus user multiplexing).

## 4 Multiplexing in TCP

TCP uses a more sophisticated multiplexing scheme than UDP. A TCP user is identified network-wide by five attributes: local port number, local host IP address, transport protocol number (6 for TCP), **remote port number**, and **remote host IP address**. The first three attributes are as in UDP. The remote port number and remote IP address are the local port number and local host IP address of the remote peer user. The remote attributes are "nil" if the remote user has not yet been identified. Once the remote user is identified, the remote attributes become non-nil and do not change.

A user's interaction with the TCP service is as follows:

- When the user requests access to the TCP transport service, the request can specify (1) a particular local port number, (2) a remote port number and IP address, or (3) both. If 1 is not specified, any local port number will do. If no appropriate port number is free, the user is denied access to the TCP service. If 2 is not specified, the remote attributes are set to nil. [Typically, a server specifies 1 but not 2 and a client specifies 2 but not 1.]

- A TCP user whose remote attributes are not nil can (1) send TCP packets whose sender attributes match the user's local attributes and destination attributes match the user's remote attributes; and (2) receive TCP packets whose destination attributes match the user's local attributes and sender attributes match the user's remote attributes.
- A TCP user whose remote attributes are nil (presumably a server) can receive TCP packets whose destination atributes match the user's local attributes *provided* there is no other local TCP user whose local and remote attributes match, respectively, the destination and sender attributes of the packet (such a user, if present, would get the packet).
- When a user stops using the TCP transport service, its local port number becomes free.

This approach of using both local and remote attributes to identify a user supports the client-server paradigm very nicely, in particular, the handling of many clients simultaneously using the same service. Consider a host H providing a service over TCP. H dedicates a prespecified local port number, say p1, to the service (standard services have prespecified, or "well-known" port numbers, e.g., FTP has port number 21). H creates a server user, say S, with local port number set to p1, transport protocol number set appropriately, and remote port number and IP address set to nil. When a client user, say C, on another host G wants to avail of this service, C would get local port number set to an arbitrary value, say p2, remote port number set to p1, remote IP address set to H's IP address, and transport protocol number set appropriately. When the request packet arrives at the transport layer in H, it gives the packet to S (assuming that there is no user at H with local port number p1, remote port number p2, remote IP address equal to G's IP address). The server S then can create another server specifically for servicing client C; this new server would have remote port number set to p2 and remote IP address set to G's IP address, and hence it can use local port number p1, same as S.

## 5 A Data-Transfer Protocol

This section describes the sliding-window method for achieving reliable flow-controlled data transfer, *assuming that users are always connected and correctly initialized.* Later we incorporate connection management. Throughout these sections, we consider the configuration shown in Figure 2, with users c and s, and associated transport entities c and s connected by unreliable channels.

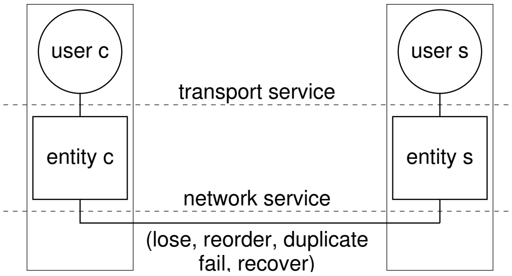

Figure 2: Network with two users.

Consider the two entities c and s connected by unreliable network channels. User c produces data blocks to be delivered to user s. Because the channels can lose messages, every data block has to be retransmitted until it is acknowledged. For throughput reasons, entity c should be able to have several data blocks *outstanding*, i.e., sent but not acknowledged. Similarly, entity s should be able to buffer data blocks received out-of-sequence (due to retransmissions or network-layer reordering).

Let us number the data blocks produced by user c with successively increasing sequence numbers, starting from 0. The sliding-window mechanism maintains two windows of sequence numbers, one at each entity, as shown in Figure 3,

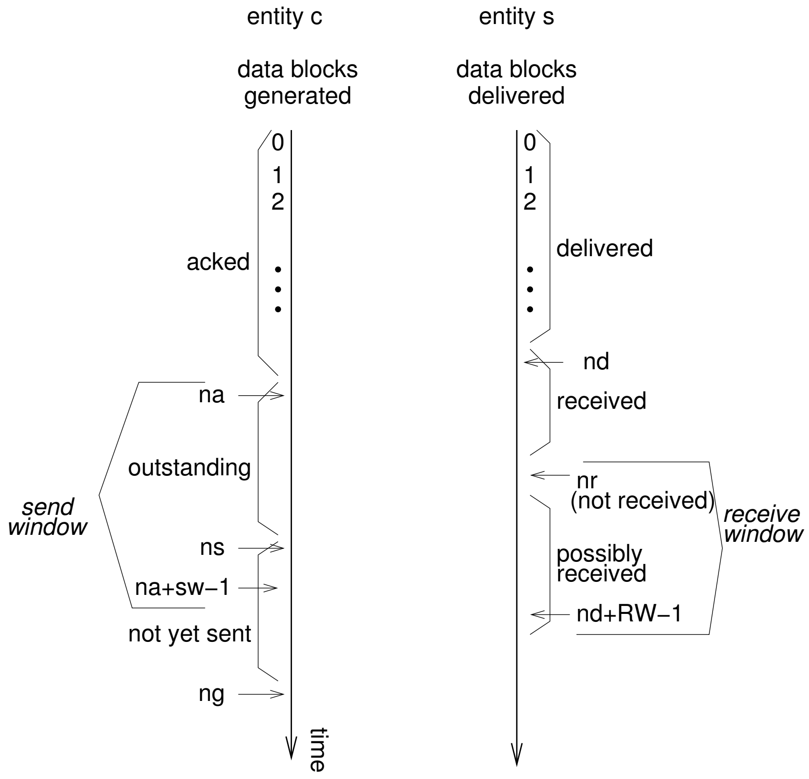

Figure 3: Sliding window mechanism.

