Computer Networks

Chapter 06

prof. dr ir Maarten van Steen

Vrije Universiteit Amsterdam
Faculty of Science
Dept. Mathematics and Computer Science
Room R4.20. Tel: (020) 444 7784

steen@cs.vu.nl

Contents

Transport Layer (1/2)

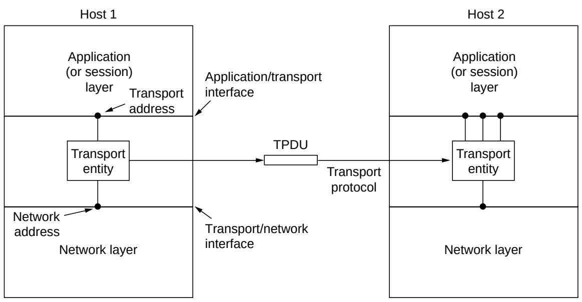
Essence: The transport layer is responsible for completing the services of the underlying network to the extent that application development can take place:

- provide reliable connection-oriented services
- provide unreliable connectionless services
- provide parameters for specifying quality of services

Important: we're talking about efficient and cost-effective services, in particular reliable connections.

Note: depending on the services offered by the network layer, the added functionality in the transport layer can vary considerably.

Transport Layer (2/2)



Note: The issue here is that the network layer is in the hands of carriers: organizations that offer a (generally wide-area) computer network to their clients. Clients have no say in what the carrier actually offers.

Consequence: If we want to develop applications that are independent of the particular services offered by a carrier, we'll have to devise a standard communication interface and implement that interface at the client's sites. The transport layer contains such implementations.

Transport Layer Interface

Example: Consider the Berkeley **socket interface**, which has been adopted by most UNIX systems, as well as Windows 95/NT:

SOCKET	Create a new communication endpoint
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept N connections
ACCEPT	Block until someone remote wants to establish a connection
CONNECT	Attempt to establish a connection
SEND	Send data over a connection
RECEIVE	Receive data over a connection
CLOSE	Release the connection

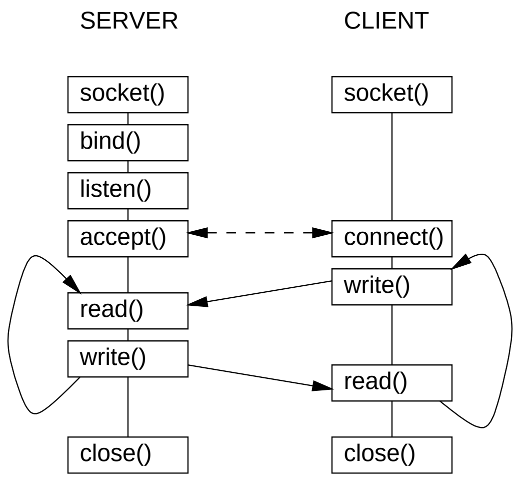
Socket Communication

- The client and server each **bind** a transport-level address and a name to the locally created socket.
- The server must **listen** to its socket, thereby telling the kernel that it will subsequently wait for connections from clients.
- After that, the server can **accept** or **select** connections from clients.
- The client **connects** to the socket. It needs to provide the transport-level address by which it can locate the server.

After a connection has been accepted (or selected), the client and server communicate through **read/write** operations on their respective sockets.

Communication ends when a connection is **closed**.

Connection-Oriented Socket Communication



Question: What about connectionless communication?

Sockets – Server Side

```
serverAddress : TransportAddress /* Publicly known address */
...
PROCESS Server IS
  clientSocket : Socket; /* Private socket */
  ...
  BEGIN
    serverSocket := NEW Socket;
    serverSocket.bind(serverAddress);
    serverSocket.listen(maxConnections);
    LOOP
        serverSocket.accept(clientSocket);
        clientSocket.read(request); /* receive */
        clientSocket.write(answer); /* send */
        clientSocket.close();
    END LOOP;
END Server;
```

