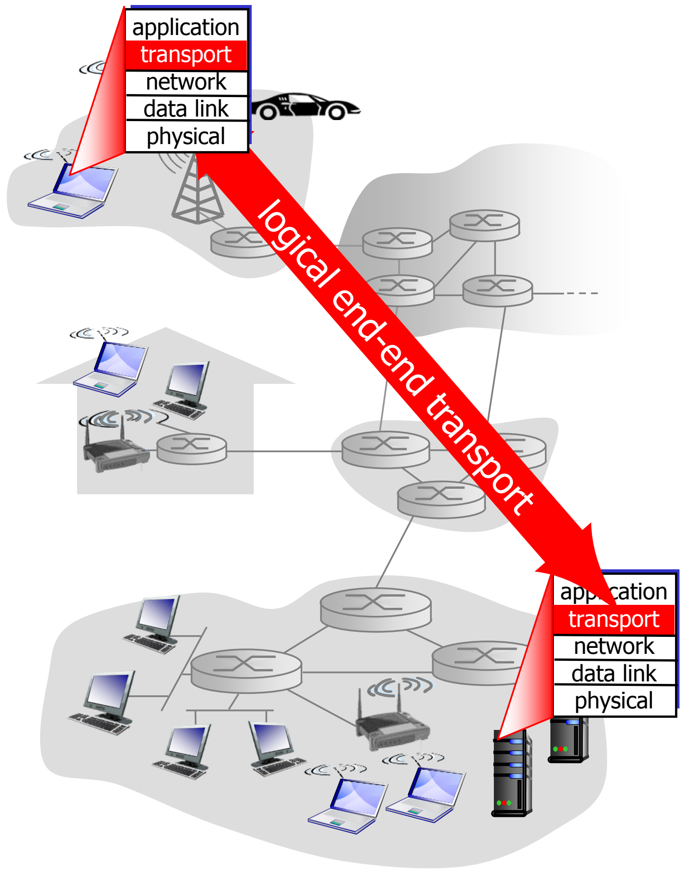
Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

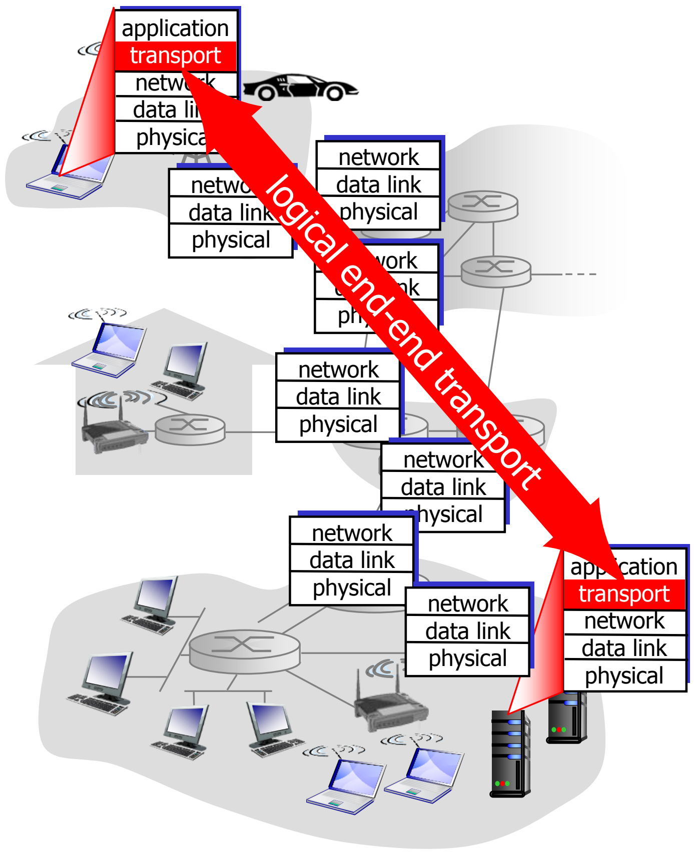
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