Entity c maintains the following variables:

- sbuff: buffer of size SW data blocks.
- ng: { 0, 1, ... }; initially 0. Number of data blocks generated by the local user.
- ns: { 0, 1, ... }; initially 0. Number of data blocks sent at least once.
- na: { 0, 1, ... }; initially 0. Number of data blocks acknowledged.
- sw: { 0, 1, ..., SW }; initially SW (any value is acceptable). Maximum number of data blocks that can be outstanding. Used for flow control, i.e., controlling the rate at which packets are sent.

Data blocks na to ng−1 have been accepted from the local user and are not yet acked, so they must be buffered. Data blocks na to ns−1 are outstanding. The sequence numbers na to na+sw−1 constitute the **send window**; only data blocks in this window can be sent. Note that sw is the size of the send window; flow control is achieved by controlling sw. The following hold at all times: na≤ns≤ng, ns−na≤sw, and ng−na≤SW. Note that na + sw can be higher or lower than ng; that is, entity c may be allowed to send more data blocks than it has.

Entity s maintains the following variables:

- rbuff: buffer of size RW data blocks.
- nd: { 0, 1, ... }; initially 0. Number of data blocks delivered to the local user.
- nr: { 0, 1, ... }; initially 0. Number of data blocks received in sequence from the remote user.
- rw: { 0, 1, ..., RW }; initially RW. Number of out-of-sequence data blocks that the entity can buffer. Always equals RW−nr+nd.

Data blocks 0 to nd−1 have been delivered in sequence to the local user. Data blocks nd to nr−1 have been received and ready for delivery to the local user, but until then they must be bufered. At all times, nd≤nr holds. Entity s can buffer up to RW data blocks, that is, the ones in the range nd to nd+RW−1. Data block nr has not been received. Any data block in the range nr+1 to nd+RW−1 that has been received (out of sequence) is buffered. The sequence numbers nr to nd+RW−1 constitute the **receive window**, and rw indicates its size.

We now consider how an entity can identify a received data block or acknowledgement. The easiest way is for the message to include the sequence number of the concerned data block. But such a sequence number field would grow without bound, which is unsuitable for packet formats. The typical solution is to use cyclic sequence numbers in packets; i.e., mod(N,j) for some N, instead of the "unbounded" sequence number j. (Note: mod(N,j) is defined as satisfying $0 \leq \mathrm{mod}(\mathrm{N},\mathrm{j}) < \mathrm{N}$ and $\mathrm{j} = \mathrm{mod}(\mathrm{N},\mathrm{j}) + \mathrm{k}$ for some integer k.)

When entity s receives a cyclic sequence number cj, it looks for a matching unbounded sequence number j in the receive window, i.e., j such that mod(N,j) equals cj; if such a j exists it treats cj as corresponding to that. Note that $\mathrm{nr} + \mathrm{mod}(\mathrm{N}, \mathrm{cj} - \mathrm{nr})$ is the first unbounded number on or after nr that matches cj; it lies in the receive window if $\mathrm{mod}(\mathrm{N}, \mathrm{cj} - \mathrm{nr}) \leq \mathrm{rw}$.

Similarly, when entity c receives a cyclic sequence number cj, it looks for a matching unbounded sequence number j in the send window i.e., j such that mod(N,j) equals cj; if such a j exists it treats cj as corresponding to that. Again, $\mathrm{na} + \mathrm{mod}(\mathrm{N}, \mathrm{cj} - \mathrm{na})$ is the first unbounded number on or after na whose modulo N value equals cj; it lies in the send window if $\mathrm{mod}(\mathrm{N}, \mathrm{cj} - \mathrm{nr}) \leq \mathrm{rw}$.

To ensure that a received cyclic sequence number is correctly interpreted, it is necessary for the network to enforce a maximum message lifetime, i.e., no message older than the lifetime remains in the network. It then suffices if N satisfies

$$\mathrm{N} \;\geq\; \mathrm{SW} + \mathrm{RW} + \frac{\mathrm{L}}{\delta}$$

where SW and RW are the maximum sizes of the send and receive windows, L is the maximum message lifetime, and $\delta$ is the minimum time between successive data block sends. This bound is derived below.

**Flow-control** is another issue in data transfer, i.e., entity c should not send data faster than the network or entity s can handle. By dynamically varying the send window size, the sliding window mechanism can also achieve flow control. In particular, consumer-directed flow control works as follows: entity s regularly informs entity c of its current receive-window size and entity c sets its send-window size accordingly. Note that in this case, the above condition on N reduces to $\mathrm{N} \;\geq\; 2\mathrm{RW} + \mathrm{L}/\delta$.

We finish this section with a specification of the sliding-window protocol, under the following conventions:

- The data messages of the protocol have the form (D, sid, rid, data, cj), where sid is the sender's id, rid is the intended receiver's id, data is a data block, and cj is its cyclic sequence number.

- The ack messages of the protocol have the form (ACK, sid, rid, cj, w), where sid and rid are as above, cj is a cyclic sequence number and w is a window size. When the message is sent, cj and w are set to the values of mod(N,nr) and rw, respectively. Thus the message indicates the data block next expected in sequence. Because it acknowledges all earlier data blocks, it is referred to as a "cumulative ack".

- Treat sbuff as a sequence of SW entries indexed from 0 to $\mathrm{SW}-1$, such that entry sbuff[i] either holds data block na+i or is empty. In particular, sbuff[i] is empty iff (if and only if) $\mathrm{ng}-\mathrm{na} \le \mathrm{i} \le \mathrm{SW}-1$.

  We say "slide sbuff by k" to mean that the entries 0 to $\mathrm{k}-1$ are dropped from sbuff and k empty blocks are appended to sbuff at the other end (i.e., sbuff becomes the sequence sbuff[k..$\mathrm{SW}-1$] followed by k empty entries).

- Treat rbuff as a sequence of RW entries indexed from 0 to $\mathrm{RW}-1$, such that entry rbuff[i] either holds data block nd+i or is empty. So rbuff[i] is always empty for $\mathrm{i} = \mathrm{nr}-\mathrm{nd}$ and may be empty for $\mathrm{nr}-\mathrm{nd}+1 \le \mathrm{i} \le \mathrm{RW}-1$. "Slide rbuff by k" is defined in the same way as "slide sbuff by k".