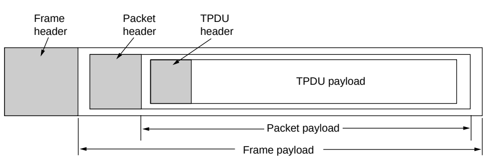
Sockets – Client Side

```
serverAddress : TransportAddress /* Publicly known address */
...
PROCESS Client IS
    clientAddress : TransportAddress; /* Private address */
    clientSocket : Socket;      /* Private socket  */
    ...
    BEGIN
    clientAddress := NEW TransportAddress;
    clientSocket := NEW Socket;

    clientSocket.bind(clientAddress);
    LOOP
        IF clientSocket.connect(serverAddress)
            THEN EXIT;
            ELSE sleep(1);
        END IF;
    END LOOP;

    clientSocket.write(request); /* send */
    clientSocket.read(answer); /* read */
    clientSocket.close();
    END Client;
```

Question: What am I doing in the loop?

Some Observations

Note 1: Messages sent by clients are encapsulated as **transport protocol data units** (TPDUs) to the network layer:

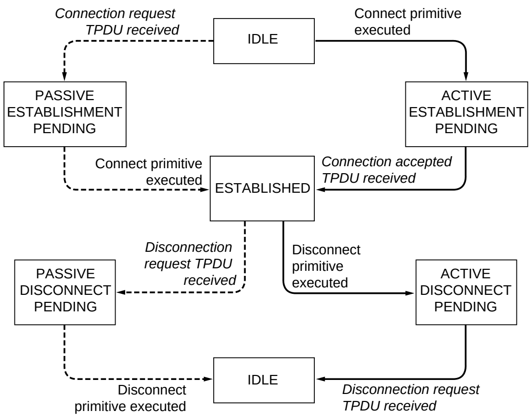


Note 2: A real hard part is establishing and releasing connections. The model can be either symmetric or asymmetric:

Symmetric: one side sends a disconnect request, and waits for the other to acknowledge that the connection is closed. Yes, there are some problems with this model. In fact, it turns out it is impossible to implement.

Asymmetric: one side just closes the connection, and that's it. Yes, it's simple, but you may lose some data this way. Not really acceptable.

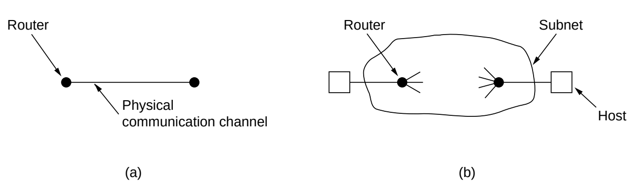
Sockets – State Diagram



Note: Dashed lines are server state transitions; solid lines client state transitions.

Transport Protocol

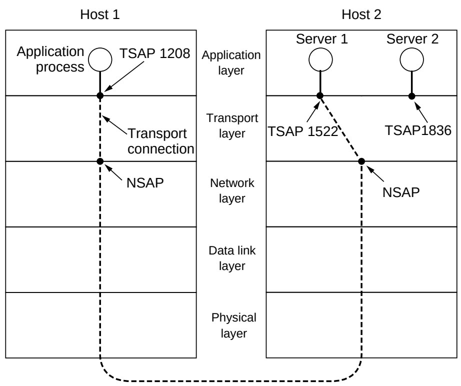
Observation: transport protocols strongly resemble those in the data link layer: e.g. lots of error and flow control. Big differences when it comes to solutions!



- explicit addressing
- establishing, maintaining, and releasing connections
- the many connections require different solutions
- handle effects of subnet storage capabilities

Addressing

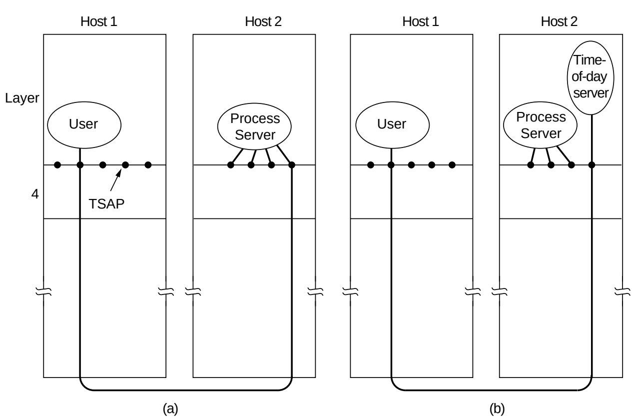
Note: Each layer has its own way of dealing with addresses. In IP, a **transport service access point** is an IP address with a port number.