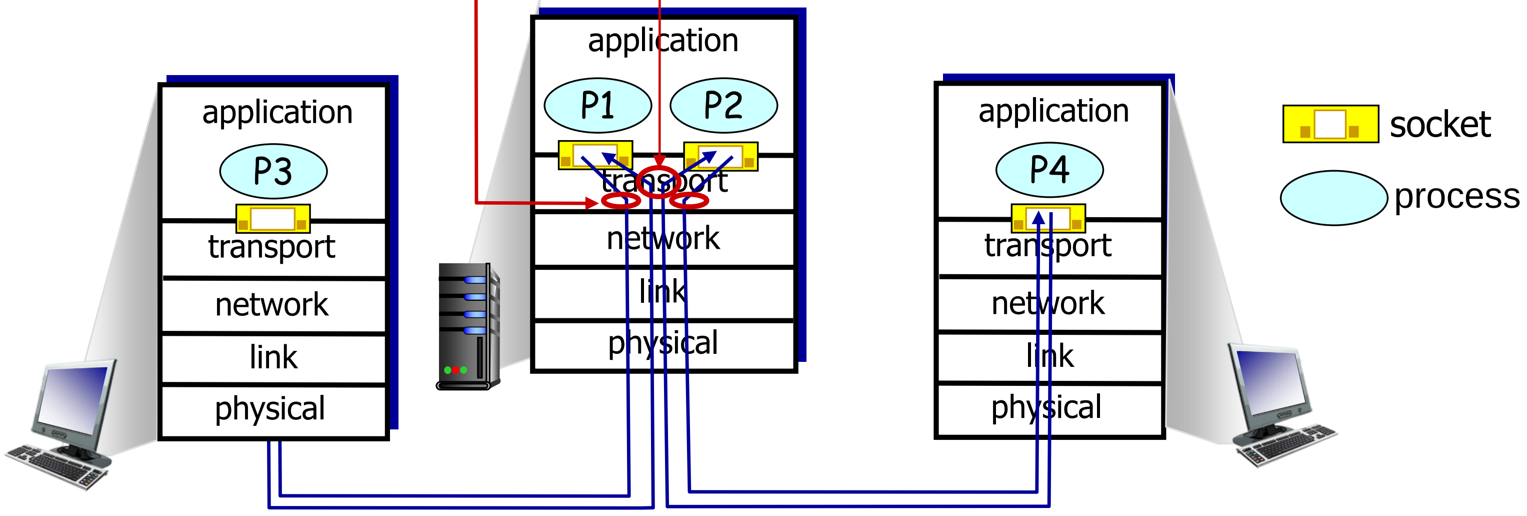
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing/demultiplexing

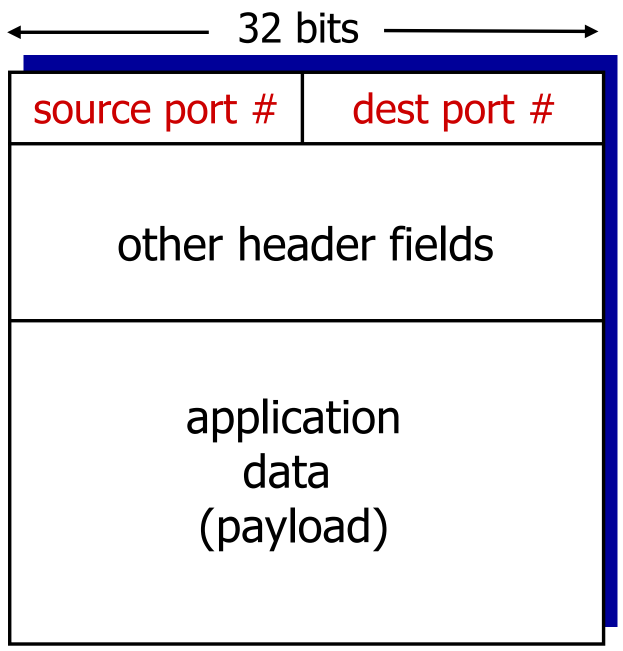
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