- The activity of the producer and consumer entities are shown in Figure 4, using an event-based notation. There are two types of events. A "nonreceive" event has an enabling condition, denoted ec, and an action, denoted ac; the action can be executed whenever the event is enabled. A receive event for a message has only an action; it is executed whenever the message is received.

[Note: For the case tmp=0 in the receive ACK event, it may seem that sw := max(sw, w) should be sw := w, in order to keep up with the most recent value of rw. In fact, sw := max(sw, w) is the correct thing to do. The reason is that the top of the receive window, nr+rw, never decreases. Consequently, if there are two ack messages with the same nr value and different rw values, the message with the higher rw value is more recently sent. Note that doing sw := w, can degrade performance (by causing the source to mistakenly withold sending data for a while).]

[Note: This event-based notation can be implemented in a standard programming language such as Java as follows. Each entity is implemented as an object (i.e., class instance) with zero or more threads. Events are of two types: events initiated by the entity itself (e.g., send datablock) and events initiated by the entity's environment (e.g., accept datablock, receive message). Locally-initiated events would be executed by the threads of the object; typically the threads would introduce time delays between event executions (i.e., schedule events). Externally-initiated events would be represented by public methods of the object. Each event execution should be atomic, and this can be ensured by using appropriate locking mechanisms (e.g., semaphores, locks, the synchronized construct in Java).]

There are various ways to extend the above protocol.

- The above protocol uses cumulative acknowledgments. We can also use "negative" acknowledgements (nacks) to indicate gaps in the received data. Nacks allow the data source to retransmit missing data sooner than cumulative acks. The protocol can also use "selective" acknowledgements (sacks) to indicate out-of-sequence data received. This allows the data source to retransmit only what is needed, rather than everything outstanding. Selective acks and nacks are not usually used in TCP, although they are available as options and there are studies indicating that they can improve performance significantly.

- The above protocol uses fixed-size data blocks. An alternative is to send variable-sized data blocks. This can be done by augmenting the data messages with a field indicating the size of

**Entity c**

```
Accept datablock from user
 ec:  ng−na < SW
 ac:  sbuff[ng−na] := datablock;
      ng := ng+1

Send new datablock
 ec:  ns < ng and ns < na + sw
 ac:  Send (D, c, s, sbuff[ns-na], mod(N,ns));
      ns := ns+1

Resend datablock (na+j)
 ec:  0 ≤ j ≤ ns−na−1
 ac:  Send (D, c, s, sbuff[j], mod(N,na+j))

Reception of (ACK, s, c, cj, w)
 ac:  tmp := mod(N, cj−na);
      if 1 ≤ tmp ≤ ns−na then
         // cj matches na+tmp
         slide sbuff by tmp;
         na := na+tmp;
         sw := w
      else if tmp = 0 then sw := max(sw, w)
      // else tmp > ns−na; do nothing
```

**Entity s**

```
Deliver datablock to user
 ec:  nd < nr
 ac:  deliver data block in rbuff[0];
      slide rbuff by 1;
      nd := nd+1;

Send acknowledgement
 ec:  true       // also does resends
 ac:  Send (ACK, s, c, mod(N,nr), rw)

Reception of (D, c, s, data, cj)
 ac:  tmp := mod(N, cj−nr);
      if 0 ≤ tmp < rw then
         // cj matches nr+tmp;
         rbuff[nr − nd + tmp] := data;
      // else tmp≥rw; do nothing
      if tmp=0 then      // maximize nr
         while rbuff[nr-nd] not empty
            do nr := nr + 1
```

Figure 4: Events of sliding-window protocol

the data block. Another alternative is to send a variable number of data blocks in a message; if the data blocks are consecutive, the message needs only identify the sequence number of the first data block and the number of data blocks. TCP does the latter with an octet, or byte, as the data block size. A similar modification would be needed for selective and negative acks.

## Deriving the bound on N

We now derive the bound on N shown above. It's convenient to assume that each message contains the unbounded sequence number corresponding to the modulo-N sequence number. So a (data or ack) message j means a message with unbounded sequence number j.

Consider what happens when a data message j is received. The receiver has access to mod(N,j) only. It looks for a "matching" k in the receive window, that is, k satisfying $\mathrm{nr} \le \mathrm{k} < \mathrm{nr} + \mathrm{rw}$ and $\mathrm{mod}(\mathrm{N}, \mathrm{k}) = \mathrm{cj}$. If a matching k exists it is assumed to be the message's unbounded sequence number, and so we would want k to equal j. If no matching k exists, the message is ignored and so k does not have to equal j.

What values of j ensure correct interpretation? If j lies in the receive window, it is correctly interpreted. If j is very much lower than nr or very much higher than $\mathrm{nr} + \mathrm{rw}$, then mod(N,j) wraps around and incorrectly matches a number in the receive window.

Consider decreasing values of j starting from nr, i.e., $\mathrm{nr} - 1$, $\mathrm{nr} - 2$, ... The first encountered value that is incorrectly interpreted is $\mathrm{nr} + \mathrm{rw} - 1 - \mathrm{N}$. So we want $\mathrm{j} \geq \mathrm{nr} - \mathrm{N} + \mathrm{rw}$ to hold.

Let j be received at time $\mathrm{t_0}$. We want $\mathrm{j} > \mathrm{nr(t_0)} - \mathrm{N} + \mathrm{rw(t_0)}$ to hold, where we use the notation x(t) to denote the value of a variable x at time t. It suffices if $\mathrm{j} > \mathrm{nr(t_0)} - \mathrm{N} + \mathrm{RW}$ holds (because rw is at most RW), which in turn holds if $\mathrm{j} > \mathrm{ns(t_0)} - \mathrm{N} + \mathrm{RW}$ holds (because $\mathrm{nr(t_0)}$ is at most $\mathrm{ns(t_0)}$).

