Transport Layer

- Transport Layer provides for logical communication b/w application processes running on different hosts
- Logical communication: As if hosts (running processes) were directly connected // direct, end-to-end connection

→ Provides communication services directly to the application processes

- Features:
- Implemented in end systems
- On sending side, it converts application-layer messages to into transport layer packets called **segments**

→ Done by breaking messages into smaller chunks and adding a transport-layer header to each chunk to create a segment

- segment passed to network layer where it is encapsulated to form a **datagram** (network-layer packet) and sent to destination

Note:
- Routers only act on network layer fields i.e: they do not examine transport-layer segment fields encapsulated with the datagram
- On receiving end, router extract the segment and passes it to transport layer

- Transport processes segment and passes the message to application layer
- Transport layer provides logical communication b/w **processes**
- Network layer provides logical communication b/w **hosts**
- Certain services can be provided by transport layer if the underlying layer doesn't provide it

Q- 2 houses, very distant from each other. In respective houses Bill, Anne responsible for delivering and collecting mails from house members. Postal service responsible for delivering mails b/w houses

- Message = letters
- processes = house members
- Transport layer protocol = Anne, Bill
- Network layer protocol = Postal service

- IP service model is a best-effort delivery service, makes best effort to deliver packets but makes no guarantee orderly delivery of segments

→ Thus, unreliable service

- Hosts have at least one IP address

- Protocols at transport layer:
- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol)

- Common features of UDP and TCP
- Both protocols extend host-to-host delivery to process-to-process delivery and is called **transport layer multiplexing** and **de-multiplexing**

- Provide integrity checking by including error-detection fields in their headers

- TCP
- Provides reliable data transfer - flow control, seq numbers, congestion control, acks, timers

→ Ensures **correct** and **in-order** delivery to the receiving process.

→ congestion control prevents TCP connections from swamping the links and routers b/w communicating hosts with an excessive amount of traffic - done by giving the connection an equal share of the link bandwidth, by regulating rate of sending side at which they can send traffic into network

- TCP converts IP's unreliability by providing features that IP doesn't i.e: gurranteed correct order, no corruption, acks ...

- UDP
- Unreliable
- UDP traffic is unregulated // can send data at any rate

Multiplexing

- A process can have one or more sockets

- De-multiplexing:
- At the receiving end, transport layer examines the segment headers // fields to identify the receiving socket, and directs the segment to that socket

collect
- Multiplexing:
- Gathering of data chunks at the source host from different sockets, encapsulating each of them with header information to create segments, and passing the segments to network layer

- Data chunks are stream of bytes that are received from sockets. After attaching header info, the segments are divided into packets and sent to network layer

- sockets have unique identifiers → Port Numbers
- Each segment has special fields

- source Port and destination Port numbers = special Fields
- Each port num is 16-bit. 80
- Ports 0—1023 are well-known ports // reserved for protocols like HTTP .....

Connection - Connectionless Multiplexing and demultiplexing

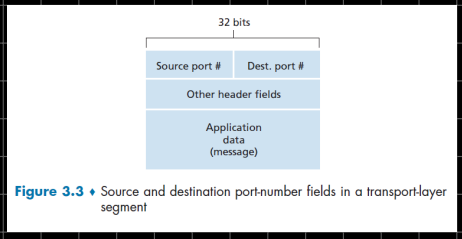


Figure 3.3 ♦ Source and destination port-number fields in a transport-layer segment

- TCP socket as a 4-tuple model
→ (source IP, source port, destination IP, destination port)