Question: How do we get to know where the other party is?

Service Locations Fixed Addresses

General solution: have a single process, located at a **well-known** address, handle a large number of services (inetd in the UNIX world):



Service Locations Unknown Addresses

Problem: Sometimes you just can't have a process handle all services, e.g. because the service requires special hardware (file server) ⇒ find address of the server

Solution: you'll have to use a name server.

Question: Great, so how do we find the name server?

Next problem: A name server returns a TSAP, not an NSAP. So how do we get to know the network address?

Question: at what level do you think name servers fit in?

Connection Establishment

Basic idea: To establish a connection, you send off a connection request to the other end. The other end then accepts the connection, and returns an acknowledgment.

Big problem: Suppose you don't get an answer, so you do another request.

- Your first request didn't make it: no harm done.
- The ack didn't make it back: you're establishing a second connection, but this can probably be detected.
- Your first request didn't make it *yet*: now you're really making a second connection and no one knows you didn't do this on purpose.

Main cause: The network has storage capabilities, and unpredictable delays. This means that things can pop up out of the blue.

Attacking Duplicates (1/2)

Solution: Restrict the lifetime of TPDUs – if the maximum lifetime is known in advance, we can be sure that a previous packet is discarded and that it won't interfere with successive ones.

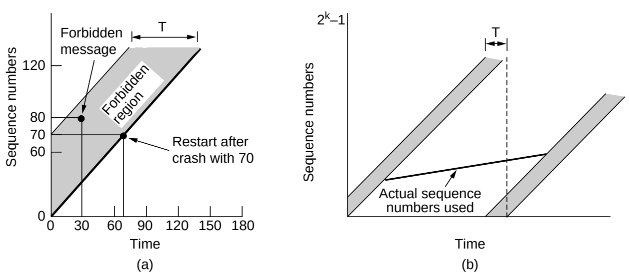
Basic idea: Assign sequence numbers to TPDUs, and let the sequence number space be so large that no two outstanding TPDUs can have the same number.

Problem: When a host crashes, it has to start numbering TPDUs again. So, where does it start?

- You can't just wait the maximum packet lifetime T and start counting from the start again: in wide-area systems, T may be too large to do that.
- The point is that you must avoid that an *initial* sequence number corresponds to a TPDU still floating around. So, just find the right initial number.

Attacking Duplicates (2/2)

Solution: Assign sequence numbers in accordance to clock ticks, and assume that the clock continues ticking during a crash. This leads to a **forbidden region**:



Every time you want to assign a next sequence number, check whether that number is in the forbidden region.

Watch it: when sequence numbers are assigned at a lower pace than the clock ticks, we may enter the region "from the top." Likewise, assigning them too fast makes you enter the region "from the bottom."

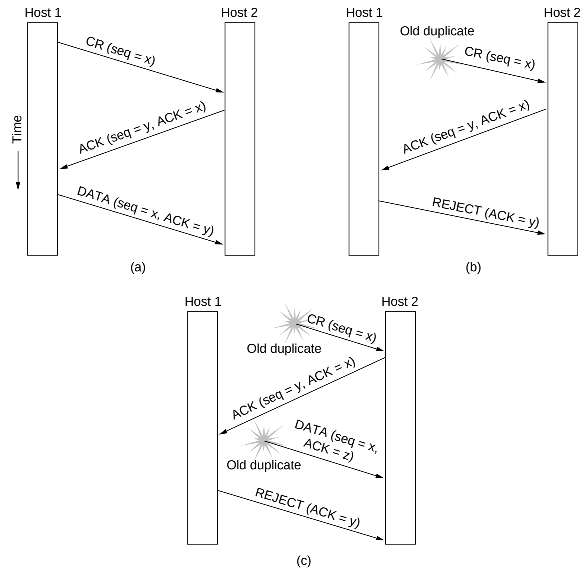
Error-Free Connection Establishment (1/2)

Problem: Great, we have a way of avoiding duplicates, but how do we get a connection in the first place?

