Transport Services and Protocols

Transport Services and Protocols

- The **transport layer** is responsible for providing **logical communication** between **application procceses** running on **different hosts**.
- There are two actions the transport layer needs to achieve:
  1. **Send action**: The transport layer needs to **divide application messages into segments**, and pass them to the **network layer**.
  2. **Recieve action**: The transport layer needs to **reassemble segments** from the **network layer** into **messages**, and pass them to the **application layer**.
- There are **two transport protocols** available: the **User Datagram Protocol (UDP)**, and the **Transmission Control Protocol (TCP)**.

The Application Layer, The Transport Layer, and The Network Layer

- The **application layer** is the **process** that is running on the **network host**.
- The **transport layer** is responsible for the **logical communcation** between **processes**.
- The **network layer** is responseible for the **logical communication** between **network hosts**.
- When a **message is sent** from an application over the network, it follow the following order: application layer → transport layer → network layer.
- When a **message is received** from the network, and sent to an application, it follows the following order: network layer → transport layer → application layer.

Transport Layer Actions

- When a message is **sent from an application**, the **transport layer** does the following:
  1. Recieves the application-layer message.
  2. Determines segment header field values.
  3. Creates the segment.
  4. Passes the segment to the internet protocol.
- When a message is **recieved from the network**, the **transport layer** does the following:
  1. Recieves the segment from the internet protocol.
  2. Checks the header values.
  3. Extracts the application-layer message.
  4. Demultiplexes message up to the application via a socket.

User Datagram Protocol and Transmission Control Protocol

- The **User Datagram Protocol (UDP)** is an **unreliable**, **unordered** delivery protocol. This protocol has **minimal overhead** but is **not reliable**.
- The **Transmission Control Protocol (TCP)** is a **reliable**, **in-order delivery** protocol that supports **congestion control**, **flow control**, and **connection setup**. This protocol has **significant overhead**, but is **very reliable**.
- Both **UDP** and **TCP** do not provide **delay guarantees** and **bandwith guarantees**.

Multiplexing

Multiplexing

- **Multiplexing** is a method used by **networks** to **consolidate multiple signals** into a **single composit signal** that is then **transported over a common medium**.
- When **sending data**, **multiplexing** is used to **transmit segments** from **several sockets** via **transport headers**.
- When **recieving data**, the **header info** is used to **demultiplex** the data, sending it to the **correct socket**.
- **Demultiplexing** works by receiving **IP datagrams** containing a **source and destination IP address and port as headers**. Each datagram contains a **single segment**.
  - The IP addresses and port numbers are used to route the segment to the correct socket.
  - This is why you must specify a port (and sometimes an address) number when creating a socket.

Connectionless Multiplexing

- When creating a **server socket** you must specify a **port**, that the **socket will be bound to**.
- When creating a **datagram (UDP)** you must specify the **destination host and port**.
- When a **host** receives a **datagram** with the **port destination** that matches the **port the server socket is bound to**, it will **route the datagram** to **said socket**.
  - The host address does not matter when a datagram is recieved by a host, it will simply attempt to route it to the destination port. This means that several hosts can send datagrams to the same port, and they will all be available at the same socket.

Connection-Oriented Multiplexing

- When creating a **connection oriented socket (TCP)**, you create a **4-tuple** containing the **source address, source port, destination address, and destination port**.
- The **4-tuple** is used to **route inbound packets** to the **correct socket**.
- **Servers** may support **several simultaneous TCP sockets**, typically one for **each client**. Each socket will have a **unique 4-tuple**.