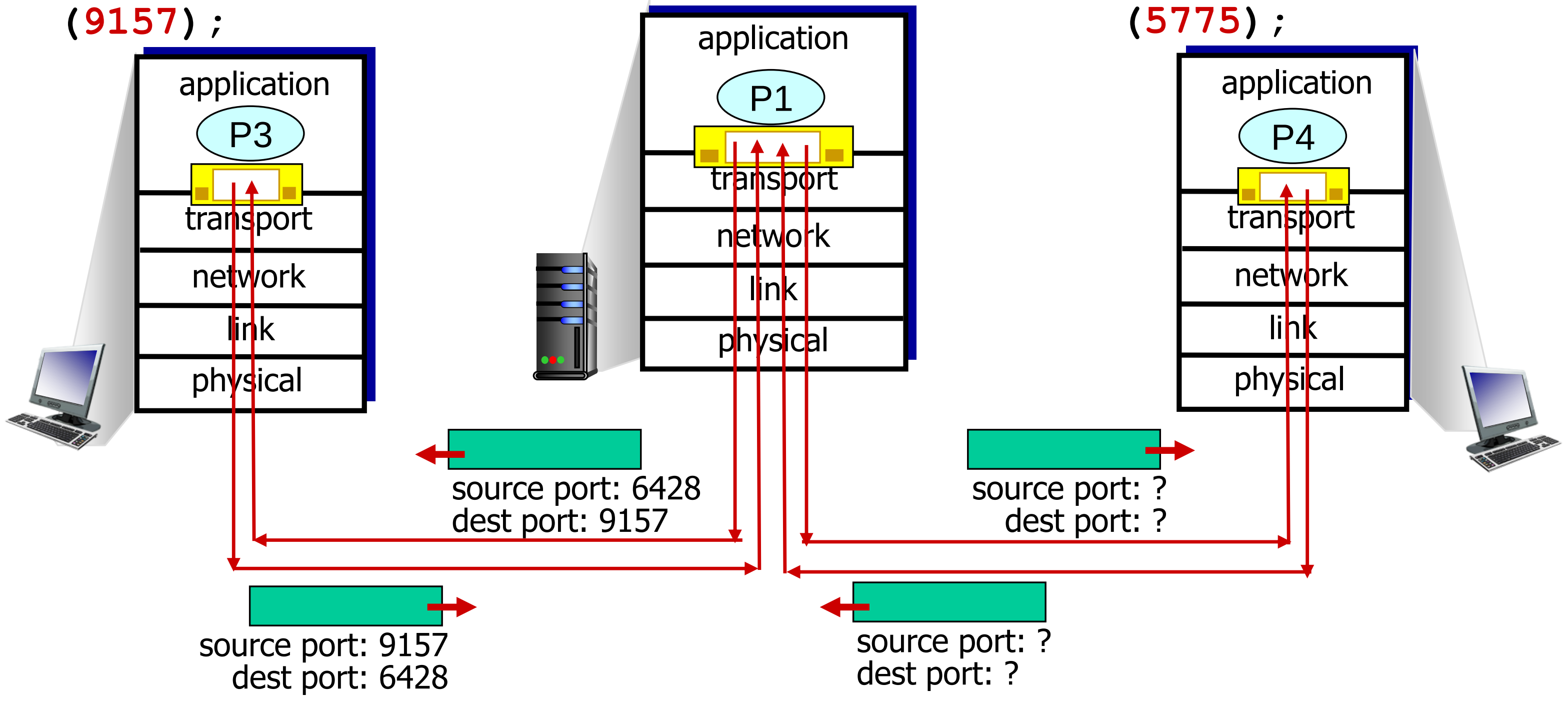
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

```
DatagramSocket
 serverSocket = new
 DatagramSocket
 (6428);
```