- TCP: Ack for every message sent

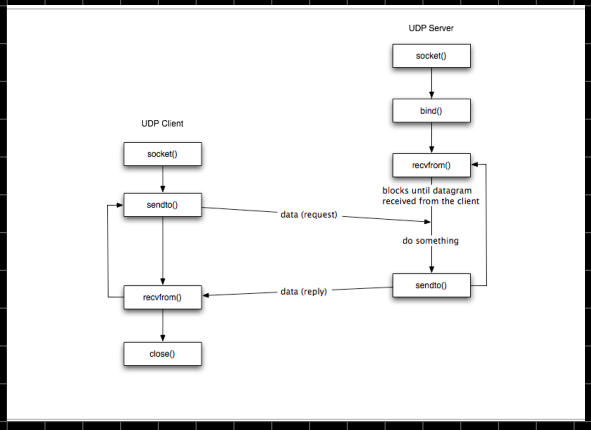
- TCP connection-oriented
- UDP connection-less

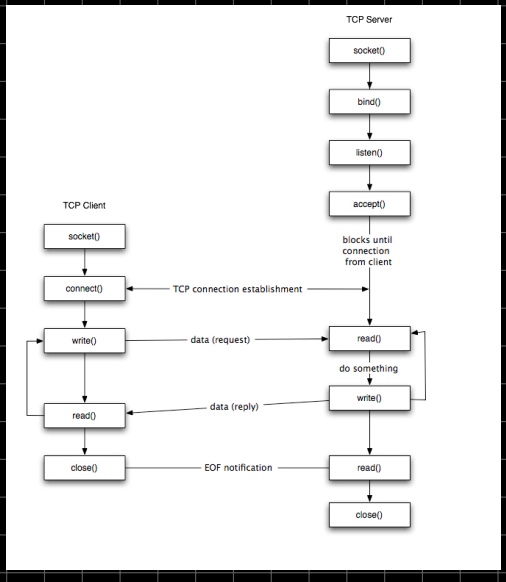
- connection establishment request sent to server, containing connection establishment bit set in TCP header, source port and destination port number → socket (AF_INET, SOCK_STREAM)
·connect (server, dest port)

- when the host OS running the server process receives the connection request, it locates the server process waiting to accept the connection on the port number and creates a new socket accept()

- The new socket created is identified by the 4-tuples in the connection request source IP, source port, destination IP, destination port)
- server host may support simultaneous TCP sockets

- UDP uses dest port only for demux
- TCP uses all 4 tuples for demux





- Socket workflows

UDP

- Connection less because no handshaking b/w client and receiver e.g: DNS uses UDP
- Host waits for reply after sending, if no reply then resend with some changes e.g: send to another name server (not compulsory)
- Why UDP is better ?

- Finer application-level control over what data is sent and when
→ Suited for realtime applications requiring minimum sending rate, do not want overly delay segment transmission, and can tolerate some data loss.
→ These additional services can be implemented as part of the application if required

- No connection establishment
→ Eliminates delay for connection establishment (DNS)

- No connection state
→ Allows to run more active clients
→ does not store receive, send buffers, congestion control and sequence, acknowledgment params

- Small packet overhead
→ UDP segment has 8 bytes of overhead
→ TCP segment has 20 bytes of overhead

x ——— x

- Many applications switch to UDP like network management, DNS, new versions of HTTP, some streaming as well

segment structure UDP

- length field specifies number of bytes in the segment (header + data)

- UDP header has only 4 fields each of 2 bytes // 16 bit each
- Checksum for error detection

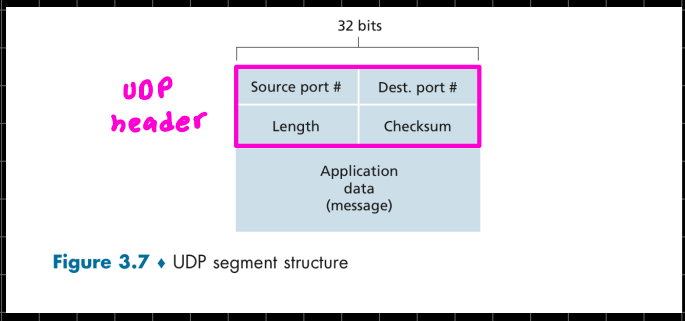


Figure 3.7 ♦ UDP segment structure

- Checksum:
sender side ↓
- Source port and destination port is summed up
- If any overflow, then it is added to the LSB side (wrap around)
- 1's complement applied to the sum and added to checksum field