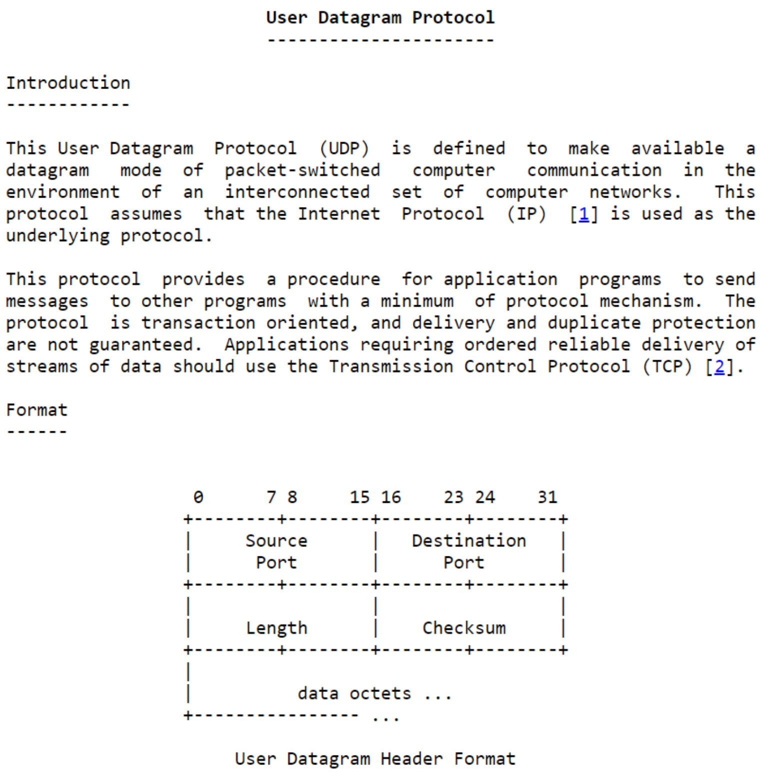
Summary

- Multiplexing and demultiplexing are based on segman, datagram header values.
- UDP uses the port for demultiplexing.
- TCP uses the 4-tuple for demultiplexing.
- Multiplexing happens at all layers.

User Datagram Protocol

User Datagram Protocol

- The **User Datagram Protocol (UDP)** is a **connectionless (no handshake) communication protocol** that uses the **internet protcol**.
- **UDP** has **minimal overhead**, at the cost of having **no connection state, no congestion control, and no packet delivery verification**.
- **UDP** is used for **loss-tolerant, rate sensitive** applications.
- It is possible to have **reliable UDP communication** by implementing it at the **application layer**.



```
                         User Datagram Protocol
                         ----------------------

Introduction
------------

This User Datagram  Protocol  (UDP)  is  defined  to  make  available  a
datagram   mode  of  packet-switched   computer   communication  in  the
environment  of  an  interconnected  set  of  computer  networks.   This
protocol  assumes  that the Internet  Protocol  (IP)  [1] is used as the
underlying protocol.

This protocol  provides  a procedure  for application  programs  to send
messages  to other programs  with a minimum  of protocol mechanism.  The
protocol  is transaction oriented, and delivery and duplicate protection
are not guaranteed.  Applications requiring ordered reliable delivery of
streams of data should use the Transmission Control Protocol (TCP) [2].

Format
------


                  0      7 8     15 16    23 24    31
                 +--------+--------+--------+--------+
                 |     Source      |   Destination   |
                 |      Port       |      Port       |
                 +--------+--------+--------+--------+
                 |                 |                 |
                 |     Length      |    Checksum     |
                 +--------+--------+--------+--------+
                 |
                 |          data octets ...
                 +---------------- ...

                      User Datagram Header Format
```

- **UDP datagrams** have the following headers:
  1. **Destination port** — The port the datagram is being sent to.
  2. **Length** — The length in octets of the datagram.
  3. **Checksum** — The checksum is used to ensure the packet received was not corrupted during transmission.
  4. **Pseudo** — A header containing the source address, destination address, the protocol, and the UDP length. This information is used to give protection against misrouted datagrams.
- The **checksum** is computed by taking the **one's complement sum** of the **entire content** of the **datagram**, treating the bits of the datagram as a **series of 16-bit integers**.
  - The checksum is not perfect, several different datagrams can result in the same checksum.

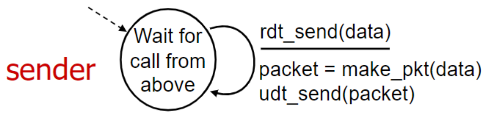
Reliable Data Transfer

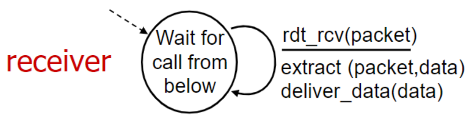
Reliable Data Transfer Overview

- The **complexity** of the **reliable data transfer protocol** will strongly **depend** on the **characteristics** of the **unreliable channel** (drop packes, corruption, data reordering, etc).
- The **sender** and **receiver do not know the state of each other**, unless it is **communicated**.