Note: One way or the other we have to get the sender and receiver to agree on initial sequence numbers. We need to avoid that an old (unnumbered) connection request pops up.

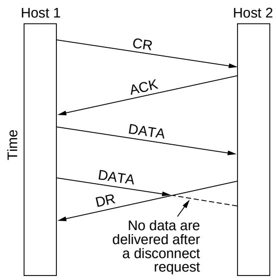
Error-free Connection Establishment (2/2)

Solution: Three-way handshake.



Error-free Connection Release

Asymmetric release: one party just closes down the connection. May result in loss of data:



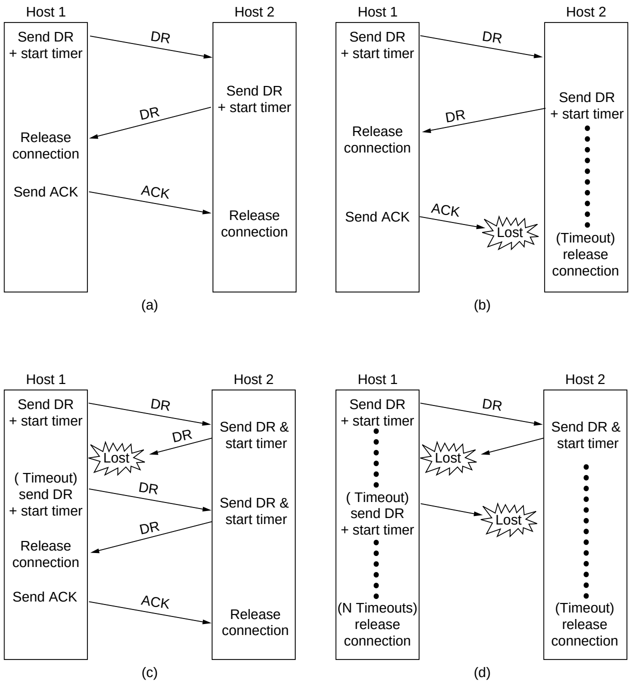
Symmetric Connection Release (1/2)

Big problem: Can we devise a solution to release a connection such that the two parties will *always* agree. The answer is simple: **NO**.

- **Normal case:** Host 1 sends disconnect request (DR). Host 2 responds with a DR. Host 1 acknowledges, and ACK arrives at host 2.
- **ACK is lost:** What should host 2 do? It doesn't know for sure that its DR came through.
- **Host 2's DR is lost:** What should host 1 do? Of course, send another DR, but this brings us back to the normal case. This still means that the ACK sent by host 1 may still get lost.

Pragmatic solution: Use timeout mechanisms. This will catch most cases, but it is never a fool-proof solution: the initial DR and all retransmissions may still be lost, resulting in a **half-open connection**.

Symmetric Connection Release (2/2)



Flow Control and Buffering

Main problem: Hosts may have so many connections that it becomes infeasible to allocate a fixed number of buffers per connection to implement a proper sliding window protocol $\Rightarrow$ we need a dynamic buffer allocation scheme.

- With an unreliable network, i.e. unreliable datagram service provided by the network layer, the sender will have to buffer TPDUs until they are acknowledged.
- The receiver may decide to drop incoming TPDUs if it has no buffer space available.
- With a reliable network, the sender will still have to buffer a TPDU until it is acknowledged: the network layer cannot help here! (**WHY NOT?**)

In general: the sender and receiver need to negotiate the number of TPDUs that can be transmitted in sequence, only because buffer space no longer comes for free.

Buffer Reservation

Basic idea: The sender requests a number of buffers at the receiver's side when opening a connection. The receiver responds with a **credit grant**. After that, the receiver grants more credit when bufferspace becomes available:

	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	• • •	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	A may now send 5
12	←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	A is now blocked again
15	←	<ack = 6, buf = 0>	←	A is still blocked
16	• • •	<ack = 6, buf = 4>	←	Potential deadlock

Question: what can we do about the potential deadlock?

Flow Control – The Network

Problem: Now that we've adjusted the transmission rate between the sender and receiver, let's consider the network capacity as well: it may not be enough for what the sender and receiver want to do.