Let j have been sent at time $\mathrm{t_1}$. Then $\mathrm{t_1} > \mathrm{t_0} - \mathrm{L}$ (from the maximum message lifetime). We have $\mathrm{j} \geq \mathrm{na(t_1)}$ (otherwise j would not have been sent). We have $\mathrm{na(t_1)} \geq \mathrm{ns(t_1)} - \mathrm{SW}$ (because sw is at most SW). So we have $\mathrm{j} \geq \mathrm{ns(t_1)} - \mathrm{SW}$. We also have $\mathrm{ns(t_1)} \geq \mathrm{ns(t_0)} - \mathrm{L}/\delta$, because ns can increase at most by 1 in time $\delta$. So we have $\mathrm{j} \geq \mathrm{ns(t_0)} - \mathrm{SW} - \mathrm{L}/\delta$.

So $\mathrm{j} > \mathrm{ns(t_0)} - \mathrm{N} + \mathrm{RW}$ holds if $\mathrm{SW} + \mathrm{L}/\delta \leq \mathrm{N} - \mathrm{RW}$, which is iff $\mathrm{N} \geq \mathrm{SW} + \mathrm{RW} + \mathrm{L}/\delta$.

This ensures that a data message j will not be too small. We also need to ensure that j does not become too high. We also need to ensure the same for ack messages. All this is left as an exercise (the resulting constraints are all subsumed by $\mathrm{N} \geq \mathrm{SW} + \mathrm{RW} + \mathrm{L}/\delta$).

## 6 A Connection-Management Protocol

This section describes a connection-management protocol. Traditional transport protocols, including TCP, identify successive incarnations by increasing, though not necessarily successive, incarnation numbers from some modulo-N space. Every entity uses a counter or a real-time clock to generate incarnation numbers for local incarnations.

Another feature of traditional transport protocols is that an entity stores a remote incarnation's number *only while it is connected to the remote incarnation.* This necessitates a **3-way handshake** for connection establishment. A client that wants to connect to a server sends a connection request with its incarnation number, say x. When the server receives this, it responds by sending a response containing x and a server incarnation number, say y. When the client receives the response, it becomes open to y and responds by sending an ack containing x and y. The server becomes open when it receives the ack. The server could not become open when it received the connection request containing only x, because it may have been a duplicate from previous now terminated connection.

A **2-way handshake** suffices for connection closing: an open entity sends a disconnect request that is acknowledged by the other entity.

A 2-way handshake also suffices for connection rejection. It is obvious that a server may have to reject a connection request of a client. What is not so obvious is that a client may have to reject a "connection request" of a server. Specifically, if a server receives an old connection request from a terminated incarnation of the client, the server attempts to complete the second "stage" of the 3-way handshake. In this case, the client has to reject the server.

The unreliable channels imply that a **k-way handshake** has the following structure: In every stage except the last, a message is sent repeatedly until the message of the next stage is received. The message of the last stage is sent only in response, otherwise the handshake would never terminate.

It is convenient to think of the protocol as a distributed system that is driven by user requests. Each user request causes the associated entity to initiate a 2 or 3-way handshake with the other entity. At each stage of the handshake, one entity learns something about the other entity and may issue an appropriate indication to its local user. At the end of the handshake, the protocol has 'served' the user request. The protocol's behavior can be complex because two handshakes can be executing concurrently, with one of them conveying information that is relevant to the other.

We now give a specification of the connection-management protocol.

A client entity maintains the following variables for each server s:

- status[s]: { CLOSED, OPENING, OPEN, CLOSING }; initially CLOSED. Status of client's relationship with server s. CLOSED iff client has no incarnation involved with s. OPENING means client has an incarnation requesting a connection with s. OPEN means client has an incarnation open to s. CLOSING means client has an incarnation closing a connection with s.

- lin[s]: { NIL, 0, 1, ... }; initially NIL. Local incarnation number. NIL if status[s] = CLOSED. Otherwise identifies client incarnation involved with server s.

- din[s]: { NIL, 0, 1, ... }; initially NIL. Distant incarnation number. NIL if status[s] equals CLOSED or OPENING. Otherwise identifies the incarnation of server s with which the client incarnation is involved.

A server entity maintains the following state variables for each client c:

- status[c]: { CLOSED, OPENING, OPEN }; initially CLOSED. Status of server's relationship with client c. CLOSED iff server has no incarnation involved with c. OPENING means server has an incarnation accepting a connection request from c. OPEN means server has an incarnation open to c.

- lin[c]: { NIL, 0, 1, ... }; initially NIL. Local incarnation number. NIL if status[c] = CLOSED. Otherwise identifies server incarnation involved with client c.

- din[c]: { NIL, 0, 1, ... }; initially NIL. Distant incarnation number. NIL if status[c] = CLOSED. Otherwise identifies the incarnation of client c with which the server incarnation is involved.

The messages of the protocol have the form (M, sid, rid, sin, rin), where M is the type of the message, sid is the sender's id, rid is the intended receiver's id, sin is the sender's incarnation number, and rin is the intended receiver's incarnation number. In some messages, sin or rin may be absent.

Each message is either a "primary" message or a "secondary" message. A **primary message** is sent repeatedly until a response is received or the maximum wait duration has elapsed. A **secondary message** is sent only in response to the reception of a primary message. Note that the response to a primary message may be another primary message, as in a 3-way handshake.

The messages sent by clients are as follows:

- (CR, sid, rid, sin ): Connection request. Sent when opening. Primary message.

- (CRRACK, sid, rid, sin, rin): Acknowledgement to connection request reply (CRR). Secondary message.

- (DR, sid, rid, sin, rin): Disconnect request. Sent when closing. Primary message.

- (REJ, sid, rid, rin): Reject response to a connection request reply (CRR) that is received when closed. The sin of the received CRR is used as the value of rin. Secondary message.

The messages sent by servers are as follows:

- (CRR, sid, rid, sin, rin ): Reply to connection request in 3-way handshake. Sent when opening. Primary message.

- (DRACK, sid, rid, sin, rin): Response to disconnect request. Secondary message.

- (REJ, sid, rid, rin): Reject response to a CR received when closed. The sin of the received message is used as the value of rin. Secondary message.

The events of the client and server entities are shown in Figures 5 and 6, assuming unbounded incarnation numbers. Figures 7, 8, and 9 illustrate some of the possible evolutions of the protocol.

