Transport Layer: TCP

Prof. Jean-Yves Le Boudec
Prof. Andrzej Duda
Prof. Patrick Thiran
ICA, EPFL

CH-1015 Ecublens
Andrzej.Duda@imag.fr
http://icawww.epfl.ch
Objectives

- Know different concepts of elements of procedure:
  - ARQ
  - sliding window
  - flow control
  - SRP
  - Go Back $n$
  - acknowledgements

- Describe UDP and know how to use it

- Understand how TCP works
  - connection finite state machine
  - TCP algorithms
  - flow control
  - congestion control

- Describe RTP/RTCP
Contents

- Part A: UDP
- Part B: Elements of Procedure
- Part C: TCP
- Part D: Additional algorithms
- Part E: Congestion Control
Part A. Transport Layer: UDP, TCP, ports, sockets

- **Remainder: transport layer**
  - network + data link + physical functions carry packets end-to-end
  - **transport layer** = makes network services available to programs
  - is end to end only, not in routers

- **in TCP/IP there are two transport protocols**
  - **UDP (User Datagram Protocol)**
    - unreliable
    - offers a datagram service to the application (unit of information is a message)
  - **TCP (Transmission Control Protocol)**
    - reliable
    - offers a stream service (unit of information is a byte)

- **application uses UDP or TCP depending on requirements**
  - **socket API**: a library of C functions
  - socket is similar to a file descriptor; controls a communication endpoint
    - is associated with an IP address, a port number
Transport Layer

Connectionless sockets

program

UDP

IP

Ethernet

id=3

id=4

buffer

buffer

port=32456

port=32654

address=128.178.151.84

address=FE:A1:56:87:98:12

socket

socket

socket

socket

socket

socket
Connection-oriented sockets

- Program
- TCP
- IP
- Ethernet

- id=3
- id=4
- id=5
- Socket
- Buffer
- Port=32456
- Address=128.178.151.84
Let us have a closer look at UDP, the unreliable transport protocol used in the Internet.

Two processes (= application programs) pa, and pb, are communicating. Each of them is associated locally with a port, as shown in the figure.

In addition, every machine (in reality: every communication adapter) has an IP address.

The example shows a packet sent by the name resolver process at host A, to the name server process at host B. The UDP header contains the source and destination ports. The destination port number is used to contact the name server process at B; the source port is not used directly; it will be used in the response from B to A.

The UDP header also contains a checksum the protect the UDP data plus the IP addresses and packet length. Checksum computation is not performed by all systems.
Transport Layer

Port Assignment

- Multiplexing based on source and destination numbers called port numbers
  - example: DNS query source port = ____, dest port = ____
  - Some ports are statically defined (well-known ports)
    - eg. DNS server port = 53
  - Other ports are allocated on request from application program (ephemeral ports)
    - eg. client port for DNS queries
  - Application level protocols specify the use of ports

- Examples: assigned ports
  - echo 7/UDP
  - discard 9/UDP
  - domain 53/UDP
  - talk 517/UDP
  - snmp 161/UDP
  - snmp-trap 161/UDP

Ports are 16 bits unsigned integers. They are defined statically or dynamically. Typically, a server uses a port number defined statically.

Standard services use well-known ports; for example, all DNS servers use port 53 (look at /etc/services).

Ports that are allocated dynamically are called ephemeral. They are usually above 1024.

If you write your own client server application on a multiprogramming machine, you need to define your own server port number and code it into your application (we will see how later).
The UDP service interface uses the concept of a message. If machine A sends one or several messages to machine B, then the messages are delivered exactly as they were submitted, without alteration, if they are delivered. Since UDP is unreliable and since some packet losses do occur, messages may be discarded silently. No one is informed, it is up to the application program to handle this.

For example, the query/response protocol of DNS specifies that queries are sent k times until a response is received or a timer expires. If the timer expires, the name resolver will try another domain name server, if it is correctly configured.

The UDP service does not go beyond one message. If two messages, M1 and M2, are sent one after the other from A to B, then in most cases, B will receive M1 before M2 (if no message is lost). However, it is possible, though infrequent, that B receives first M2 then M1.

UDP is used mainly for short transactions, or for real time multimedia applications:
- DNS queries
- Network File System
- Remote Procedure Call (RPC)
- any distributed program that requires raw, unreliable transport
- RTP, IP telephony

It is also used by all applications that use multicast, since TCP does not support it.
The UDP protocol

- A checksum is used to verify
  - source and destination addresses
  - port numbers
  - the data

- optional with IPv4, mandatory with IPv6

- based on pseudo-header method:
  - checksum is computed over: UDP datagram + pseudo header
  - checksum is put in UDP header; pseudo header is not transmitted

<table>
<thead>
<tr>
<th>IP src addr</th>
<th>IP dest addr</th>
<th>0</th>
<th>prot</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entire UDP datagram

checksum = 0

- sum modulo $2^{16} - 1$ all 16bit words
- checksum = 1’s complement

length = length of UDP datagram
Client Server Model

- processes (application programs) are associated (dynamically or statically) with port numbers
  - **dest port** used for presenting data to the corresponding program
    (= demultiplexing at destination)
  - **src port** stored by destination for responses

- server program
  - program that is ready to receive data at any time
    - on a given port
    - associated with a process running all time

- client program
  - program that sends data to a server program
    - does not expect to receive data before taking an initiative

- client server computing
  - server programs started in advance
  - client programs (on some other machines) talk to server programs
    - new tasks/processes and/or ports created as result of interaction

- See Module: Socket Interface

The diagram below illustrates encapsulated headers: fill in the name of the multiplexing schemes at every layer:

```
ports: application data
  | UDP hdr | UDP data
  | IP hdr  | IP data
  | LLC hdr | LLC frame
  | MAC hdr | MAC frame
```
Part B: Elements of Procedure

ARQ

- Errors in data transmission do occur:
  - corrupted data: noise, crosstalk, intersymbol interference
  - lost data: buffer overflow, control information error
  - misinserted data: control information error