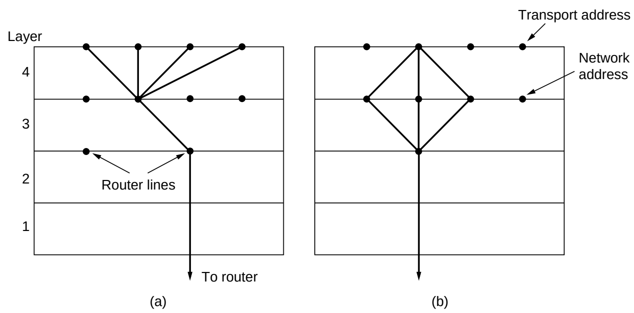
Issue: If the network can handle c TPDUs per second, and takes a total of r seconds to transmit, propagate, queue and process the TPDU, and to send an ACK, the sender need only maintain $c \cdot r$ buffers. More buffers is overkill of the network.

Solution: Let the sender estimate c and r (**HOW?**) and adjust its own number of buffers.

Multiplexing

Basic idea 1: Assuming that the network offers only a limited number of virtual circuits, or that a user doesn't want to pay so much, then use a single circuit for several connections ⇒ **upward multiplexing**.

Basic idea 2: If a user requires a lot of bandwidth that cannot be supported by a single network virtual circuit, use several circuits for a single connection ⇒ **downward multiplexing**.



Crash Recovery (1/2)

Problem: A host responds to the receipt of a TPDU by performing an operation and returning an acknowledgment. How should the sending host respond when the receiving host crashes before, during, or after its response?

Crash Recovery (2/2)

Situation: Assume the sender is informed that the receiver has just recovered from a crash. Should the sender retransmit the TPDU it just sent, or not? Distinguish between:

- S0: sender had no outstanding (unacknowledged) TPDUs
- S1: sender had one outstanding TPDU

Strategy used by receiving host

Strategy used by sending host	First ACK, then write: AC(W)	AWC	C(AW)	First write, then ACK: C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Example Protocol

Service Primitives	
LISTEN	Block connection request comes in
CONNECT	Attempt to establish a connection
SEND	Send data over a connection
RECEIVE	Receive data over a connection
DISCONNECT	Release the connection

Network Layer Packets	
CALL REQUEST	Sent to establish a connection
CALL ACCEPTED	Response to CALL REQUEST
CLEAR REQUEST	Sent to release a connection
CLEAR CONFIRM	Response to CLEAR CONNECTION
DATA	Used to transport data
CREDIT	For managing the window

State of a Connection	
IDLE	Not yet established
WAITING	CONNECT called; CALL REQ. sent
QUEUED	CALL REQ. arrived; LISTEN not called
ESTABLISHED	Connection established
SENDING	Waiting for permission to send TPDU
RECEIVING	RECEIVE has just been called
DISCONNECTING	DISCONNECT ahs just been called

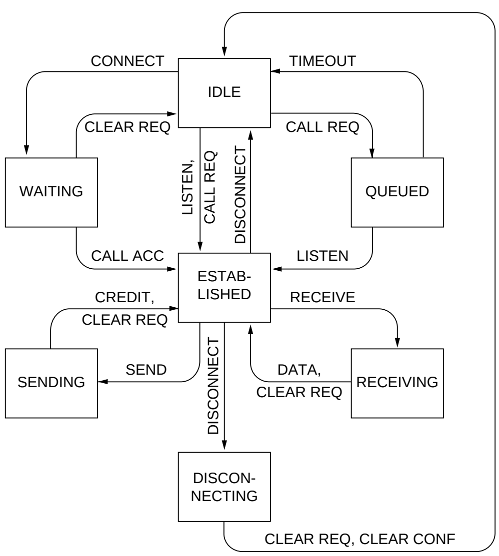
Example Protocol – FSM (1/2)

		State						
		Idle	Waiting	Queued	Established	Sending	Receiving	Dis-connecting
Primitives	LISTEN	P1: ~/Idle P2: A1/Estab $\overline{P2}$: A2/Idle		~/Estab				
	CONNECT	P1: ~/Idle $\overline{P1}$: A3/Wait						
	DISCONNECT				P4: A5/Idle $\overline{P4}$: A6/Disc			
	SEND				P5: A7/Estab $\overline{P5}$: A8/Send			
	RECEIVE				A9/Receiving			
Incoming packets	Call_req	P3: A1/Estab $\overline{P3}$: A4/Queu'd						
	Call_acc		~/Estab					
	Clear_req		~/Idle		A10/Estab	A10/Estab	A10/Estab	~/Idle
	Clear_conf							~/Idle
	DataPkt						A12/Estab	
	Credit				A11/Estab	A7/Estab		
Clock	Timeout			~/Idle				