### Modulo-N incarnation numbers

We now show how the unbounded incarnation numbers used above can be changed to modulo-N incarnation numbers. There is an intricate relationship between the modulo-N space of the incarnation numbers and the handshaking algorithms, much more so than in the case of data transfer, since the latter assumes correctly initialized users. To achieve correct interpretation of received cyclic incarnation numbers, it is necessary to have bounds on message lifetime, incarnation lifetime, wait duration, and recovery duration. Under the reasonable assumption that the incarnation lifetime dominates the wait and recovery durations, it is sufficient and necessary to have

$$\mathrm{N} \;\geq\; \frac{4\mathrm{L}+\mathrm{I}}{\alpha}$$

where L is the maximum message lifetime, I is the maximum incarnation lifetime, and $\alpha$ is the minimum time between successive incarnation creations at an entity. Most references in the literature incorrectly assume that $\mathrm{N} \;\geq\; 2\mathrm{L}/\alpha$ is sufficient.

The above bound may not be satisfiable for exceedingly long-lived incarnations, say, of the order of days. In that case, if we assume that the probability of two successive connections having identical modulo-N client and server incarnation numbers is negligible (it is approximately $1/\mathrm{N}^2$ under reasonable assumptions of incarnation lifetimes), then the following bound which does not depend on I suffices:

$$\mathrm{N} \;\geq\; \frac{4\mathrm{L}}{\alpha}$$

The events for modulo-N incarnation numbers are obtained by making the following changes, where $\mathrm{M} = 2\mathrm{L}/\alpha$:

- Redefine the domains of variables lin[s], din[s], lin[c], din[c] and message fields sin, rin to be { NIL, 0 , ... , N-1 }.
- The *equality* tests involving these variables and fields (e.g. sin = din[s]) are unchanged (but now each side is a modulo-N number).
- The inequality tests involving these variables and fields are changed as follows:
  - Replace the test sin > din[s] in the client by $1 \leq \mathrm{mod}(\mathrm{N}, \mathrm{sin} - \mathrm{din[s]}) \leq \mathrm{M}$
  - Replace the test sin > din[c] in the server by $1 \leq \mathrm{mod}(\mathrm{N}, \mathrm{sin} - \mathrm{din[c]}) \leq \mathrm{M}$

# 7 A Transport Protocol

A transport protocol between a client entity c and a server entity s consists of a connection management protocol augmented with two data-transfer protocols, one for data transfer from c to s and another for data transfer from s to c. At each entity, the data-transfer protocol is initialized each time the entity becomes open and its events are executed only while open. The data-transfer messages are augmented by incarnation number fields, which are used by receiving entities to filter out data-transfer messages of old connections.

We illustrate with the protocols of the previous sections. Start with the connection-management protocol of Section 6 between c and s. Add two sliding-window protocols, one from c to s and one from s to c, as follows:

- At each entity, introduce variables ng, ns, na, sw and sbuff for the outgoing data transfer, and nd, nr, rw and rbuff for the incoming data transfer. These data-transfer variables are initialized whenever the entity becomes open. Whenever the entity becomes closed, these variables are deallocated.
- Modify the client as follows. Add status[s]=OPEN to the enabling condition of every data-transfer event. Add sin and rin fields to the sliding-window protocol messages. When a data-transfer message is sent, sin is set to the local incarnation number lin[s] and rin is set to the remote incarnation number din[s]. When a data-transfer message is received, first test for status[s]=OPEN, sin=din[s] and rin=lin[s]. If the test fails, ignore the message, otherwise process the message as in the sliding-window protocol specification.
- Modify the server similarly.

There are various ways to extend the transport protocol, and we mention some of them below.

- The messages of the two data-transfer protocols can be combined. For example, the data messages sent by an entity can have additional fields to "piggy-back" acknowledgement information for incoming data, i.e., fields for nr and rw. This is done in TCP.
- The above protocol allows either user to close a connection at any point, without waiting for data transfer to be completed. An alternative is so-called "graceful closing", where a user can close only its outgoing data transfer. The user must continue to handle incoming data until the remote user issues a close also. TCP has graceful closing. It is a simple matter to add graceful closing to a protocol that does not have it (How?).
- It is possible to merge connection establishment, data transfer, and connection termination. The connection request can contain data, which would be delivered after the server becomes open. The connection request can also indicate that after the data is delivered the connection is to be closed. TCP allows this.

## Comparison with TCP

TCP uses a single 32-bit cyclic sequence number space to identify both incarnations and data blocks. When an incarnation is created at an entity, an **initial sequence number** is chosen and assigned to the incarnation. Successive new message sent by the incarnation, whether of connection-management or data-transfer, occupy increasing sequence numbers starting from this initial sequence number.

TCP messages integrate both data transfer and connection management. Every TCP message has fields indicating the sequence number of the message, the next sequence number expected, the data segment (if any), the segment length, and receive window size. A connection-management message that requires an acknowledgement is considered to use up a sequence number. So if the last such message sent by an entity had sequence number n, then the next new connection management message sent by the entity, whether or not it requires an acknowledgement, would have the sequence number n+1. The remote entity can acknowledge a connection-management message by sending a message of any type with its next expected sequence number field equal to n+1.

The TCP messages SYN, SYN-ACK, ACK, FIN, FIN-ACK and RST correspond respectively to the messages CR, CRR, CRRACK, DR, DRACK, REJ of our protocol. An example evolution of TCP is shown in Figure 10.

TCP provides **balanced-opening**, a service that is outside the client-server paradigm. Here, if two entities request connections to each other at the same time, a *single* connection is established. In fact, TCP's algorithm for balanced opening is flawed: in certain situations it can result in valid connection requests being rejected and invalid connection requests leading to connections. Fortunately, no application seems to use TCP's balanced-opening service.

## 8 Minimum Latency Transport Protocols

The delay in connection establishment incurred by the 3-way handshake is unacceptable for many transaction-oriented applications (such as remote procedure calls). Note that although transaction data can be sent with a connection request, the server cannot process the transaction until it confirms that this is a new request. This has motivated the development of transport protocols where the server can determine the "newness" of a connection request as soon as it is received, thereby achieving connection establishment with a 2-way handshake, which is the minimum possible latency.