- Automatic Repeat Request (ARQ)
  - one method to handle data errors
  - principle is:
    - 1. detection of error or loss
    - 2. retransmission
ARQ Protocols and Services

- There is not one ARQ protocol, but a family of protocols
  - examples:
    - TCP (transport layer of the Internet)
    - HDLC (data link layer over modems)
    - LLC-2 (data link layer over LANs, used with SNA)
    - Stop and Go (interface between a CPU and a peripheral)

- The service offered by all ARQ protocols is
  - ordered, loss-free, error-free delivery of packets

- ARQ protocol differs in their design choices
  - does the lower layer preserve order?
    - yes: HDLC, LLC-2
    - no: TCP

In the next slide we start with a simple example of ARQ protocol. Then we will study a systematic description of the concepts used by ARQ protocols in general.
Example of ARQ: Stop and Go

- **Definition**: ARQ method in which a packet is sent only after previous one is acknowledged.

- Utilization is low in general: \( r = \frac{b}{\omega + \beta/L} \)
  - with \( \omega = \frac{L}{I} \) = ratio of overhead and \( \beta = 2Db \) = bandwidth-delay product
  - used on half duplex or duplex links ("Idle RQ")

Examples (assume processing time + ack transmit times are negligible, 1000 bits per packet)

<table>
<thead>
<tr>
<th>Distance</th>
<th>1 km</th>
<th>200 km</th>
<th>50000 km</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 kb/s</td>
<td>100%</td>
<td>99.8%</td>
<td>67%</td>
</tr>
<tr>
<td>1 Mb/s</td>
<td>99%</td>
<td>33%</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

In presence of errors (rate of lost cycles = p), see exercise.
The limitation of Stop and Go is its little efficiency because of idle periods, when the bandwidth delay product is not negligible. A solution is to allow multiple transmissions, which in turn requires a sliding window in order to avoid unlimited buffer at the destination.

- **Objective**: Increase throughput of Stop and Wait by allowing multiple transmissions
- **Errors and Delivery in Order**: Requires unlimited buffer unless flow control is exercised
- **Solution**: Use a sliding window
On the example, packets are numbered 0, 1, 2, ..
The sliding window principle works as follows:
- a window size $W$ is defined. In this example it is fixed. In general, it may vary based on messages sent by the receiver. The sliding window principle requires that, at any time: number of unacknowledged packets at the receiver $\leq W$
- the maximum send window, also called offered window is the set of packet numbers for packets that either have been sent but are not (yet) acknowledged or have not been sent but may be sent.
- the usable window is the set of packet numbers for packets that may be sent for the first time. The usable window is always contained in the maximum send window.
- the lower bound of the maximum send window is the smallest packet number that has been sent and not acknowledged
- the maximum window slides (moves to the right) if the acknowledgement for the packet with the lowest number in the window is received

A sliding window protocol is a protocol that uses the sliding window principle. With a sliding window protocol, $W$ is the maximum number of packets that the receiver needs to buffer in the re-sequencing (= receive) buffer.

If there are no losses, a sliding window protocol can have a throughput of 100% of link rate (if overhead is not accounted for) if the window size satisfies: $W \geq b / L$, where $b$ is the bandwidth delay product, and $L$ the packet size. Counted in bytes, this means that the minimum window size for 100% utilization is the bandwidth-delay product.
Elements of ARQ

- The elements of an ARQ protocol are:
  - Sliding window
    - used by all protocols
  - Loss detection
    - at sender versus at receiver
    - timeout versus gap detection
  - Retransmission Strategy
    - Selective Repeat
    - Go Back n
    - Others
  - Acknowledgements
    - cumulative versus selective
    - positive versus negative

All ARQ protocols we will use are based on the principle of sliding window.

**Loss detection** can be performed by a timer at the source (see the Stop and Go example). Other methods are: gap detection at the destination (see the Go Back n example and TCP’s “fast retransmit” heuristics).

The **retransmission strategy** can be Go Back n, Selective Repeat, or any combination. Go Back n, Selective Repeat are explained in the following slides.

**Acknowledgements** can be cumulative: acknowledging “n” means all data numbered up to n has been received. Selective acknowledgements mean that only a given packet, or explicit list of packets is acknowledged.

A positive acknowledgement indicates that the data has been received. A negative acknowledgement indicates that the data has not been received; it is a request for retransmission, issued by the destination.
Selective Repeat (SRP) is a family of protocols for which the only packets that are retransmitted are the packets assumed to be missing. In the example, packets are numbered 0, 1, 2, ...

At the sender:

- A copy of sent packets is kept in the send buffer until they are acknowledged.
- The sequence numbers of unacknowledged packets differ by less than $W$ (window size).
- A new packet may be sent only if the previous rule is satisfied. The picture shows a variable ("upper bound of maximum send window") which is equal to: the smallest unacknowledged packet number + $W - 1$. Only packets with numbers $\leq$ this variable may be sent. The variable is updated when acknowledgements are received.

At the receiver:

- Received packets are stored until they can be delivered in order.
- The variable "lowest expected packet number" is used to determine if received packets should be delivered or not. Packets are delivered when a packet with number equal to this variable is received. The variable is then increased to the number of the last packet delivered + 1.
You can skip this slide at first reading.

**Proof** that $N \geq 2W$ is enough. Consider that at time $t$, the receiver just received packet number $n$ (absolute numbers) and delivered it, but that the ack of packet number $n$ is lost.

1. The sender can send new packets, but only up to packet $n+W-1$ (indeed, sender has not yet received the ack of packet $n$, so window rule at sender)
2. The sender has certainly received all acks until $n-W$ since it sent $n$.

So no packet with index smaller than or equal to $n-W$ can be received after time $t$ anymore.

The receiver will therefore only receive packets $n-W+1$ to $n+W-1$, at most $2W-1$ values, until the ack of packet $n$ is eventually received by the transmitter. Hence we can take $N = 2W$. 