RDT 1.0 — Reliable Transfer over a Reliable Channel

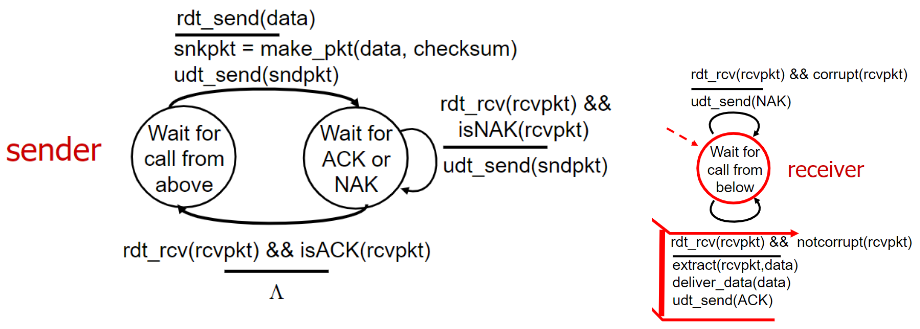
- In this case, the **underlying channel** is perfectly **reliable**; there are **no bit errors**, and **no loss of packets**.
- This is the most primitive form of reliable data transfer; the **sender sends the data**, and the **receiver reads the data**.





RDT 2.0 — Channel with Bit Errors

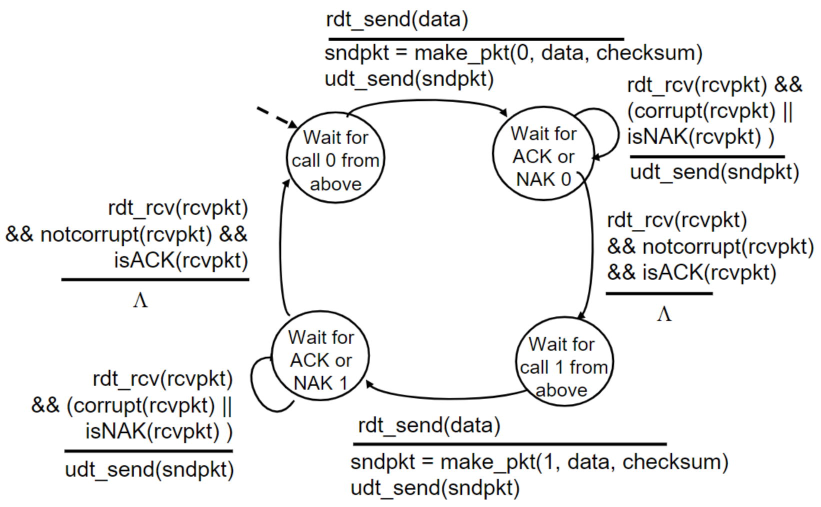
- In this case, the **underlying channel** may **flip bits in packets randomly**. To determine if this happend we can use a **checksum**.
- To **recover from errors**, we can have the **receiver send a respone**.
  - The **acknowledgement (ACK) response** indicates that the **packet was successfully transmitted**.
  - The **negative acknowledgement (NAK) response** indicates that the **packet was not successfully transmitted**.
  - If the **sender** recieves a **NAK response**, they must **retransmit the packet**.
- The **sender** must **stop and wait** for a resonse, before sending the next packet.

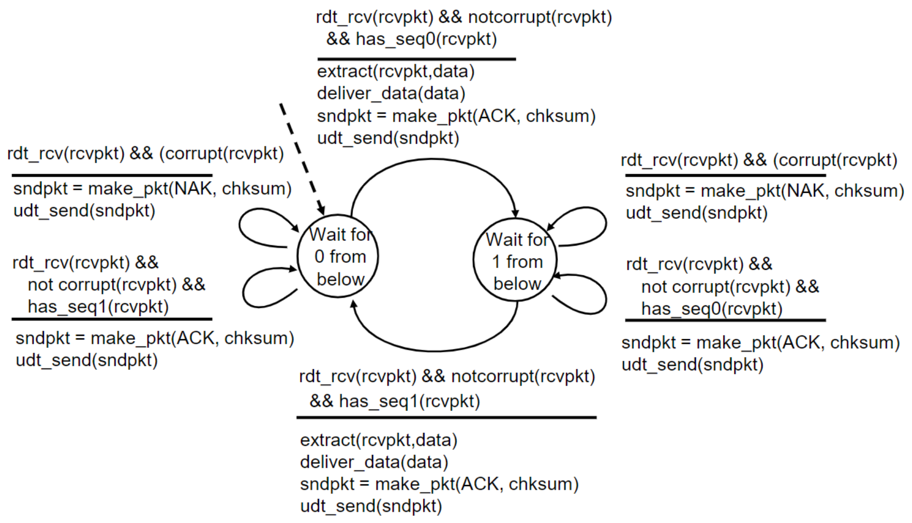


- **RDT 2.0 has a fatal flaw**, if the **ACK** or **NAK** gets **corrupted**, the **sender** won't know if the **receiver** received the packet.
  - You can't retransmit the packet, it could result in a possible duplicate.

