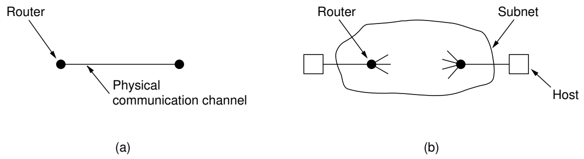
Transport Protocol

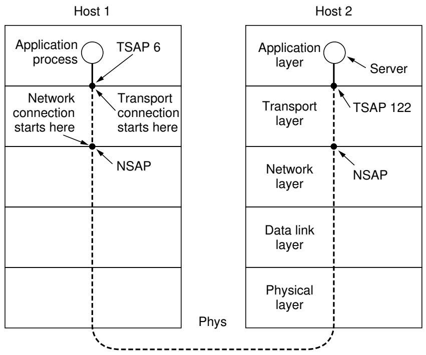
Observation: transport protocols strongly resemble those in the data link layer: e.g. lots of error and flow control. Big differences when it comes to solutions!



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

Addressing

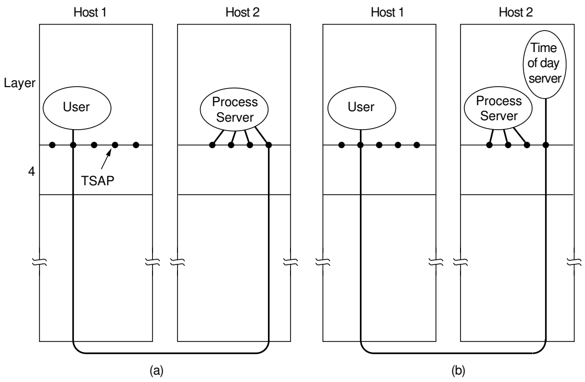
Note: Each layer has its own way of dealing with addresses. In IP, a **transport service access point** is generally an IP address with a port number.



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).

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: In order 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

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.

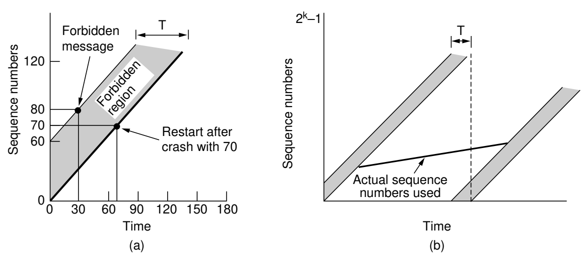
Basic idea: Assign sequence numbers to TPDUs, and let the sequence number space be so large that no two outstanding TPDUs can have the same number.

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 II

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 slower 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."

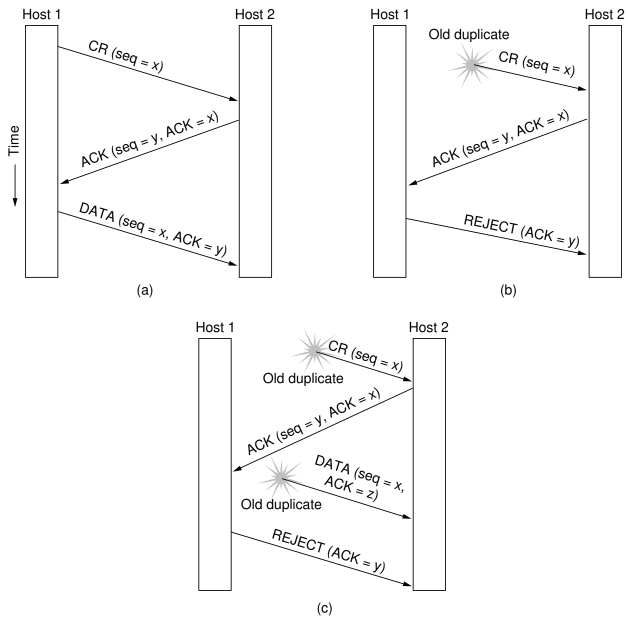
Error-free Connection Establishment

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.

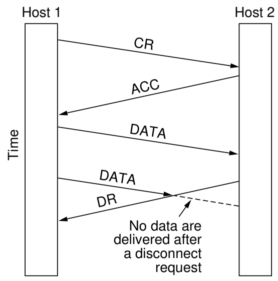
Error-free Connection Establishment II

Solution: Three-way handshake.



Error-free Connection Release

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



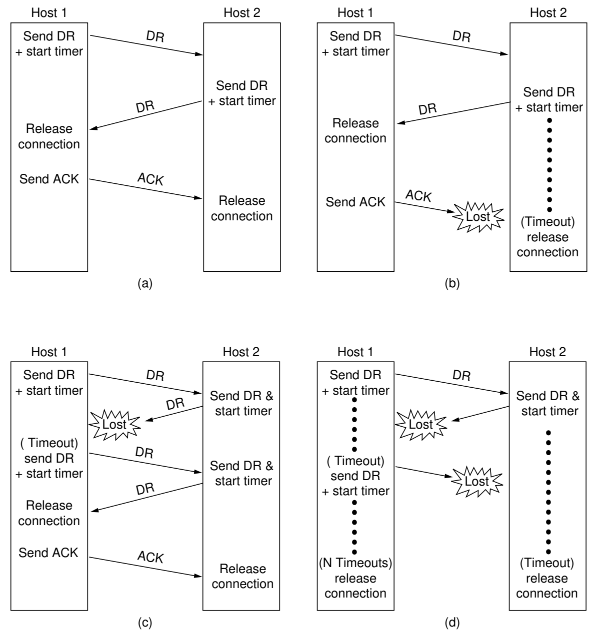
Symmetric Connection Release

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.

Symmetric Connection Release II

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



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 ⇒ 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 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.

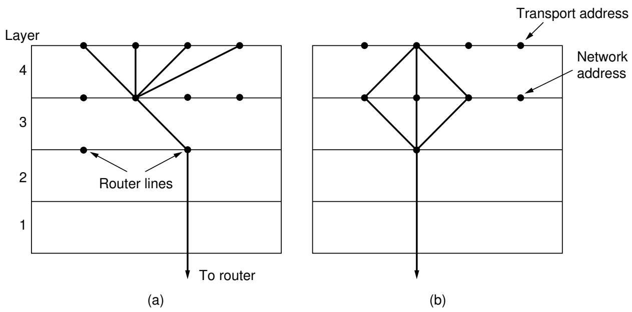
Issue: If the network can handle c TPDUs per second, and takes a total of r seconds to transmit, propagate, queue and process the TPDU, and to send an ACK, the sender need only maintain $c \cdot r$ buffers. More buffers is overkill of the network.

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

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?

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			First write, then ACK		
	AC(W)	AWC	C(AW)	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

		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