---

**SRP: packet numbering**

- **packet numbering modulo $N$:**
  - packets are numbered $0, 1, 2, \ldots, N-1, 0, 1, 2, \ldots$

- **packet numbering modulo $N$ requires:** $N \geq 2W$
  - eg. modulo 128, maximum window size is 64

![Packet numbering modulo 7 with window size = 4; find the bug](image)
Go Back N (GBN) is a family of sliding window protocols that is simpler to implement than SRP, and possibly requires less buffering at the receiver. The principle of GBN is:

- If packet numbered \( n \) is lost, then all packets starting from \( n \) are retransmitted;
- Packets out of order need not be accepted by the receiver.

In the example, packets are numbered 0, 1, 2, …

At the sender:

A copy of sent packets is kept in the send buffer until they are acknowledged.

\((R1)\) The sequence numbers of unacknowledged packets differ by less than \( W \) (window size).

The picture shows two variables: \( V(S) \) (“Next Sequence Number for Sending”) which is the number of the next packet that may be sent, and \( V(A) \) (“Lowest Unacknowledged Number”), which is equal to the number of the last acknowledged packet + 1.

A packet may be sent only if: (1) its number is equal to \( V(S) \) and (2) \( V(S) \leq V(A) + W - 1 \). The latter condition is the translation of rule \( R1 \).

\( V(S) \) is incremented after every packet transmission. It is set (decreased) to \( V(A) \) whenever a retransmission request is activated (here, by timeout at the sender).

\( V(A) \) is increased whenever an acknowledgement is received. Acknowledgements are cumulative, namely, when acknowledgement number \( n \) is received, this means that all packets until \( n \) are acknowledged. Acknowledged packets are removed from the retransmission buffer.

At any point in time we have: \( V(A) \leq V(S) \leq V(A) + W \).
Go Back n: storing out of sequence packets or not

- Implementations of Go Back n may or may not keep packets out of sequence

- On a channel that preserves packet sequence:
  - an out-of-sequence packet can be interpreted as a loss and it is reasonable not to save it, since the Go Back n principle will cause it to be retransmitted
  - HDLC, LLC-2

- On a channel that does not preserve packet sequence
  - out-of-sequence packets may simply be due to packet misordering, and it is reasonable to save it, hoping that packets with lower numbers are simply delayed (and not lost).
  - TCP

At the receiver:

the variable $V(R)$ “lowest expected packet number” is used to determine if a received packet should be delivered or not.

A received packet is accepted, and immediately delivered, if and only if its number is equal to $V(R)$. On this example, the receiver rejects packets that are not received in sequence.

$V(R)$ is incremented for every packet received in sequence. Packets received in sequence are acknowledged, either immediately, or in grouped (cumulative) acknowledgement. When the receiver sends acknowledgement $n$, this always means that all packets until $n$ have been received in sequence and delivered.

The picture shows acknowledgements that are lost, and some that are delayed. This occurs for example if there are intermediate systems in the data path that are congested (buffers full, or almost full). Such intermediate systems could be IP routers (the protocol would then be TCP) or bridges (the protocol would be LLC-2).
Go Back n (continued 2)

- Principle of Go Back n
  - if packet n is lost, all packets from n are retransmitted
  - out of order packets need not be accepted by receiver

- Less efficient than SRP, but good efficiency with lower complexity
  - cumulative acknowledgements

- On a sequence preserving channel (eg. layer 2): Packet Numbering is usually done modulo M (typically 8 or 128) – see next slide
  - packets are numbered 0, 1, 2, ..., M-1, 0, 1, 2, ...

- With TCP (non-sequence preserving channel: IP) bytes are numbered modulo 2^{32}
  - maximum window size is set to 2^{16} (except with window scale option)
  - byte numbers can be considered absolute numbers (except with “window scale option”, where window size is 2^{16} x 2^{scale}, scale = 0, 14; max. window size = 2^{30}).
Go Back $n$: packet numbering

- On a sequence preserving channel (e.g., layer 2): Packet Numbering is usually done modulo $M$ (typically 8 or 128)
  - packets are numbered $0, 1, 2, \ldots, M-1$, $0, 1, 2, \ldots$
- packet numbering modulo $N$ requires: $N \geq W + 1$

eg. modulo 128, maximum window size is 127

You can skip this slide at first reading.

Proof that $N \geq W + 1$ is enough. Consider that at time $t$, the receiver just received packet number $n$ (absolute numbers) and delivered it, but that the ack of packet number $n$ is lost.

(1) The sender can send new packets, but only up to packet $n+W-1$
(2) Depending on which transmissions arrive with or without errors, $V(R)$ will take a value between $(n+1) \mod N$ and $(n+W) \mod N$.

The receiver will therefore never receive packet $n+W$ unless the ack of packet $n$ has been received before by the transmitter. Hence we can take $N = W$. 

You can skip this slide at first reading.

Proof that $N \geq W + 1$ is enough. Consider that at time $t$, the receiver just received packet number $n$ (absolute numbers) and delivered it, but that the ack of packet number $n$ is lost.

(1) The sender can send new packets, but only up to packet $n+W-1$
(2) Depending on which transmissions arrive with or without errors, $V(R)$ will take a value between $(n+1) \mod N$ and $(n+W) \mod N$.

The receiver will therefore never receive packet $n+W$ unless the ack of packet $n$ has been received before by the transmitter. Hence we can take $N = W$. 

You can skip this slide at first reading.
Negative acknowledgements are an alternative to timeouts at sender for triggering retransmissions.

Negative acknowledgements are generated by the receiver, upon reception of an out-of-sequence packet.

\[ \text{NACK, } A=n \text{ means: all packets until } n \text{ have been received in sequence and delivered, and packets after } n \text{ should be retransmitted} \]

Negative acknowledgements are implemented mainly on sequence preserving channels (for example, with HDLC, LLC-2, etc). They are not defined with TCP.

Negative acknowledgements increase the retransmission efficiency since losses can be detected earlier.