To achieve this, the server has to retain information about clients even when it is not connected to them. Consider a 3-way handshake between client incarnation x and server incarnation y. If the server had remembered the incarnation number, say z, that the client had previously used when it connected to the server, then the server could determine that the connection request with x was new (because x would be greater than z). In that case, the server could have become open at once, resulting in a 2-way handshake connection establishment. A server cannot be expected to indefinitely remember the last incarnation number of every client to which it was connected, due to the enormous number of clients in a typical internetwork. However, a cacheing scheme is feasible, and several have been proposed, culminating in a proposed modification to TCP.

An alternative to cacheing is to use timer-based mechanisms. Here also, a server is required to maintain information on each client it has served for a duration comparable to that in cache-based mechanisms (the major component in both is the network-layer message lifetime). In most timer-based protocols, if a client's entry is removed before the specified duration, e.g., due to a crash or memory limitation, then the server can incorrectly accept old connection requests of that client. There is a timer-based protocol, called SCMP, that overcomes this problem by assuming synchronized clocks. It maintains correctness but it may reject new connections for a period of time depending on clock skews and other parameters. In any case, timer-based approaches do not have a back-up 3-way handshake.

## 9 Conclusions

We have described the services expected of a transport layer and presented protocols that achieve these services. In particular, the protocol for reliable transport service is comparable to TCP and illustrates the inner workings of TCP. We have also outlined the basic approach to congestion control in TCP/IP networks.

Detailed analysis and extensions of the data transfer and connection management protocols described in this note may be found in the following:

- Data-transfer protocol with selective acks and rejects:
  Shankar, A.U. 1989. Verified Data Transfer Protocols with Variable Flow Control, *ACM Transactions on Computer Systems.* 7(3):281-316. August 1989.

- Connection management protocols including cacheing-based protocols and info on timer-based protocols:
  Shankar, A.U. and Lee, D. 1995. Minimum Latency Transport Protocols with Modulo-N Incarnation Numbers, *IEEE/ACM Transactions on Networking.* 3(3):255-268. June 1995.

- Formal composition of data transfer and connection management:
  Shankar, A.U. 1991. Modular Design Principles for Protocols with an Application to the Transport Layer, *Proceedings of the IEEE.* 79(12):1687-1709. December 1991.

- Balanced opening and the flaws of TCP:
  Murphy, S.L. and Shankar, A.U. 1991. Connection Management for the Transport Layer: Service Specification and Protocol Verification, *IEEE Transactions on Communications.* 39(12):1762-1775. December 1991.

**Client entity c: events concerning server s**

```
ConnectRequest(s)
 ec:  status[s] := CLOSED
 ac:  status[s] := OPENING ;  lin[s] := new incarnation number

DisconnectRequest(s)
 ec:  status[s] = OPEN
 ac:  status[s] := CLOSING

Abort(s)
 ec:  status[s] ≠ CLOSED  and  "response timeout"
 ac:  status[s] := CLOSED ;  lin[s] := NIL ;  din[s] := NIL

SendCR(s)
 ec:  status[s] = OPENING
 ac:  Send ( CR, c, s, lin[s] )

SendDR(s)
 ec:  status[s] = CLOSING
 ac:  Send ( DR, c, s, lin[s], din[s] )

Receive ( CRR, s, c, sin, rin )
 ac:  if status[s] = OPENING  and  rin = lin[s]  then
         status[s] := OPEN ;  din[s] := sin ;
         Send ( CRRACK, c, s, lin[s], din[s] )

      else if status[s] = OPEN  and  rin = lin[s]  and  sin = din[s]  then
         // duplicate CRR
         Send ( CRRACK, c, s, lin[s], din[s] )

      else if status[s] = OPEN  and  rin = lin[s]  and  sin>din[s]  then
         // server crashed, recovered, responding to old CR
         Send ( REJ, c, s, sin ) ;  status[s] := CLOSED ;
         din[s] := NIL ;  lin[s] := NIL

      else if (status[s] is CLOSED or CLOSING)  then Send ( REJ, c, s, sin )

Receive ( REJ, s, c, rin )
 ac:  if (status[s] is OPENING or CLOSING)  and  rin = lin[s]  then
         status[s] := CLOSED ;  din[s] := NIL ;  lin[s] := NIL
      // else status[s] is OPEN or CLOSED; do nothing

Receive ( DRACK, s, c, sin, rin )
 ac:  if status[s] = CLOSING  and  rin = lin[s]  and  sin = din[s]  then
         status[s] := CLOSED ;  din[s] := NIL ;  lin[s] := NIL
      // else status[s] is OPENING or OPEN or CLOSED; do nothing
```

Figure 5: Client events of connection management protocol

**Server entity s: events concerning client c**

```
Abort(c)
 ec: status[c] $\neq$ CLOSED  and  “response timeout”
 ac: status[c] := CLOSED ;  lin[c] := NIL ;  din[c] := NIL

SendCRR(c)
 ec: status[c] = OPENING
 ac: Send ( CRR, s, c, lin[c], din[c] )

Receive ( CR, c, s, sin )
 ac: if status[c] = CLOSED  and  “rejecting connnections”  then
         Send ( REJ, s, c, sin ) ;
     else if status[c] = CLOSED  and  “accepting connections”  then
         lin[c] := new incarnation number ;
         status[c] := OPENING ;  din[c] := sin
     else if status[c] = OPENING  and  sin>din[c]  then
         // previous din[c] value was from some old CR
         din[c] := sin
     else if status[c] = OPEN  and  sin>din[c]  then
         // client crashed, reconnecting
         if “willing to reopen”  then
             lin[c] := new incarnation number ;
             din[c] := sin ; status[c] := OPENING
         else status[c] := CLOSED ;  lin[c] := NIL ;  din[c] := NIL
     // else status[c] = OPEN  and  sin $\leq$ din[c]; do nothing

Receive ( CRRACK, c, s, sin,, rin, )
 ac: if status[c] = OPENING  and  sin = din[c]  and  rin = lin[c]  then
         status[c] := OPEN
     // else status[c] is OPEN or CLOSED; do nothing

Receive ( DR, c, s, sin, rin )
 ac: if status[c] = OPEN  and  sin = din[c]  and  rin = lin[c]  then
         Send ( DRACK, s, c, lin[c], din[c] ) ;
         status[c] := CLOSED ;  lin[c] := NIL ;  din[c] := NIL
     else if status[c] = CLOSED  then Send ( DRACK, s, c, rin, sin ) ;
     // else status[c] = OPENING; do nothing

Receive ( REJ, c, s, rin )
 ac: if status[c] = OPENING  and  rin = lin[c]  then
         status[c] := CLOSED ;  lin[c] := NIL ;  din[c] := NIL
     // else status[c] is OPEN or CLOSED; do nothing
```