RDT 2.1 — Handling Corrupted ACKs and NAKs

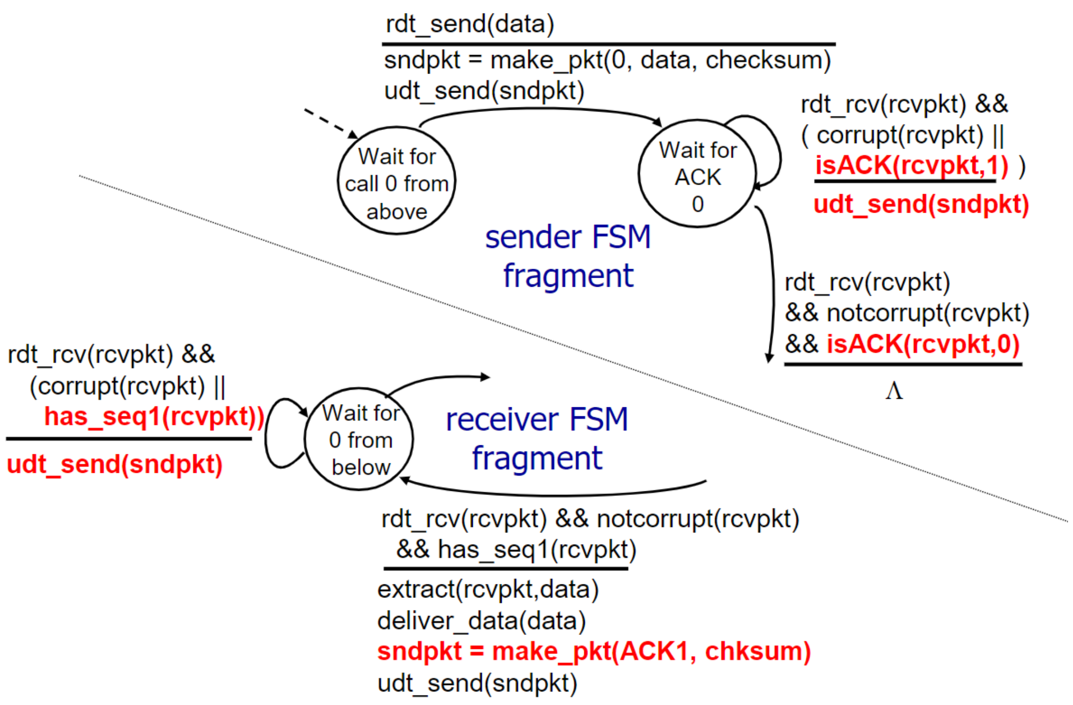
- In the case where the **receiver's response** gets **corrupted**, we can add a **sequence number to each packet**. This will allow the reciever to detect duplicate packets.