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```

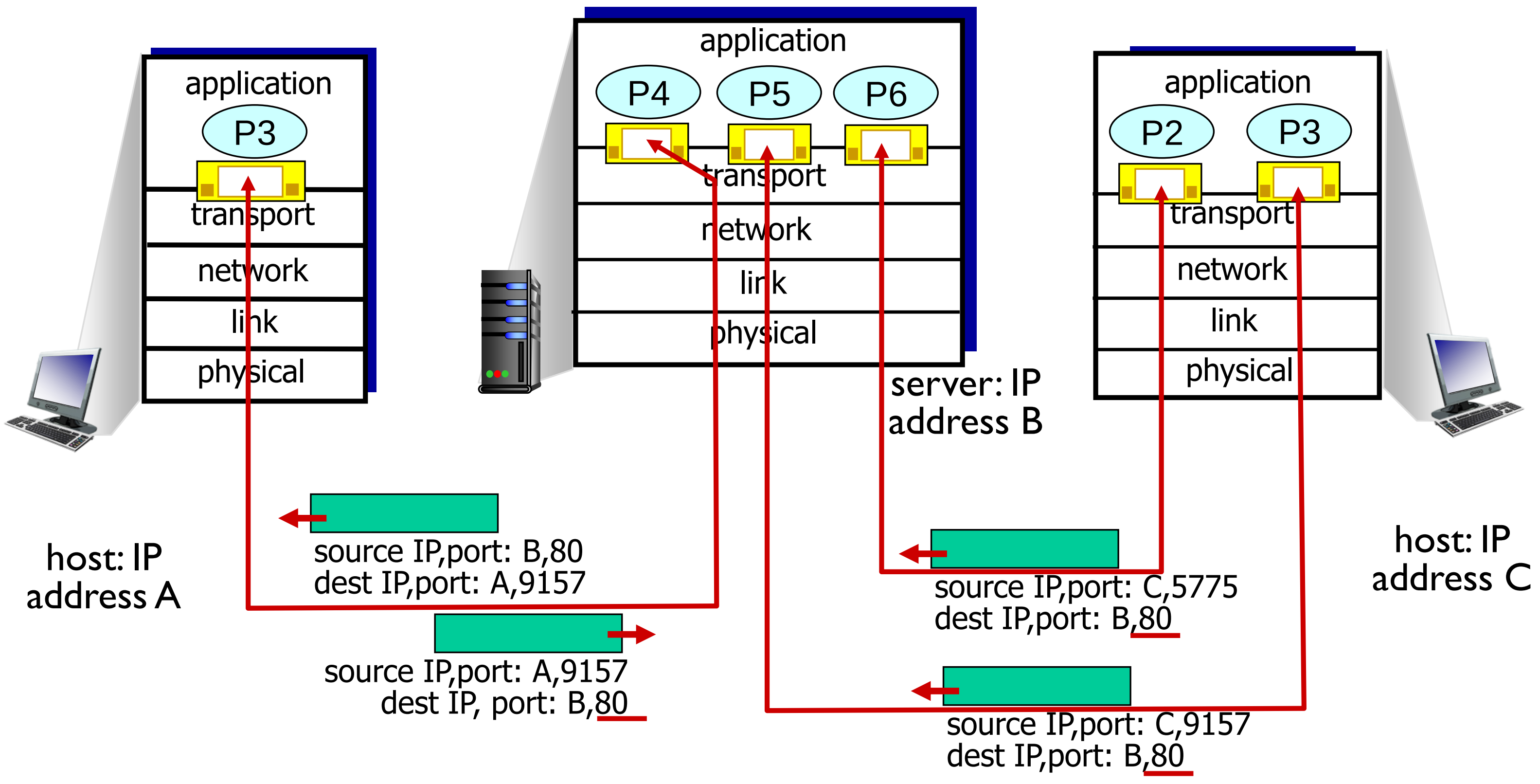


Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

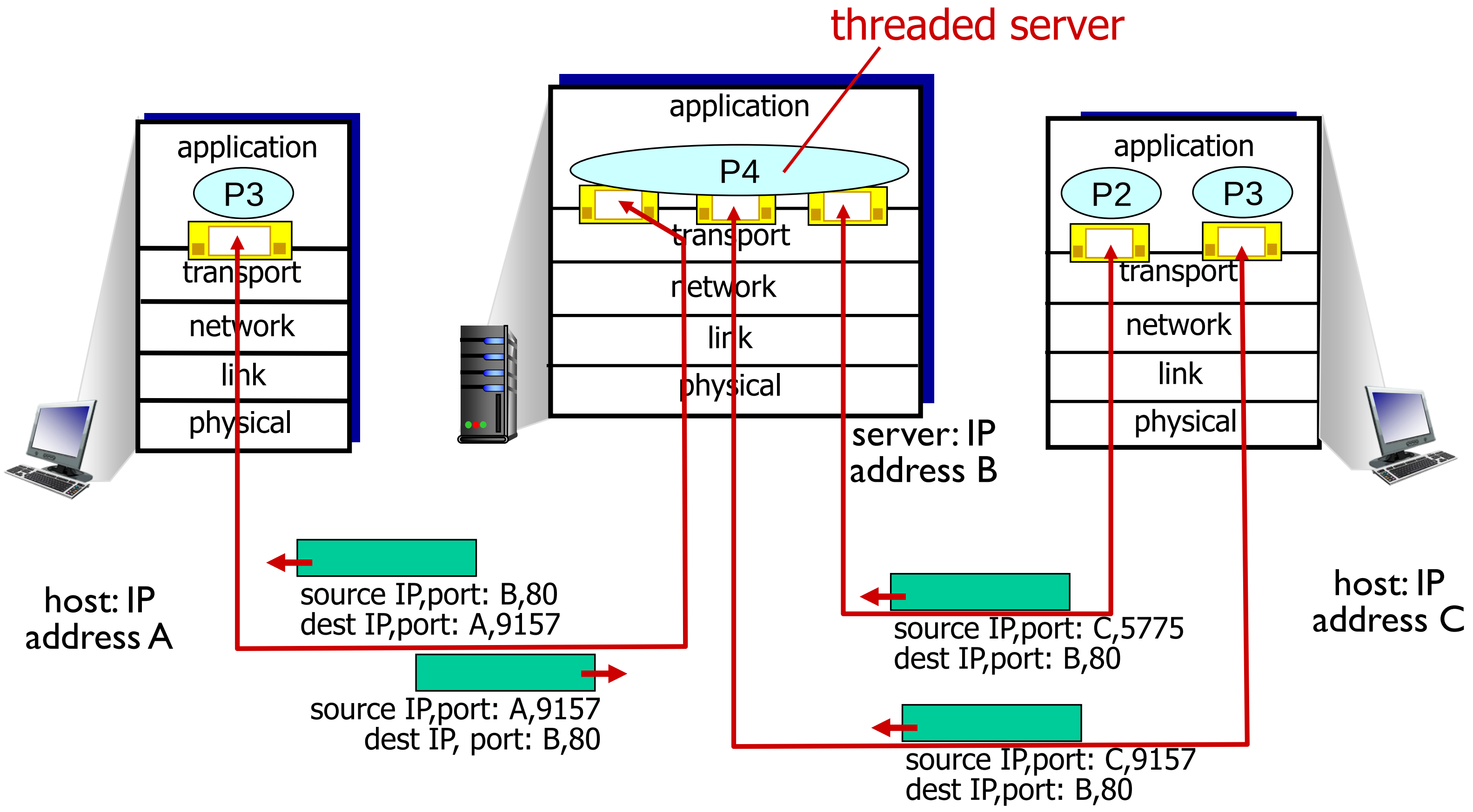
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