Since NACKs may be lost, it is still necessary to implement timeouts, typically at the sender. However, the timeout value may be set with much less accuracy since it is used only in those cases where both the packet and the NACK are lost. In contrast, if there are no NACKs, it is necessary to have a timeout value exactly slightly larger than the maximum response time (round trip plus processing, plus ack withholding times).
ARQ Protocols

- **HDLC** protocols (used on Modems)
  - SRP with selective acknowledgements
  - Go Back n on old equipment

- **LLC-2**
  - old, used with SNA over bridged networks
  - Go Back n
  - with negative acknowledgements

- **TCP**
  - a hybrid
  - originally designed as Go Back n with cumulative acks
  - today modern implementation have both
    - selective acks
    - cumulative acks

Example of timeout for loss detection at receiver: in NETBLT, sender tells in advance which PDUs it will send.

SSCOP is a protocol used in signalling networks for telephony, ATM: periodic poll by sender with list of sent PDUs; response (solicited stat) by receiver with missing PDUs + credit
Error Correction Alternatives

- ARQ suited for traditional point to point data
  - variable delay due to retransmissions
  - correct data is guaranteed

- Alternative 1: forward error correction
  - principle
    - add redundancy (Reed Solomon codes) and use it to recover errors and losses
  - for real time traffic (voice, video, circuit emulation)
  - or on data on links with very large delays (satellites)

- Alternative 2: use of parities
  - code n blocks of data into k parities (RS codes)
  - any n out of n+k blocks can be used to recover the data
  - used for multicast
Flow Control

- Purpose: prevent buffer overflow at receiver
  - receiver not ready (software not ready)
  - many senders to same receiver (overload focused on receiver)
  - receiver slower than sender

- Solutions: Backpressure, Sliding Window, Credit

- Flow Control is not the same as Congestion control (inside the network)
Backpressure Flow Control

- Stop and Go principle
- Implemented in
  - RTS / CTS
  - ATM backpressure (Go / No Go) (IBM ATM LAN)
  - Pause packets in 802.3x
- Can be combined with ARQ acknowledgement schemes or not
- Receiver requires storage for the maximum round trip per sender
- Requires low bandwidth delay product
Sliding Window Flow Control

- Number of packets sent but unacknowledged $\leq W$
- Included in SRP and Go Back N protocols
  - assuming acknowledgements sent when receive buffer freed for packets received in order
- Receiver requires storage for at most $W$ packets per sender

Sliding window protocols have an automatic flow control effect: the source can send new data if it has received enough acknowledgements. Thus, a destination can slow down a source by delaying, or withholding acknowledgements.

This can be used in simple devices; in complex systems (for example: a computer) this does not solve the problem of a destination where the data are not consumed by the application (because the application is slow). This is why such environments usually implement another form, called credit based flow control.
With a credit scheme, the receiver informs the sender about how much data it is willing to receive (and have buffer for). Credits may be the basis for a stand-alone protocol (Gigaswitch protocol of DEC, similar in objective to ATM backpressure) or, as shown here, be a part of an ARQ protocol. Credits are used by TCP, under the name of “window advertisement”. Credit schemes allow a receiver to share buffer between several connections, and also to send acknowledgements before packets are consumed by the receiving upper layer (packets received in sequence may be ready to be delivered, but the application program may take some time to actually read them).

The picture shows the maximum send window (called “offered window” in TCP) (red border) and the usable window (pink box). On the picture, like with TCP, credits (= window advertisements) are sent together with acknowledgements. The acknowledgements on the picture are cumulative.

Credits are used to move the right edge of the maximum send window. (Remember that acknowledgements are used to move the left edge of the maximum send window).

By acknowledging all packets up to number \( n \) and sending a credit of \( k \), the receiver commits to have enough buffer to receive all packets from \( n+1 \) to \( n+k \). In principle, the receiver (who sends acks and credits) should make sure that \( n+k \) is non-decreasing, namely, that the right edge of the maximum send window does not move to the left (because packets may have been sent already by the time the sdr receives the credit).

A receiver is blocked from sending if it receives credit = 0, or more generally, if the received credit is equal to the number of unacknowledged packets. By the rule above, the received credits should never be less than the number of unacknowledged packets.

With TCP, a sender may always send one byte of data even if there is no credit (window probe, triggered by persistTimer) and test the receiver’s advertised window, in order to avoid deadlocks (lost credits).
Transport Layer

The figure shows the relation between buffer occupancy and the credits sent to the source. This is an ideal representation. Typical TCP implementations differ because of misunderstandings by the implementers.

The picture shows how credits are triggered by the status of the receive buffer. The flows are the same as on the previous picture.

The receiver has a buffer space of 4 data packets (assumed here to be of constant size for simplicity). Data packets may be stored in the buffer either because they are received out of sequence (not shown here; some ARQ protocols such as LLC-2 simply reject packets received out of sequence), or because the receiving application, or upper layer, has not yet read them.

The receiver sends window updates (=credits) in every acknowledgement. The credit is equal to the available buffer space.

Loss conditions are not shown on the picture. If losses occur, there may be packets stored in the receive buffer that cannot be read by the application (received out of sequence). In all cases, the credit sent to the source is equal to the buffer size, minus the number of packets that have been received in sequence. This is because the sender is expected to move its window based only on the smallest ack number received. See also exercises.
Solution Exercise 1a

A = -1, credit = 2
P = 0

A = 0, credit = 2
P = 1

A = 0, credit = 4
P = 2

P = 3

P = 4

P = 5

A = 2, credit = 4

P = 6

A = 4, credit = 2

P = 5

A = 5, credit = 1
Connection Control

Reliable communication with ARQ must be connection oriented
- sequence numbers must be synchronized
- thus, reliable transport protocols always have three phases:
  - setup
  - data transfer
  - release

Connection Control = connection setup and connection release
- see TCP part for the TCP connection control states of TCP
Applications that use UDP must implement some form of equivalent control, if the application cares about not losing data.