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

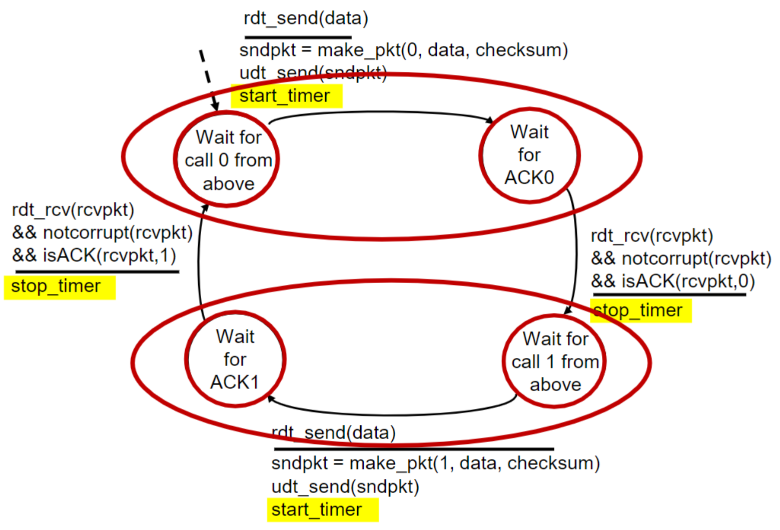
RDT 2.2 — A NAK-free Protocol

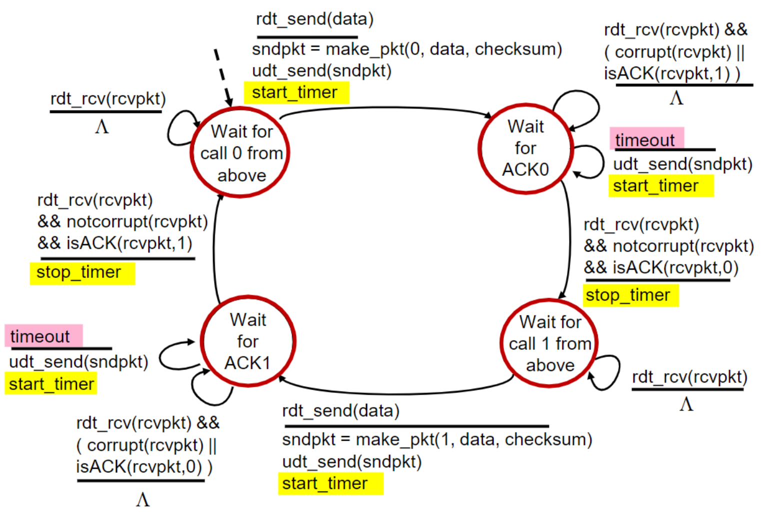
- This protocol has the **same functionality as 2.1**, but it only uses **ACKs**.
- **Instead of NAK**, the **receiver** sends **ACK** for the **last packet recieved OK**.
  - The receiver must explicitally include the sequence number of the packet being ACKed.
- A **duplicated ACK** at the **sender** results in the **same action as NAK**; retransmit the current packet.


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
udt_send(sndpkt)
sender FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
udt_send(sndpkt)
Wait for 0 from below
receiver FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

RDT 3.0 — Channels with Bit Errors and Packet Loss

- In this case the **underlying channel** can **lose packets, and corrupt packets**.
- The checksum, sequence of numbers, ACKs, and restransmission will help, but will not be enough.
- In this protocol, the **sender waits a "reasonable" anout of time** for an **ACK**, if **ACK is not recieved** within a reasonable amount of time, the **sender** will **retransmit the packet**.
  - If the packet ACK was just delayed and not lost, the sequence number will allow to reciever to discard the duplicate packet.
  - When the reciever is sending the ACK, they must specify the packet number they are ACKing.


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
Wait for call 0 from above
Wait for ACK0
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
Wait for ACK1
Wait for call 1 from above
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
Wait for ACK1
Wait for call 1 from above
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ

- The **stop-and-wait** operation the **sender** has to undergo is **very slow**; It reduces the maximum transmission rate.

RDT 3.0 — Pipelined Protocol Operations

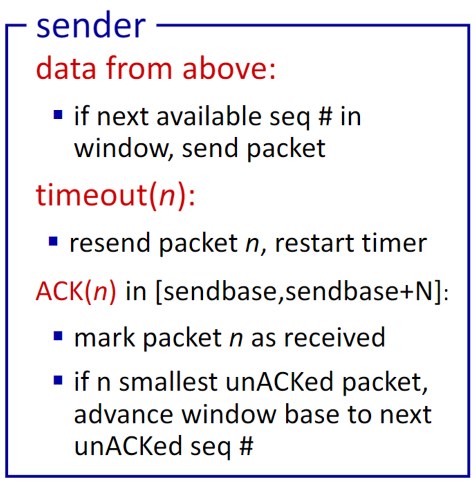
- **Pipelining** is when the **sender** allows for **multiple in-flight** (yet to be acknowledged) **packets**.
  - The range of the sequence numbers must be increased.
  - Buffering is required and the sender's end and the reciever's end.
- **Pipelining** allows for increased utilization.