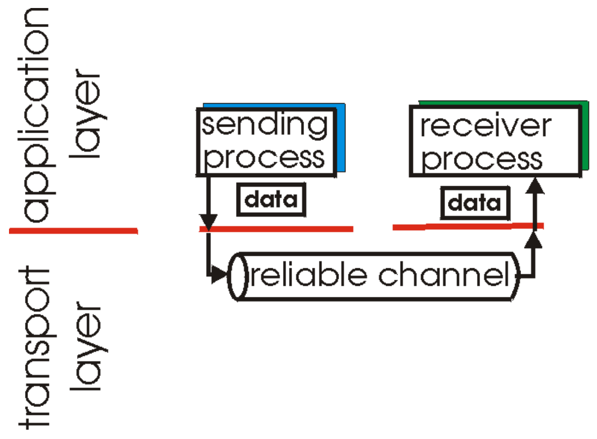
three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example



Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

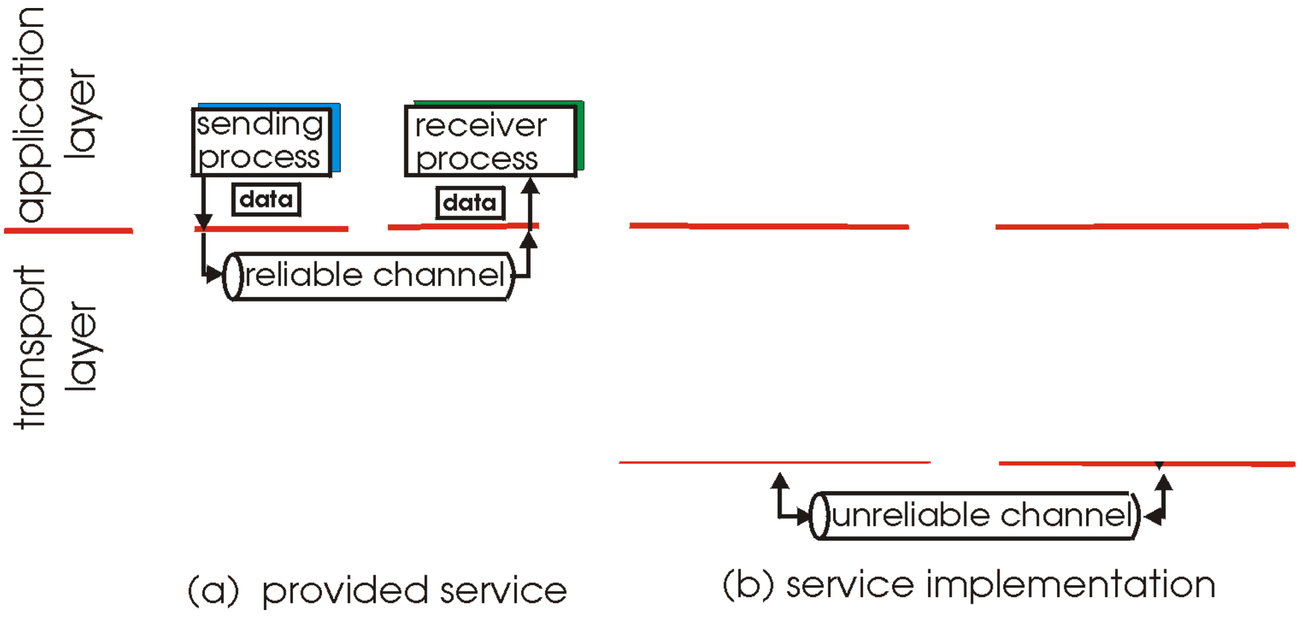


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

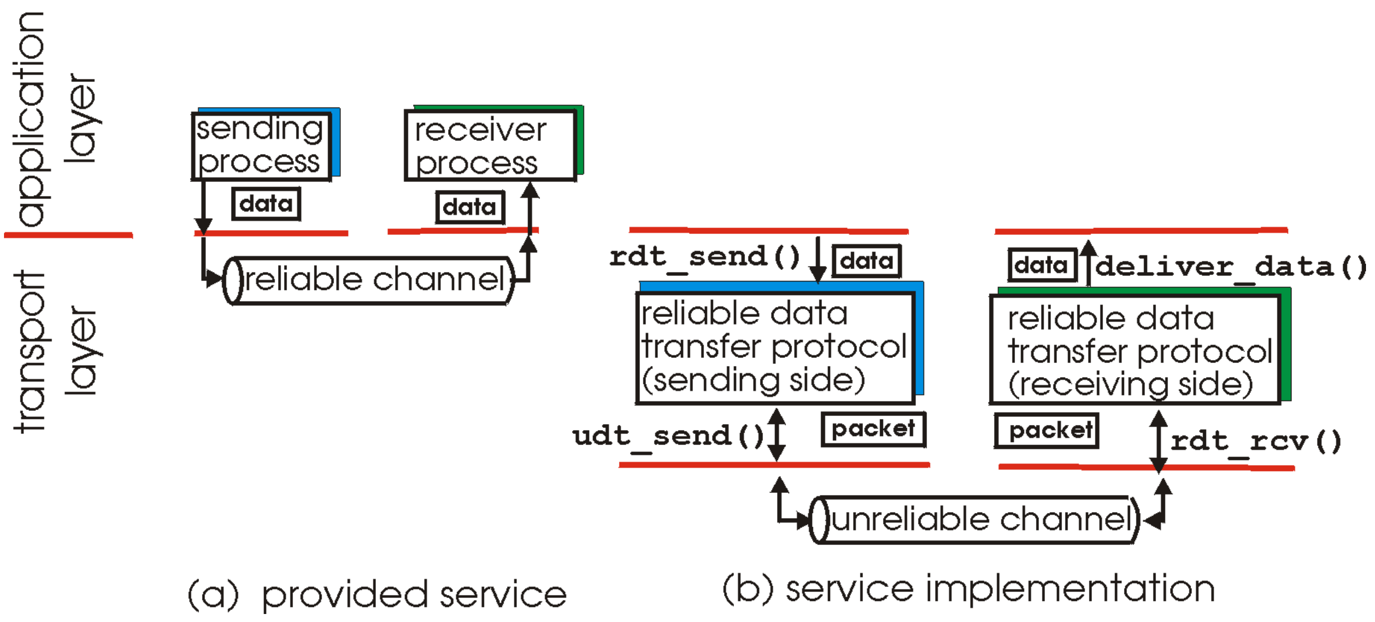
- important in application, transport, link layers
  - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

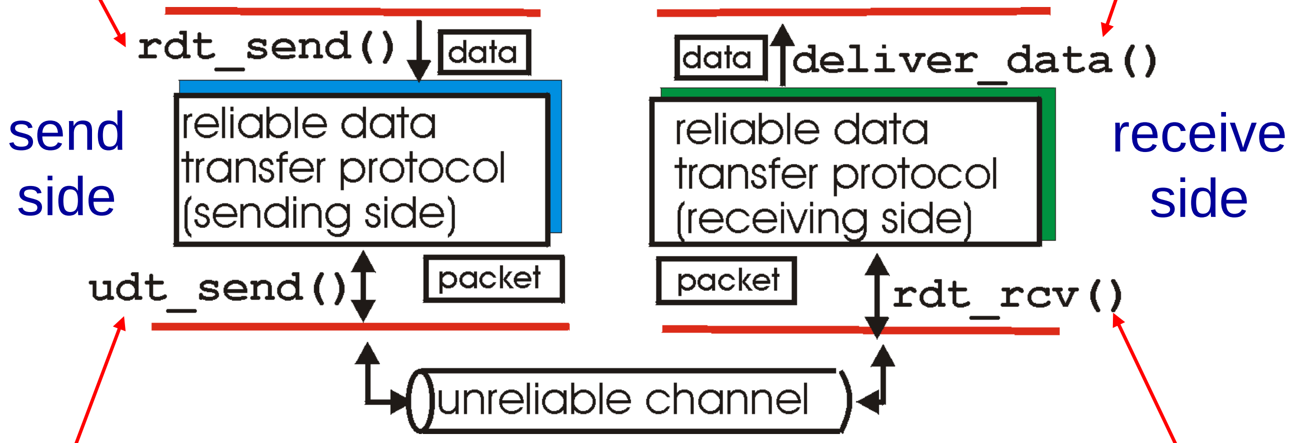


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel slides.pdf We store the state in a circle and we connect it with a line that describes the event trigger and action taken to get to the next state.