* different in slides

Q. 0110011001100000
0101010101010101
1000111100001100

overflow

0000000 0000
0110011001100000
0101010101010101
1000111100001100
0100101011000001
1
0100101011000010

0100101011000010

- Final string = 1011 0101 0011 1101

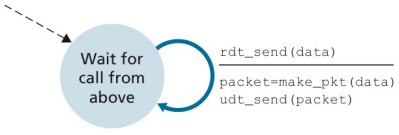
- At receiver end, all four 16 bit words are added, and if no errors are introduced the checksum will result in 1111 1111 1111 1111.

x ——— x

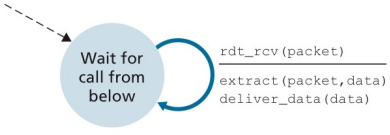
Reliable Data Transfer

- Responsibility of delivering data in-order, without any corruption and data losses
- Network layer is an unreliable channel, so implemented in transport layer
- RDT is a service abstraction that is made real by TCP

RDT 1.0



a. rdt1.0: sending side



b. rdt1.0: receiving side

Figure 3.9 ♦ `rdt1.0`—A protocol for a completely reliable channel

- rdt= reliable data transfer
- udt= unreliable data transfer

- Everything discussed is uni-directional

- Assuming the underlying channel (network layer) is completely reliable

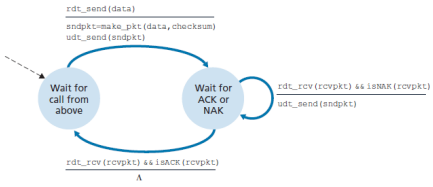
- **Sending side:**
- rdt_send(data) event, creates a packet containing data using make_pkt(data), and sends it to the channel using udt_send()
- rdt_send() invoked by upper-layer application

- **Receiving side:**
- rdt_recv(packet) event, removes data from packet using extract(packet,data) and passes data to upper layer via deliver_data(data)

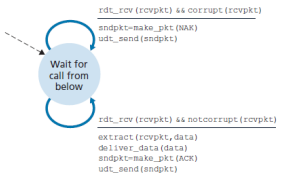
- rdt_recv() invoked from underlying layer

- No need for receiver to ask sender to slow down or send any feedback

RDT 2.0



a. rdt2.0: sending side



b. rdt2.0: receiving side

Figure 3.10 ♦ rdt2.0—A protocol for a channel with bit errors

- Assuming packets sent are received (can have errors), and not lost
- uses positive acknowledgments (Ack) and negative acknowledgments (NAk)
- These control messages allow sender to know, that data has been received correctly/incorrectly
- Retransmission protocols known as automatic repeat request protocols (ARQ)

- ARQ features to handle bit errors:

① Error Detection techniques e.g: checksum
② Receiver feedback using Acks and NAks
③ Retransmission of packet that is received in error

- Sending side:

- sndpkt contains data to be sent along with checksum
- Ack or NAk is waited for
- If a NAk is received, the protocol retransmits the last packet and waits for an Ack
- If Ack is received, the protocol moves to the state of waiting for the packet from the upper layer

- Can't get more data from upper layer until Ack is received, hence rdt 2.0 is a stop-and-wait protocol

- Receiving side:
→ simple workflow see diagram

- Flaw:
- Ack / NAk packet can be corrupted
- Can not retransmit b/c it would introduce duplicate packets
- Adding additional checksum bits to make a mathematically strong checksum value, only applicable for channels that can corrupt packets but not lose them

→ solution is to add a sequence number of the packets. Receiver uses sequence number field to identify retransmission. sequence number moves forward in a modullo 2 arithmetic fashion.