UDP applications (like NFS, DNS) typically send data blocks and expect a response, which can be used as an application layer acknowledgement.
B: TCP: Transmission Control Protocol

- Provides a reliable transport service
  - first, a connection is opened between two processes
  - then TCP guarantees that all data is delivered in sequence and without loss, unless the connection is broken
  - in the end, the connection is closed

- Uses port numbers like UDP
  - eg. TCP port 53 is also used for DNS
  - TCP connection is identified by: src IP addr, src port, dest IP addr, dest port

- TCP does not work with multicast IP addresses, UDP does
- TCP uses connections, UDP is connectionless
TCP is an ARQ protocol

- **Basic operation:**
  - sliding window
  - loss detection by timeout at sender
  - retransmission is a hybrid of go back and selective repeat
  - cumulative

- **Supplementary elements**
  - fast retransmit
  - selective acknowledgements

- **Flow control is by credit**

- **Congestion control**
  - adapt to network conditions

TCP also implements congestion control functions, which have nothing to do with ARQ (see later). Do not confuse flow control and congestion control.
TCP views data as a stream of bytes

- bytes put in packets called TCP segments
  - bytes accumulated in buffer until sending TCP decides to create a segment
  - MSS = maximum “segment” size (maximum data part size)
    - “B sends MSS = 236” means that segments, without header, sent to B should not exceed 236 bytes
  - 536 bytes by default (576 bytes IP packet)
- sequence numbers based on byte counts, not packet counts
- TCP builds segments independent of how application data is broken
  - unlike UDP
- TCP segments never fragmented at source
  - possibly at intermediate points with IPv4
  - where are fragments re-assembled?
- TCP uses a sliding window protocol with automatic repetition of lost packets (in other words, TCP is an ARQ protocol). The picture shows a sample exchange of messages. Every packet carries the sequence number for the bytes in the packet; in the reverse direction, packets contain the acknowledgements for the bytes already received in sequence. The connection is bidirectional, with acknowledgements and sequence numbers for each direction. Acknowledgements are not sent in separate packets ("piggybacking"), but are in the TCP header. Every segment thus contains a sequence number (for itself), plus an ack number (for the reverse direction). The following notation is used:

  firstByte:"lastByte+1 "("segmentDataLength") ack" ackNumber+1
  "win" offeredWindowSize. Note the +1 with ack and lastByte numbers.

- At line 8, a retransmission timer expires, causing the retransmission of data starting with byte number 8501 (Go Back n principle). Note however that after segment 9 is received, transmission continues with byte number 10001. This is because the receiver stores segments received out of order.

- The window field (win) gives to the sender the size of the window. Only byte numbers that are in the window may be sent. This makes sure the destination is not flooded with data it cannot handle.

- Note that numbers on the figure are rounded for simplicity. Real examples use non-round numbers between 0 and $2^{32} - 1$. The initial sequence number is not 0, but is chosen at random using a 4 µsec clock.

The figure shows the implementation of TCP known as "TCP Reno", which is the basis for current implementations. An earlier implementation ("TCP Tahoe") did not reset the pending timers after a timeout; thus, this was implementing a true Go Back n protocol; the drawback was that packets were retransmitted unnecessarily, because packet losses are usually simple.
Fast Retransmit

- Issue: retransmission timeout in practice often very large
  - earlier retransmission would increase efficiency
  - add SRP behaviour

- **Fast retransmit** heuristics
  - if 3 duplicate acks for the same bytes are received before retransmission timeout, then retransmit

- implemented in all modern versions of TCP; is an IETF standard.
Selective Acknowledgements

- Newest TCP versions implement selective acknowledgements ("TCP-SACK")
  - up to 3 SACK blocks are in TCP option, on the return path
  - a SACK block is a positive ack for an interval of bytes
  - first block is most recently received
  - allows a source to detect a loss by gap in received acknowledgement
- TCP-SACK (Fall and Floyd, 1996):
  - when a loss is detected at source by means of SACK, fast retransmit is entered
  - when all gaps are repaired, fast retransmit is exited

For the (complex) details, see:

"Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", Kevin Fall and Sally Floyd, IEEE ToN, 1996

Before data transfer takes place, the TCP connection is opened using SYN packets. The effect is to synchronize the counters on both sides.

The initial sequence number is a random number.

The connection can be closed in a number of ways. The picture shows a graceful release where both sides of the connection are closed in turn.

Remember that TCP connections involve only two hosts; routers in between are not involved.
If the application issues a half-close (e.g. shutdown(1)) then data can be received in states FIN_WAIT_1 and FIN_WAIT_2. “TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request” (RFC 793). The connection stays in that state for a time of 2*MSL, where MSL = maximum segment lifetime (typically 2*2 mn). This also has the effect that the connection cannot be reused during that time.

Entering the FIN_WAIT_2 state on a full close (not on a half-close) causes the FIN_WAIT_2 timer to be set (e.g. 10 mn). If it expires, then it is set again (e.g. 75 sec) and if it expires again, then the connection is closed. This is to avoid connections staying in the half-close state for ever if the remote end disconnected.

Transitions due to RESET segments except the 2nd case are not shown on the diagram.

There is a maximum number of retransmissions allowed for any segment. After R1 retransmissions, reachability tests should be performed by the IP layer. After unsuccessful transmission lasting for at least R2 seconds, the connection is aborted. Typically, R1 = 3 and R2 is a few minutes. R2 can be set by the application and is typically a few minutes. Transitions due to those timeouts are not shown.

The values are usually set differently for a SYN packet. With BSD TCP, if the connection setup does not succeed after 75 sec (= connectionEstablishmentTimer), then the connection is aborted.