slides.pdf We want to assume that we have no errors or loss packets (this is totally not true, esspecially in a5). Then you just have to build a packet and call send. When receiving we just have to accept and depackage.

slides.pdf What if we had errors? We need to assume that the receiver detects errors (don't try to correct errors, just notice them).

slides.pdf Positive feedback is as acknowledgment sent as an ACK, basically responding that shit went through ok. A negative feedback, NACK, should trigger a resend.

slides.pdf rdt_send = Make a packet, send using unreliable send, wait on rdt_receive, check for negative ACK, is so resend, else got back to first state.

Important to note that the send is blocked until a receive happens

rdt_receive = accept packet, check for errors (checksum), if so send NACK, else extract it and send ACK.

This procedure seems simple but there is a problem, what happens if sending an ACK fails?

slides.pdf

slides.pdf We know that we've received a ACK or NACK, but that it is corrupted. We can't just ask the application to resend because we might be sending duplicate data which is unacceptable. So we incorporate sequence number. We resend the packet including this sequence number. The client now has a whole bunch of protocols around handling that number. The simplest one is called stop and wait. The sender sends a packet and waits until a good acknowledgement comes through.

slides.pdf Create a packet with the sequence 0, wait for ACK. If we receive a corrupted one or a NACK we resend the packet with the same sequence, else we increment our sequence and continue forward.

slides.pdf When we receive a packet we need to check both the checksum and the sequence number before extracting it. Then we increment our sequence number. If we receive a perfectly good packet with the wrong sequence number then we have to assume that the other side received a packet that we didn't send which implies a packet loss which we are assuming didn't happen.

slides.pdf You can get infinite loops if the channel keeps corrupting your data. TCP has a hack around this, but its hackey. Its teardown process is fairly complicated.

slides.pdf We can make do with only one time of acknowledgement by making the sequence number in the ACK wrong.

slides.pdf Sending ACKs with wrong number can get a little finicky due to duplicate data being sent all over the place (for now we pretend that everything is blocking so you don't get concurency issues). The ways that TCP handles the sequence number is different from what these slides use. TCP has the sequence number say what number we expect to receive next and these slides say which number we just got.

slides.pdf Now we incorperate loss into our system. This is where timeouts are included.

slides.pdf The state for waiting for the ACK now has a timeout value that it checks against and will just resend if the timeout is reached. There might be duplicate packets, but we will have sequence numbers to handle it.

slides.pdf

slides.pdf Fuck I missed this.

THERE WILL BE A QUESTION ON THE ASSINGMENT ABOUT THE BUG IN THESE SLIDES BASED ON THE STATEMACHINE

slides2.pdf

slides2.pdf The processing delay is how much you get bogged down processing stuff. An example of this is looking shit up in the routing table. Really got devices use a TCAM to speed up this look up by expanding the prefix and store in in a CAM (content addressable memory). CAM works by letting you present content and use that too look up something else instead of requiring you to know the address of the thing you're looking for. This memory is super expensive so lots of the cost of a network box goes into getting more TCAM. You can overflow your TCAM if you have large prefixes.

Propogation delay is delay introduced by the physical distance you are covering. We want to assume that shit is moving at the speed of light (it most definitly is not, but we assume so).

slides2.pdf

slides2.pdf In this analogy we pretend that the processing delay is the toll booth and the speed of the car is the propogation delay. We want to know how long it takes all of the cars to reuinte at the toll booth. Basically the point of this is to show that the propagation delay is dominant.