- Due to assumption, Acks / NAks do not need to identify packets they are acknowledging, it is obviously of the most recent packet sent

RDT 2.1

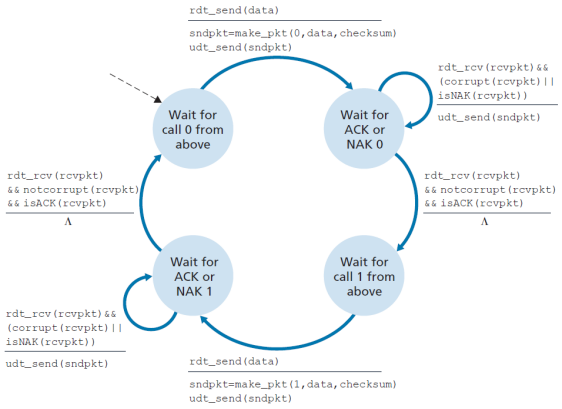


Figure 3.11 ♦ `rdt2.1` sender

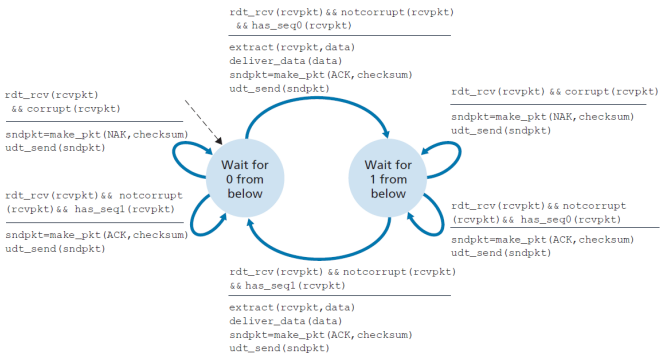


Figure 3.12 ♦ `rdt2.1` receiver

- When an out-of-order packet received, ACk is sent.
- When corrupted packet is received, NAk is sent
- A sender that receives two acks for the same packet (duplicate ACKs) means receiver did not correctly receive the packet following the packet that is ACked twice
- Receiver now includes sequence number of the packet being acknowledged by adding argument in make_pkt by ACK 0, ACK 1.
- sender examines the sequence number of packet being acknowledged through isACk

RDT 2.2

- Many link-layer protocols provide error checking capability, so why does UDP provide checksum?

→ No gurrantee that all links b/w source and destination provide error checking, some might be using protocols that do not provide error checking

→ Bit errors can be introduced in router's memory

- Link by link reliability, and in-memory error detection not gurranteed
- UDP does not do anything to recover from an error. Some implementations of UDP either discard or pass the damaged segment to application layer

- Length is of 16 bits size, so theoratical max size of UDP packet = $2^{16} - 1 = 65535$
- Most physical layers have MTU (Max Transmission unit) to dictate how much data can be transmitted in 1 packet
- If UDP exceeds this amount, the segment will be fragmented according to MTU.
- Fragmentation increases likelihood of packet loss, as a single fragment loss results in the loss of entire original datagram

→ Hence, UDP packets are limited to very small sizes to avoid fragmentation

- Path MTU: smallest MTU along the entire path b/w 2 IP hosts
- Typical UDP size = 1500 - 20 = 1480 ; 20 bytes of UDP header
- For IPv4: 65535 bytes for entire IPv4 packet - 16 bit
- Data that can be fit into a UDP datagram sent using IPv4 = 65535 - 20 - 8
  (20: IP header, 8: UDP header)
  = 65507
- For IPv6: 65535 bytes - IPv6 payload limit - 16 bit
- Data = 65535 - 8 = 65527

- IPv4 header size variable, but min size = 20 bytes
- IPv6 header = 40 bytes

- Checksum:
- optional for IPv4, mandatory in IPv6
- Use 0x0000 when need to include checksum field in the checksum algo.
- In IPv4 → 0x0000, UDP checksum not in use
- If checksum value comes out to 0x0000, change it to 0xFFFF
- If checksum is in use, and on receiver's side if checksum not verified, packet is usually thrown away
- Error detection codes: Reed-Soloman codes, Hamming codes
- Demux key at IP layer

→ RDT