Figure 6: Server events of connection management protocol

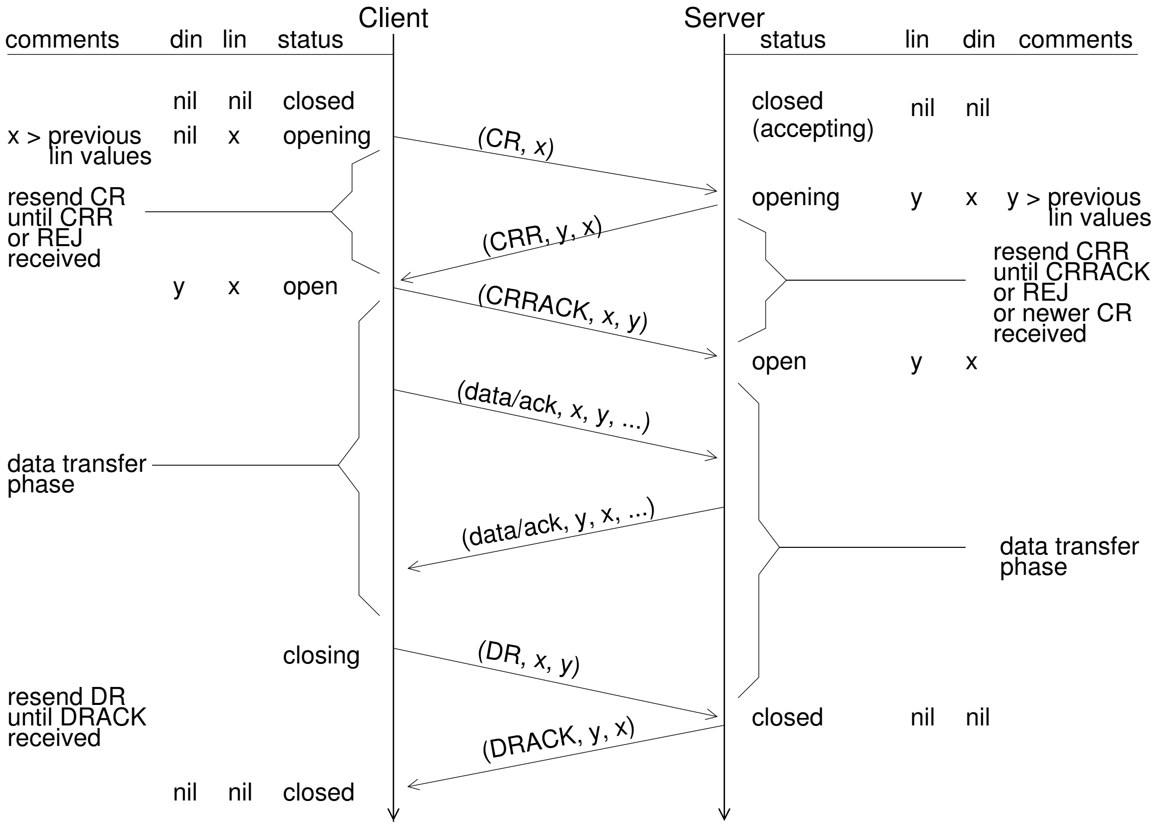

Figure 7: Connection management: “normal” operation

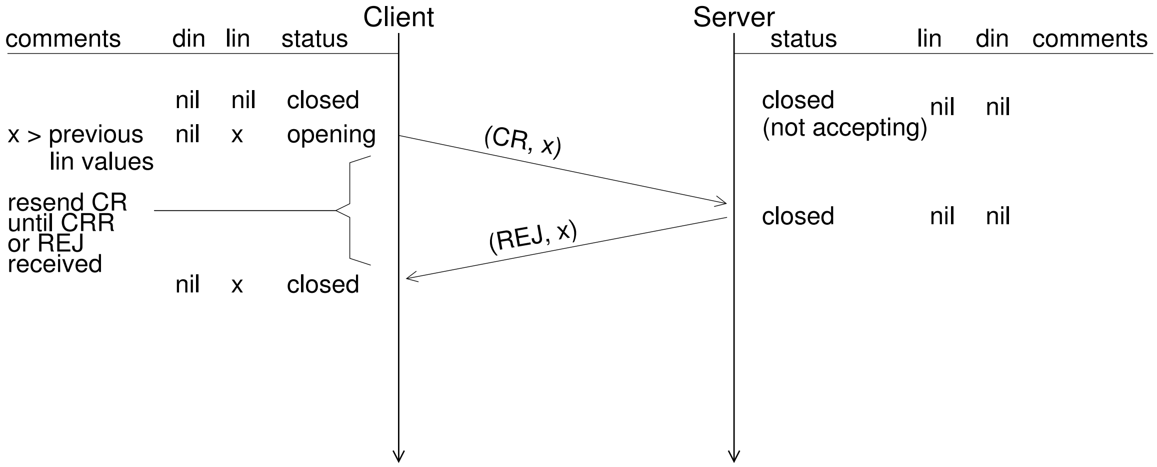

Figure 8: Connection management: connection rejection

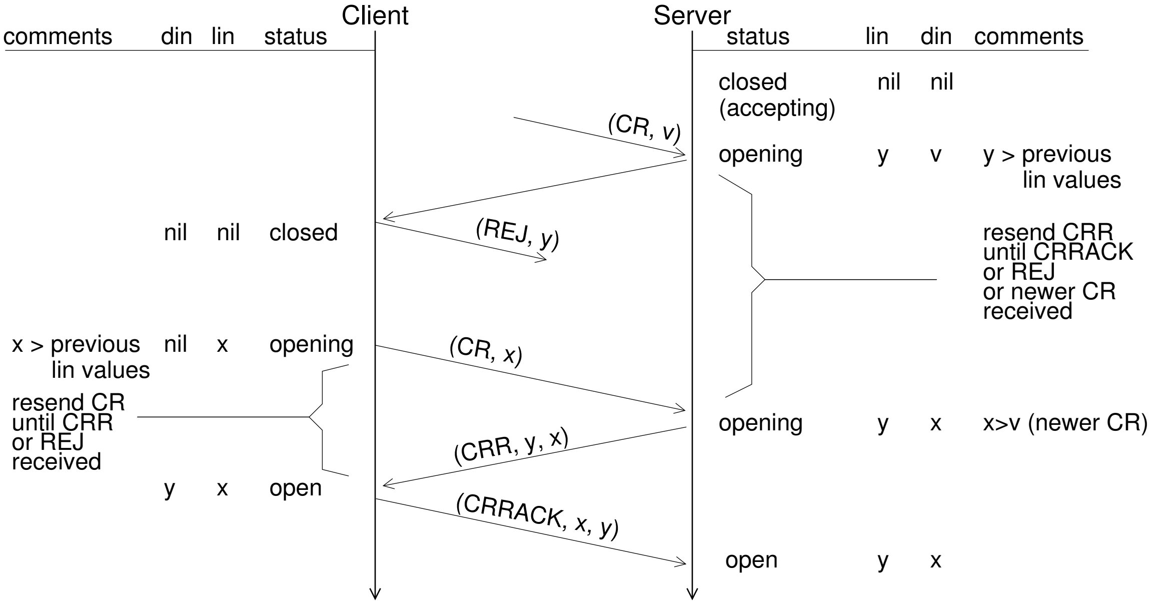

Figure 9: Connection management: old connection request at server

Example evolution of TCP protocol

– Entity variables as follows
  status: closed, opening, open, etc
  ssn: send sequence number to use on next new message