The diagram does not show looping transitions; for example, from TIME-WAIT state, reception of a FIN packet causes an ACK to be sent and a loop into the TIME-WAIT state itself.
Reset segments are used to abort a connection. See RFCs 793 and 1122 for an exact description. In short, they are sent:

- when a connection opening is attempted to a port where no passive open was performed;
- while in the SYN_SENT state, when a segment with invalid ack number arrives;
- when an application performs a “half-close”: application calls close and then (by the definition of the half close system call) cannot read incoming data anymore. In that case, incoming data is discarded and responded to by a Reset segment.

On reception of a Reset segment, it is checked for validity (seq number) and if it is valid:

- in the LISTEN state, it is ignored
- in the SYN_RCVD state, the state is moved to LISTEN provided it was the state from which SYN_RCVD was entered
- otherwise, the connection is aborted (move to CLOSED state)

### Resetting a TCP connection

Here is an example of use of the RESET segment:

<table>
<thead>
<tr>
<th></th>
<th>TCP A</th>
<th>TCP B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(CRASH)</td>
<td>(send 300, receive 100)</td>
</tr>
<tr>
<td>2</td>
<td>CLOSED</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>3</td>
<td>SYN-SENT --&gt; &lt;SEQ=400&gt;&lt;CTL=SYN&gt;</td>
<td>--&gt; (?)</td>
</tr>
<tr>
<td>4</td>
<td>(!!)</td>
<td>&lt;--&gt; &lt;SEQ=300&gt;&lt;ACK=100&gt;&lt;CTL=ACK&gt; &lt;-- ESTABLISHED</td>
</tr>
<tr>
<td>5</td>
<td>SYN-SENT --&gt; &lt;SEQ=100&gt;&lt;CTL=RST&gt;</td>
<td>--&gt; (Abort!!)</td>
</tr>
<tr>
<td>6</td>
<td>SYN-SENT</td>
<td>CLOSED</td>
</tr>
<tr>
<td>7</td>
<td>SYN-SENT --&gt; &lt;SEQ=400&gt;&lt;CTL=SYN&gt;</td>
<td>--&gt;</td>
</tr>
</tbody>
</table>
Ex. 5.11: simultaneous active close.
the push bit can be used by the upper layer using TCP; it forces TCP on the sending side to create a segment immediately. If it is not set, TCP may pack together several SDUs (=data passed to TCP by the upper layer) into one PDU (= segment). On the receiving side, the push bit forces TCP to deliver the data immediately. If it is not set, TCP may pack together several PDUs into one SDU. This is because of the stream orientation of TCP. TCP accepts and delivers contiguous sets of bytes, without any structure visible to TCP. The push bit used by Telnet after every end of line.

- the urgent bit indicates that there is urgent data, pointed to by the urgent pointer (the urgent data need not be in the segment). The receiving TCP must inform the application that there is urgent data. Otherwise, the segments do not receive any special treatment. This is used by Telnet to send interrupt type commands.

- RST is used to indicate a RESET command. Its reception causes the connection to be aborted.

- SYN and FIN are used to indicate connection setup and close. They each consume one sequence number.

- The sequence number is that of the first byte in the data. The ack number is the next expected sequence number.

- Options contain for example the Maximum Segment Size (MSS) normally in SYN segments (negotiation of the maximum size for the connection results in the smallest value to be selected).

- The checksum is mandatory.
There are various methods, rules and algorithms that are part of the TCP specification. Some of those appeared as more experience was gained with the early implementations of TCP. They address two kinds of issues:

- specify parts of the protocol for which there is freedom of implementation. For example: when to send ACKs, when to send data, when to send window advertisements (updates of the offered window)
- address bugs that were discovered in the field: silly window syndrome avoidance
- improve algorithms for round trip estimation and setting retransmission timer values
- improve performance by allowing early retransmissions

When to send ACKs is an issue that is not fully specified. However, RFC 1122 gives implementation guidance. When receiving a data segment, a TCP receiver may send an acknowledgement immediately, or may wait until there is data to send ("piggybacking"), or until other segments are received (cumulative ack). Delaying ACKs reduces processing at both sender and receiver, and may reduce the amount of IP packets in the network. However, if ACKs are delayed too long, then receivers do not get early feedback and the performance of the ARQ scheme decreases. Also, delaying ACKs also delays new information about the window size.

RFC 1122 recommends to delay ACKs but for less than 0.5 s. In addition, in a stream of full size segments, there should be at least one ACK for every other segment.

Note that a receiving TCP should send ACKs (possibly delayed ACKs) even if the received segment is out of order. In that case, the ACK number points to the last byte received in sequence + 1.
Nagle’s Algorithm

- Example: Telnet Application: byte sent one by one by client, echoed by server. Overhead = ______
  - solution 1: delay sending data based on timer (X.25 pads)
  - problem: unnecessary delay on LANs
- the TCP solution (Nagle’s Algorithm):
  - accept only one unacknowledged tinygram (= segment smaller than MSS)
- Purpose of Nagle’s algorithm: avoid sending small packets (“Tinygrams”) (data written by upper layer) or (new ack received) ->
  - if full segment ready
    - then send segment
  - else if there is no unacknowledged data
    - then send segment
  - else start override timer; leave
  - override timer expires -> create segment and send segment
- Nagle’s algorithm can be disabled by application
  - example: X window system
  - (TCP_NODELAY socket option)
  - if Nagle enabled, then applies also to pushed data
Example: Nagle's algorithm

A

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>A (1) ack 101 win 6000</td>
</tr>
<tr>
<td>b</td>
<td>B</td>
</tr>
<tr>
<td>c</td>
<td>C (1) ack 8001 win 14000</td>
</tr>
<tr>
<td>d</td>
<td>D</td>
</tr>
<tr>
<td>e</td>
<td>E (2) ack 8003 win 13998</td>
</tr>
<tr>
<td>f</td>
<td>F</td>
</tr>
</tbody>
</table>