RDT 3.0 — Pipelined Go-Back-N Protocol

- In the **Go-Back-N protocol**, the **sender** sends a **"window"** of up to **N**, consecutively transmitted (unacknowldeg) **packes**.
- After the sender recieves a **cumulative ACK** (acknowledgement of all packets up to n), the **sender** moves the **window forward to begin at n+1**.
- If a timeout occurs, simply retransmit the window.
- In the **Go-Back-N protocol**, the **receiver** sends an **ACK** for the **correctly-received packets** so far, with the highest **in-order** sequence numbers (This may generate duplicate ACKs).
- If the **receiver** receives **out-of-order packets**, they can either reorder them, or discard and re-ACK (it depends on the implementation).

RDT 3.0 — Pipelined Selective Repeat Protocol

- In the **Selective Repeat protocol**, the **receiver individually acknowledges** all **correctly received pacakets**.
  - The receiver buffers the packets, as needed, for an eventual in-order delivery to the uppter layer.
- The **sender retransmits unACKed packets**.
  - The sender must maintain a timer for each unACKed packet.



sender

data from above:

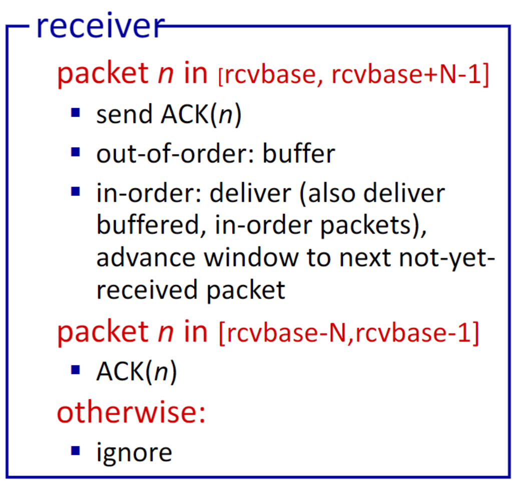
- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N]:

- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #



receiver

packet *n* in [rcvbase, rcvbase+N-1]

- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Principals of Congestion Control

- Informally, **congestion** occurs when **too may sources are sending data too fast** for a **network to handle**.
  - Indicators that congestion is occuring are **long delays, and packet loss**.
- One the indicators of congestion occur, the sender should act accordingly.
- There is also **network-assisted congestion control**; **routers** provide **direct feedback** to the **sending** and **receiving** hosts (may indicate congestion level as-well).

Connection-Oriented Transportation: TCP

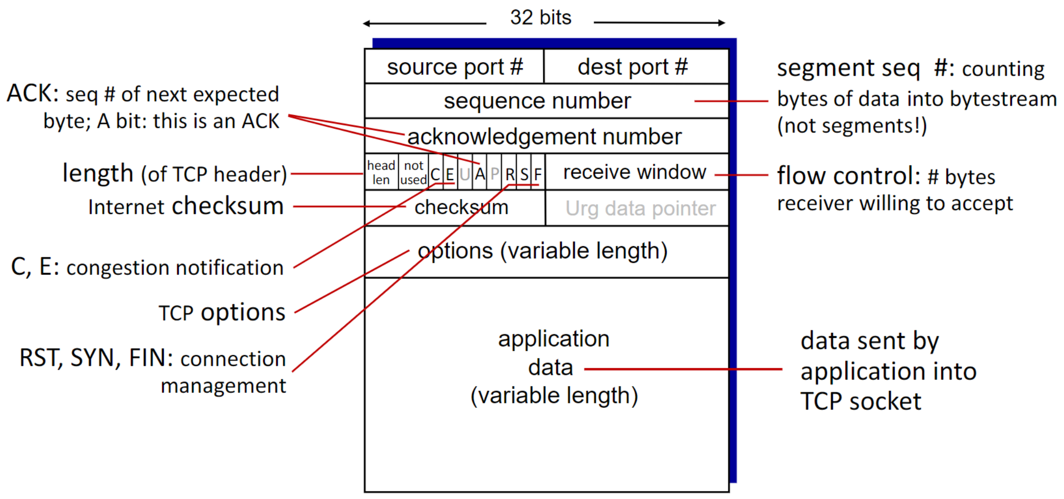
Transmission Control Protocol (TCP) Overview