Predicates
P1: Connection table full
P2: Call_req pending
P3: LISTEN pending
P4: Clear_req pending
P5: Credit available

Actions
A1: Send Call_acc
A2: Wait for Call_req
A3: Send Call_req
A4: Start timer
A5: Send Clear_conf
A6: Send Clear_req
A7: Send message
A8: Wait for credit
A9: Send credit
A10: Set Clr_req_received flag
A11: Record credit
A12: Accept message

Example Protocol – FSM (2/2)



UDP

Essence: The User Datagram Protocol is essentially just a transport-level version of IP.

32 Bits

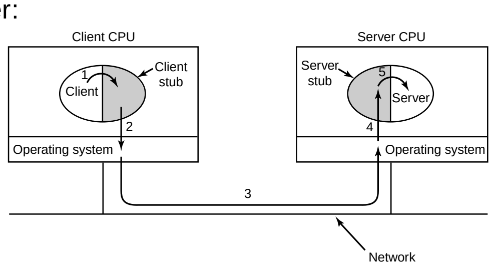
Source port	Destination port
UDP length	UDP checksum

Observation: UDP is simple: no flow control, no error control, no retransmissions

Question: So why not use IP instead?

RPC

Observation: UDP is widely used for simple client-server communication in which a procedure is made available to remote clients (**Remote Procedure Call**). The call (including its parameters) is shipped to the server:

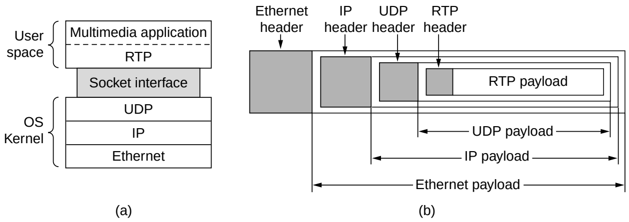


1. Client calls the procedure at a **local stub**
2. Client stub **marshalls** request: it puts everything into a (UDP) message
3. The message is transferred over the network
4. The server stub unmarshalls the message...
5. ... and calls the local implementation of the procedure.

Question: What's the difficulty with RPCs?

RTP

Problem: Can we support multimedia streaming over the Internet? The **Real-Time Transport Protocol** provides some best-effort support.



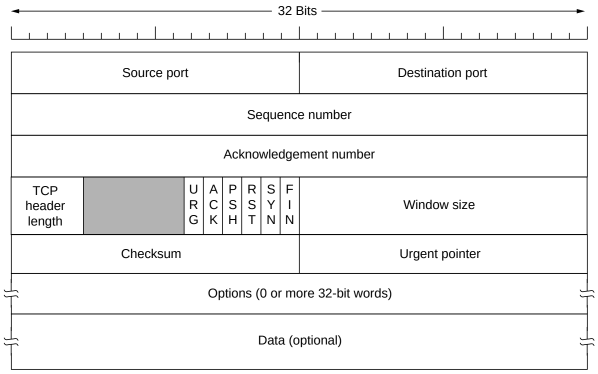
Essence: RTP essentially just multiplexes a number of multimedia streams into a single UDP stream. The receiver is responsible for *compensating* missing packets (which is highly application dependent).

Real-time: RTP packets can be timestamped: packets belonging to the same substream can receive a timestamp indicating how far off they are with respect to their predecessor. This approach allows the system to reduce jitter. In addition, timestamps can be used to synchronize multiple substreams.

Transmission Control Protocol (TCP)

- Connection-oriented service that supports **byte streams** (not message streams). A sender may send eight 512-byte packets that are received as two chunks of 1024 bytes, and one of 2048 bytes.
- Transport address consists of a 16-bit **port number**, which augments the underlying IP address.
- TCP ensures reliable, point–to–point connections. No support for multicasting or broadcasting.
- A TCP TPDU is called a **segment**, consisting of (minimal) 20-byte header, and maximum total length of 65,535 bytes. A segment is fragmented by the network layer when it is larger than the network's **maximum transfer unit** (MTU).