B

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8000:8001</td>
</tr>
<tr>
<td>2</td>
<td>8001:8003</td>
</tr>
<tr>
<td>3</td>
<td>8002:8004</td>
</tr>
<tr>
<td>4</td>
<td>8003:8005</td>
</tr>
<tr>
<td>5</td>
<td>8004:8006</td>
</tr>
<tr>
<td>6</td>
<td>8005:8007</td>
</tr>
<tr>
<td>7</td>
<td>8006:8008</td>
</tr>
<tr>
<td>8</td>
<td>8007:8009</td>
</tr>
</tbody>
</table>
Silly Window Syndrome

- Silly Window Syndrome caused by small window advertisements:

  - ack 0 win 2000 <-----
  - 0:1000 -------> bufferSize= 2000B, freebuf= 1000B
  - 1000:2000 -------> freebuf= 0B
  - ack 2000, win 0 <------

  - application reads 1 Byte: freeBuf = 1

  - ack 2000, win 1 <------
  - 2000:2001 -------> freeBuf = 0
  - application reads 1 Byte: freeBuf = 1

  - ack 2001, win 1 <------
  - 2001:2002 -------> freeBuf = 0
  - application reads 1 Byte: freeBuf = 1

  - ack 2002, win 1 <------
  - 2002:2003 -------> freeBuf = 0

- Silly Window Syndrome
  - sender has large amount of data to send
  - but small window forces sender to send small packets

SWS occurs when a slow receiver cannot read data fast enough, and reads them in small increments. The window advertisement method of TCP has the effect of avoiding buffer overflow at the receiver (flow control), however, if no additional means are taken, it results in a large number of small packets to be sent, with no benefit to the receiver since anyhow it cannot read them fast enough. The (new) TCP specification mandates that sender and receiver should implement SWS avoidance.

SWS avoidance at the receiver simply forces the window to move by large increments. As data is read from the receive buffer, the upper window edge could be moved to the right. However, the SWS avoidance algorithm specifies that this should be done only if the upper window edge can be moved by at least the value of one full segment, or, if the buffer is small, by F* receiveBuffer. As a result, there may be a fraction of the buffer (“reserve” on the picture) which is not advertised.

SWS avoidance at receiver is sufficient, however TCP senders must also implement SWS avoidance, in order to cope with receivers that do not comply with the latest standard. SWS avoidance at sender is in addition to Nagle’s algorithm (whose objective is to avoid small packets, which otherwise may occur even if the offered window is large).

The picture "SWS Avoidance Example" also shows a window probe, which aims at avoiding deadlocks if some acks are lost.
Silly Window Syndrome Avoidance: Receiver

- SWS avoidance is implemented by receiver moving the window by increments that are as large as one MSS or 1/2 receiveBuffer.

\[
\begin{align*}
\text{highestByteRead} & \quad \text{nextByteExpected} \\
\downarrow & \quad \downarrow \\
\rightarrow \quad \rightarrow \\
\text{offeredWindow} & \quad \text{reserve} \\
\text{receiveBuffer} & \\
\text{acked} & \quad \text{not ready to read} & \quad \text{not} & \quad \text{advertized}
\end{align*}
\]

- SWS avoidance (receiver)

\[
\text{keep } \text{nextByteExpected} + \text{offeredWindow} \text{ fixed until:}
\]
\[
\text{reserve} \geq \min (\text{MSS, 1/2 receiveBuffer})
\]
SWS Avoidance Example

ack 0 win 2000 <------
  0:1000 -------> bufferSize= 2000B, freebuf = 1000B, reserve = 0B
  1000:2000 -------> freebuf= 0B, reserve = 0B
ack 2000, win 0 <------

application reads 1 Byte: freeBuf=reserve=1B,

....

application has read 500 B: reserve = 500

persistTimer expires
window probe packet sent

2000:2001 ------>

data is not accepted (out of window)

ack 2000, win 0 <------

....

application has read 1000 B: reserve = 1000

ack 2000, win 1000 <------
  2000:3000 ------->
Silly Window Syndrome Avoidance: Sender

- **Purpose:**
  - delay sending small packets due to small (“silly”) window advertisements

- **At Sender:**
  - consider sending data only if data is written and
    push option called on this data
  - or at least 50% of the maximum (estimated) receiver’s window is ready to send
  - then call Nagle’s algorithm if enabled for that connection

- Estimated receiver window = maximum offered window advertised
- Used to deal with receivers that do not implement receiver SWS avoidance
- What is the difference in objective between SWS avoidance at sender and Nagle’s algorithm?
**Silly Window Syndrome Avoidance: Sender**

- What is the difference in objective between SWS avoidance at sender and Nagle’s algorithm?

  **SWS avoidance**: Used to avoid sending small segments when there are lots of data to transmit (small segments would be caused by offered windows).

  **Nagle**: Used to avoid sending small segments when application is doing small writes.

- SWS avoidance at sender is for data that is not pushed.
transport layer

round trip estimation

- retransmission is triggered by retransmission timer. Which value for the timer?
  - slightly larger than round trip time

- issue: what is the round trip time?

- tcp solution: rto (round trip estimator) given by jacobson and karels formula:

  sampleRTT = last measured round trip time
  estimatedRTT = last estimated average round trip time
  deviation = last estimated round trip deviation

  initialization (first sample):
  - estimatedRTT = sampleRTT + 0.5s; deviation = estimatedRTT/2

  new value of sampleRTT available ->
  - Err = sampleRTT - estimatedRTT
  - estimatedRTT = estimatedRTT + 0.125 * Err
  - deviation = deviation + 0.250 * (|Err| - deviation)
  - RTO = estimatedRTT + 4*deviation
Round trip estimation is based on a low pass filter. Originally, the first TCP specification used a formula similar to estimatedRTT. However, it became apparent that RTT estimates fluctuate a lot, with fluctuations sometimes meaning a change of path. The formula is based on estimation of both average and deviation (which is an estimator of the absolute value of the error). The coefficients 0.125 and 0.25 (the estimation gains) are chosen to be simple negative powers of 2, which makes implementation of the multiplication simple (a bit shift). The specific values were tuned based on measured data.