- TCP is is/does the following:
  1. **Point-To-Point**: There is one sender, and one receiver.
  2. **Reliable, in-order byte stream**: There are no "message boundaries".
  3. **Full Duplex**: There is bi-directional data flow in a single connection.
  4. **Pipelined**: TCP congestion and flow-control can set the window size.
  5. **Connection-Oriented**: Handshaking is used to initalize the sender and receiver state before the data exchange begins.
  6. **Flow Controlled**: The sender will not overwhelm the receiver.
  7. **Supports cumulative ACKs**.
- The **TCP specification does not specify** how to handle **out-of-order packets**; it is up to the specific implementation.
- The **TCP timeout period** must be carefully set. If it is **too short**, premature timeout occurs. If it is **too long**, there is a slow reaction to segment loss.
- To estimate the **route-trip time (RTT)** you do the following:
  1. Measure the time starting when a single segmant transmisstion is sent, and stopping when you receive an ACK receipt. Set $EstimatedRTT$ equal to this time length.
  2. Repeat the process as your transmit more segments, updating the $EstimatedRTT$ with the following formula: $EstimatedRTT = (1 - \alpha) \times EstimatedRTT + \alpha \times SampleRTT$ (A typical α value is 0.125).

  - When actually setting the timeout period, you should add a **saftey margin**.

- **TCP Fast Re-Transmit**: When the **sender** receives **three ACKs** for **the same data**, the **sender** should **re-transmit all unACKed segments**.

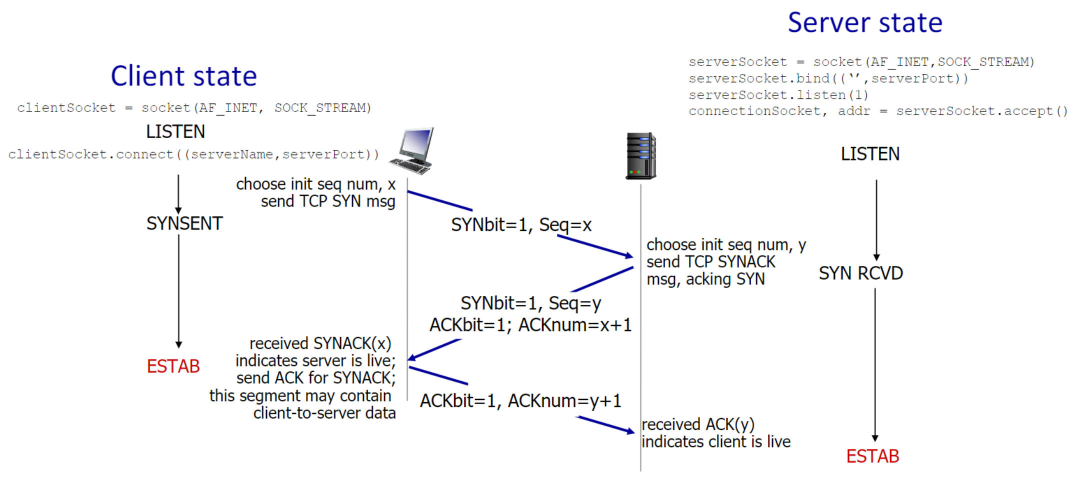
TCP Segment Structure


32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
C E U A P R S F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
ACK: seq # of next expected byte; A bit: this is an ACK
length (of TCP header)
Internet checksum
C, E: congestion notification
TCP options
RST, SYN, FIN: connection management
segment seq #: counting bytes of data into bytestream (not segments!)
flow control: # bytes receiver willing to accept
data sent by application into TCP socket

- **TCP segments** are **composed of 10 sections**:
  1. **Source Port**: The port the segment is coming from.
  2. **Destination Port**: The port the segment is going to.
  3. **Sequence Number**: The number of bytes in the bytestream.
  4. **Acknowledgements**: Sequence number of the next byte expected from the other side, and cumulative ACKs.
  5. **Length and Flags**: The length of the TCP header, and other flags.
  6. **Receive Window**: The number of bytes the receiver is willing to accept (flow control).
  7. **Checksum**: A checksum to verify the segment was received correctly.
  8. **Url Data Pointer**: Indicates how many bytes from the first are urgent.
  9. **TCP Options**: Additional options (variable length).
  10. **Applicaton Data**: The payload (variable length).

TCP Connection Creation

- **Before data** is **exchanged**, the **sender** and the **receiver** perform a **handshake**.
- This let's both sides know that **the other side agrees to establish a connection**, and that they **agree on connection parameters**.
- **TCP** uses a **3-way handshake** to establish a connection.



TCP Flow Control

- **TCP flow control** is achieved by the **receiver** setting the **window section of the TCP segment** specifying **how many segments it is able to receive**.
- This will prevent the receiver from being overwhelmed.