TCP Header



- Acknowledgments are piggybacked when ACK = 1.
- SYN is for connection setup (ACK = 0: request; ACK = 1: accepted).
- FIN is for connection release. Data sent before the release is not lost.
- URG indicates immediate processing and transmission: the receiver is signalled.

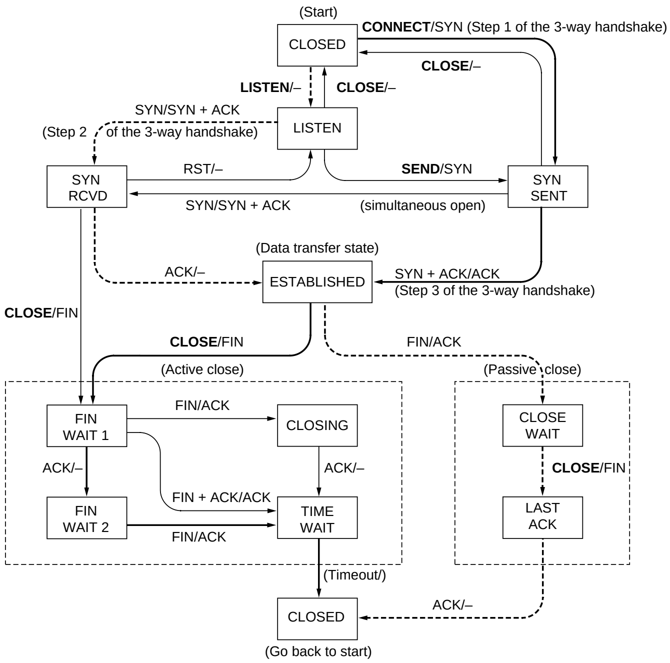
TCP Connection Management

Connection establishment: Uses three-way handshake protocol.

Connection release: To be thought of as independent releases of two simplex connections:

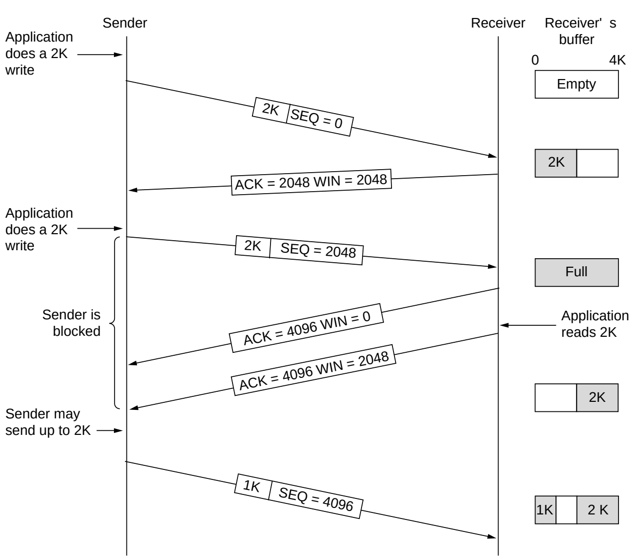
State	Textlist
CLOSED	No connection active or pending
LISTEN	Server waiting for conn. request
SYN RCVD	Conn. request has arrived; wait for ACK
SYN SENT	Conn. request just sent; wait for SYN+ACK
ESTABLISHED	Data can be sent and received
FIN WAIT 1	Client just sent conn. release
FIN WAIT 2	Server just agreed to release connection
TIMED WAIT	Wait for all packets to die
CLOSING	Client & server both tried to close
CLOSE WAIT	Other side initiated release
LAST ACK	Wait for all packets to die

TCP Connection Management Release



TCP Window Management (1/2)

Basic idea: The receiver sends an acknowledgment for the next byte that can be sent in the current stream, and the maximum number of bytes that may be sent.



TCP Window Management (2/2)

Important: The TCP entity is not obliged to immediately transmit data that the application hands over: it can do as much buffering as it likes. Same goes for acknowledgments.