In practice, most OSs do not check timeouts individually, but rather implement a timeout routine that wakes up periodically. On BSD Unix, such a routine wakes up every 0.5 seconds, which means that timers are computed with a granularity of 500 ms. This results in retransmission timeouts that may occur almost 500 ms after the due time. The same granularity applies to the RTP estimation.

There is a special rule for the beginning of a connection, before any valid sample is available:

- as long as no sample is available, \( RTO = 6 \) s (first packet sent), \( RTO = 12 \) s afterwards
- and for the Karn’s algorithm, everything is done as though the first timeout was set to 12 s.

Those rules apply in conjunction with Karn's algorithm. Thus the timer for the SYN packet is 6 s; if it expires, then the next timeout is set to \( 12 \times 2 = 24 \) s, etc.
Karn and Partridge rules

- **Rule 1** ("Karn’s algorithm"): measure round trip only for fresh transmissions
  - other cases lead to ambiguities

- **Rule 2** (Timer Exponential Backoff)
  - when `retransmissionTimer` expires, double the value of `retransmissionTimer`
    (do not use the RTO formula, use the previous value of `retransmissionTimer`)
    - purpose: avoid short timeouts due to no valid sample being gathered (because of Rule 1)
    - maximum is 240 s.
In the Internet, all TCP sources sense the status of the network and adjust their window sizes accordingly. Reducing the window size reduces the traffic. The principle is to build a feedback system, with:
- a packet loss is a negative feedback; a useful acknowledgement is a positive feedback;
- additive increase, multiplicative decrease: when a TCP source senses no loss, it increases its window size linearly; when a loss is detected, it reduces the window size by 50%.

It can be shown that, if all sources have the same round trip time, then all sources converge towards using a fair share of the network. In general, though, sources with large round trip times have a smaller share of the network bandwidth.

If some part of the Internet has a large volume of traffic not controlled by TCP, then there is a risk of congestion collapse. This is in particular caused by Internet telephony applications. In the future, all applications, even those which use UDP, will have to imitate the behaviour of TCP, at least as far as sending rates are concerned.
Slow Start and Congestion Avoidance

connection opening: $\text{twnd} = 65535$ B
$cwnd = 1$ seg

**Slow Start**

- exponential increase for $cwnd$ until $cwnd = \text{twnd}$

**Congestion Avoidance**

- additive increase for $\text{twnd}$, $cwnd = \text{twnd}$

retransmission timeout:
- multiplicative decrease for $\text{twnd}$
- $cwnd = 1$ seg

$cwnd = \text{twnd}$

notes
this shows only 2 states out of 3
$\text{twnd} = \text{target window}$
Increase/decrease

- Multiplicative decrease
  - $\text{twnd} = 0.5 \times \text{cwnd}$
  - $\text{twnd} = \max (\text{twnd}, 2 \times \text{MSS})$

- Additive increase
  - for each ACK
    - $\text{twnd} = \text{twnd} + \text{MSS} \times \text{MSS} / \text{twnd}$ ($w\leftarrow w+1$)
    - $\text{twnd} = \min (\text{twnd}, \text{max-size})$ (64KB)

- Exponential increase
  - for each ACK
    - $\text{cwnd} = \text{cwnd} + \text{MSS}$
    - if ($\text{cwnd} == \text{twnd}$) go to avoidance phase
Slow Start and Congestion Avoidance Example

created from data from: IEEE Transactions on Networking, Oct. 95, "TCP Vegas", L. Brakmo and L. Peterson
Slow Start and Congestion Avoidance Example

```
created from data from: IEEE Transactions on Networking, Oct. 95, “TCP Vegas”, L. Brahma and L. Petersen
```
twnd Additive Increase

During one round trip + interval between packets: increase by 1 packet (linear increase)

( equivalent to twnd = twnd + 1/twnd if TCP would have all segments of length MSS)
**Transport Layer**

### Slow Start

- purpose of this phase: avoid burst of data after a retransmission timeout
- /* exponential increase for cwnd */

```plaintext
non dupl. ack received during slow start ->
  cwnd = cwnd + seg (in bytes)
  if cwnd = twnd then transition to congestion avoidance
```

- window increases rapidly up to the stored value of twnd (this stored value is called ssthresh in the literature)
Fast Recovery Example

A-B, E-F: fast recovery
C-D: slow start
Transport Layer

Congestion States

- Slow Start
  - exponential increase
  \[ \text{cwnd} = \text{twnd} \]
  \[ \text{retr. timeout} \]
- Fast Recovery
  - exponential increase beyond twnd
  \[ \text{expected ack received} \]
- Fast Retransmit
  \[ \text{fast retransmit: retransmit} \]
- Retransmit

new connection:

Congestion Avoidance
- additive increase

\[ \text{expected ack received} \]
Fast Recovery

- **Multiplicative decrease**
  - $twnd = 0.5 \times cwnd$
  - $twnd = \max (twnd, 2 \times MSS)$

- **Fast Recovery**
  - $cwnd = twnd + 3 \times MSS$ (exp. increase)
  - $cwnd = \min (cwnd, 64K)$
  - retransmission of the missing segment ($n$)

- **For each duplicated ACK**
  - $cwnd = cwnd + MSS$ (exp. increase)
  - $cwnd = \min (cwnd, 64K)$
  - send following segments
Congestion Control: Summary

- Congestion control aims at avoiding congestion collapse in the network
- With TCP/IP, performed in end-systems, mainly with TCP
  **TCP Congestion control summary**
  - Principle: sender increases its sending window until losses occur, then decrease
    
    - additive increase
    - multiplicative decrease
    - slow start
    - fast recovery
    - congestion avoidance

  - target window: additive increase (no loss), multiplicative decrease (loss)
  - 3 phases:
    - **slow start**: starts with 1, exponential increase up to $\tau_{wnd}$
    - **congestion avoidance**: additive increase until loss or no increase
    - **fast recovery**: fast retransmission of one segment

- slow start entered at setup or after retransmission timeout
- fast recovery entered at fast retransmit