TCP Connection Termination

- To **terminate a TCP connection**, the **client** and the **server** both **close their side of the connection**; by sending a **TCP segment** with the **FIN** bit set to **1**.
- They respond to the **FIN** with **ACK**; the ACK response can be combined with FIN.
- Simultaneous FIN exchanges can be handled.

TCP Congestion Control: AIMD

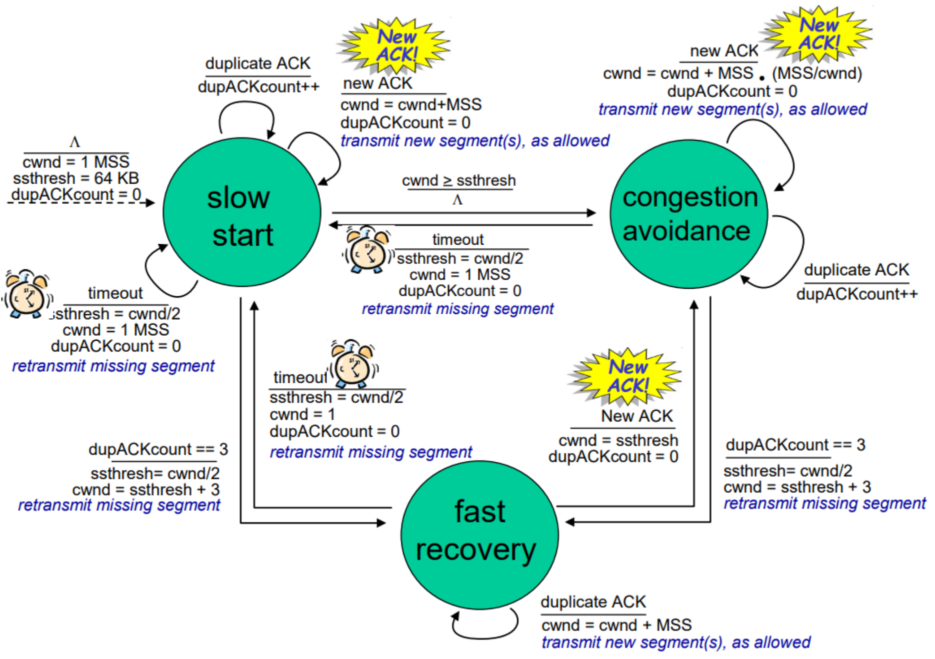
- **AIMD** stands for **Additive Increase, Multiplicative Decrease**.
- The **sender** can **increase the sending rate** until **congestion occurs**, then decrease.
  - When increasing the transmission rate, you increase the sending rate by 1 maximum segment size every RTT until loss is detected.
  - When decreasing the transmission rate, you cut the rate in half at each loss event.
- **AIMD** has been shown to **optomize congested flow rates network-wide**, and is stable.
- When a **connection is first established**, the segments should start off being sent **really slow, and increase exponentially until lose occurs**. After **loss occurs**, follow the **AIMD** principals.

TCP Congestion Congrol: Details

- **cwnd** is a dynamically adjusted variable, that is adjusted in response to observed network congestion (implementing TCP congestion control).
- TCP sends **cwnd bytes**, waits **RTT** for **acks**, then sends **more bytes**.
  - TCP Rate = $\frac{cwnd}{RTT}$ bytes/sec.
- The **TCP sender limits the transmission to LastByteSent - LastByteAcked ¡= cwnd**.

TCP Slow-Start

- When **connections are initated**, the **rate of transmission** is **increased exponentially** until the **first loss event**.
- Initially, **cwnd = 1 MSS**, and is **doubled** every **RTT** (performed by incrementing cwnd everything an ACK is recieved).
- The growth of **cwnd** should switch from an **exponential growth** to a **linear growth** when **cwnd gets to 1/2 of it's value before timeout**.
  - On a loss event, ssthresh is set to 1/2 * cwnd before the loss event occurred.



Delay-Based TCP Congestion Control

- The **goal** is to keep the **sender-to-receiver pipe "just fool eneough, but no fuller"**, meaning keep the bottleneck link busy, but avoid high delays/buffering.
- The **delay-based approach** takes the **minimum-RTT of an uncongested path**, the uncongested througput with congestion window cwnd is cwnd/minimum-RTT.
  - If the meased throughput is very close to the uncongested throughput, increase cwnd linearly.
  - If the measured throughput is far below the uncongested throughput, decrease cwnd linearly.