Example: Interactive, character-oriented applications. Rather than sending one byte at a time, buffer as much characters as possible until the previous batch is acknowledged (Nagle's algorithm). **Note:** we're always stuck to at least 40 bytes of overhead per TPDU.

Example: Avoid the **silly window syndrome** where the server is reading one byte at a time (and acknowledges one at a time). Instead, the receiver should wait until it can receive a reasonable amount of bytes in a row.

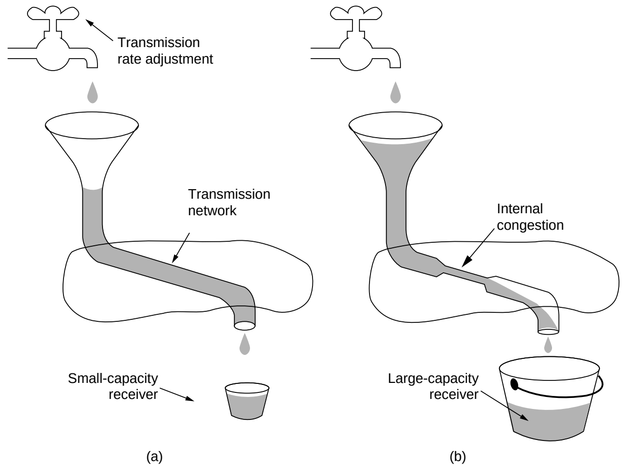
TCP Congestion Control (1/2)

Problem: As before, the transport layer has to take into account that the underlying network can be the bottleneck. Question is how to detect and react to congestion.

Solution: use a **congestion window** next to the window granted by the receiver. The actual window size is the minimum of the two.

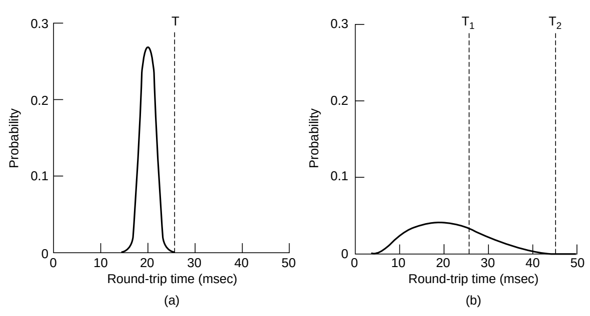
- Initialize congestion window to maximum segment size to be used in the connection. Send it off. If it gets acknowledged, double the size. Repeat until failure. Leads to initial congestion window size (**slow start**).
- In addition, use a **threshold**. On a timeout, lower the threshold to 50 % of the congestion window size, do a slow start (exponential) until new threshold, and *add* maximum segment size to congestion window size after that (linear growth).

TCP Congestion Control (2/2)



TCP Timer Management

Main issue: How do we determine the best timeout value for retransmitting segments in the face of a large standard deviation of round-trip delays:



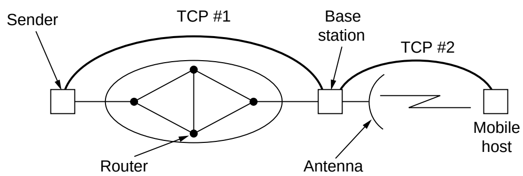
RTT	best current estimate of round-trip delay
D	estimate of deviation of round-trip delays
M	measured round-trip delay

$$
\begin{aligned}
RTT &= \alpha RTT + (1-\alpha)M \\
D &= \alpha D + (1-\alpha)|RTT - M| \\
\textit{timeout} &= RTT + 4 \cdot D
\end{aligned}
$$

Wireless TCP

Problem: TCP assumes that IP is running across wires. When packets are lost, TCP assumes this is caused by congestion and slows down. In wireless environments, packets get lost due reliability issues. In those cases, TCP should do the opposite: try harder.

Solution #1: Split TCP connections to distinguished wired/wireless IP:



Solution #2: Let the base station do at least some retransmissions, but without informing the source. Effectively, the base station makes an attempt to improve the reliability of IP as *perceived* by TCP.

Client–Server TCP

Transactional TCP: A TCP-based transport protocol aimed to support client–server interaction