  asn: send sequence number to be next acknoledged
  rsn: receive seqence number (next expected send sequence number of remote)

– Every message includes fields (A, B, C, D) where
  A: message type
  B: start sequence number of message
  C: length (i.e., number of sequence numbers occupied by message)
  D: receive seqence number of sender

– x is initial sequence number of Client

– y is initial sequence number of Server

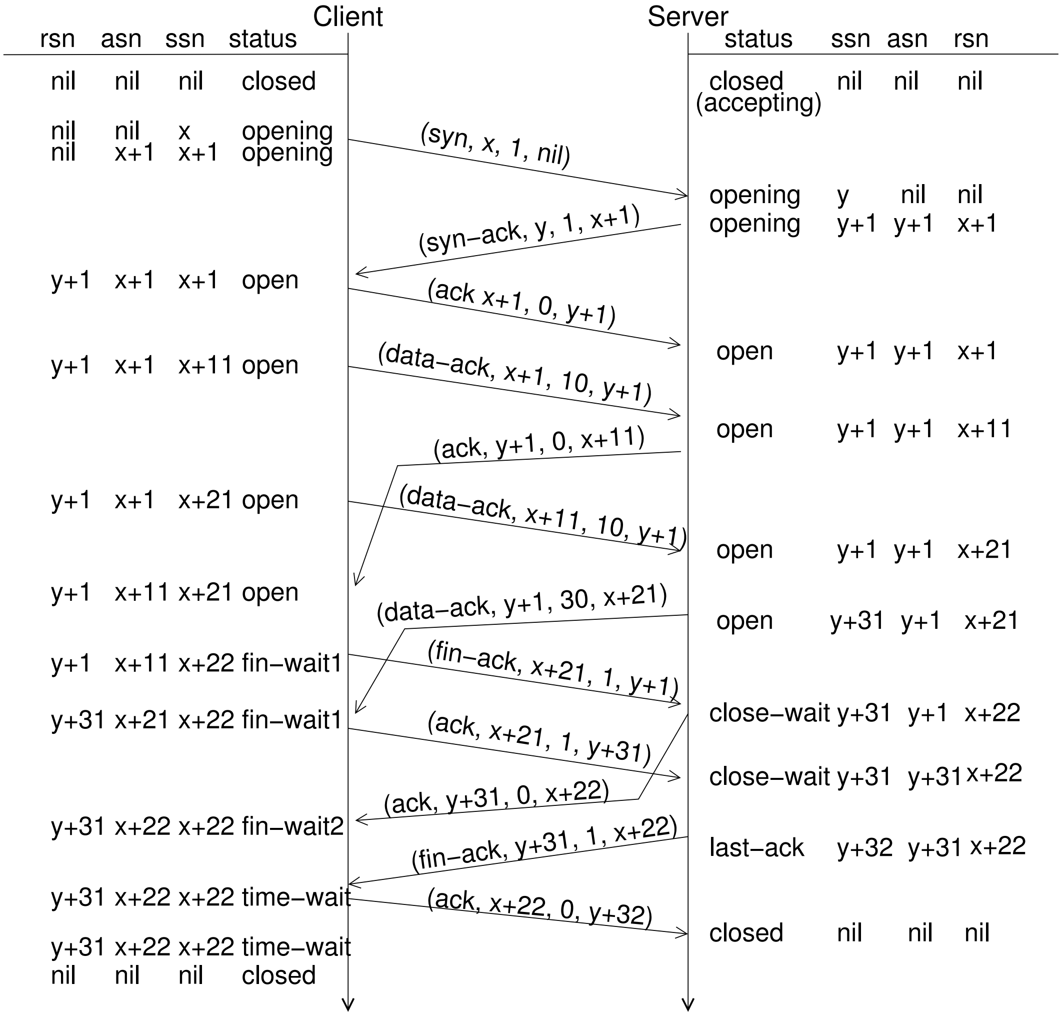

Figure 10: TCP evolution example