Computer Networks (ComNet) 3/5: Transport

O. Fourmaux - T. Friedman
(olivier.fourmaux@upmc.fr)

Version 7.0

ComNet: course 3/5 outline

1. Basic services
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol

2. Reliable service
   - Principles of reliable data transfer
   - TCP: a connection oriented protocol
   - TCP: reliability mechanisms

3. Congestion control
   - General principles
   - TCP mechanisms
   - SCTP and DCCP

Transport layer

Understanding of basic principles of the transport layer
- multiplexing
- reliable transfer
- flow control
- congestion control

Study of transport protocols in the Internet
- UDP: connectionless transport
- TCP: connection oriented transport with reliability and congestion control
- SCTP and DCCP: generalizing congestion control

The transport layer allows two or more entities to **directly communicate** without having to take into account the various network elements that are traversed:

- virtual association between **processes**
- **end-to-end** communications
  - abstract out the **topology** and the associated **technologies**
  - present on end-hosts
    - **sender**: breaks down application layer messages and sends them down to the network layer
    - **receiver**: reassembles network layer segments into messages and sends them up to the application layer

- 2 models define the functionalities associated with each layer...
Transport layer: Internet

Few standard transport layer protocols:
- unordered, unreliable transmissions: **UDP**
  - best effort service
  - lightweight
- ordered, reliable transmissions: **TCP**
  - flow oriented
  - congestion control
- message oriented reliable transmission: **SCTP**
- unordered, unreliable transmissions with congestion control: **DCCP**
- unavailable:
  - temporal guarantees (unbounded delays are unpredictable jitter)
  - bandwidth guarantees

Transport layer: 2 modes

**Connectionless mode**

- Sending host
- Network
- Receiving host
- Sending process
- Datagram
- Receiving process

**Connection oriented mode**

- Sending host
- Network
- Receiving host
- Sending process
- Virtual connection
- Receiving process

Transport layer: primitives

**Programming** interface (applications or developers)
- examples of connection-oriented mode primitives:
  - LISTEN
  - CONNECT
  - SEND
  - RECEIVE
  - DISCONNECT

pictures from Tanenbaum A. S. Computer Networks 3rd edition
Basic services
Reliable service
Congestion control
Transport layer review
Multiplexing and demultiplexing
UDP: a connectionless mode protocol

**Transport layer: call setup**

**ComNet: course 3/5 outline**

1. Basic services
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol

2. Reliable service
   - Principles of reliable data transfer
   - TCP: a connection oriented protocol
   - TCP: reliability mechanisms

3. Congestion control
   - General principles
   - TCP mechanisms
   - SCTP and DCCP

**Multiplexing/demultiplexing**

Application-level processes transmit their data to the system via sockets: multiplexing is the mixing of these data.

- **mux** (at the sender):
  - add a header to each block of data at a socket
  - collect data from several sockets

- **demux** (at the receiver):
  - provide the data to the correct socket

**Demultiplexing in connectionless mode**

Association of a socket with a port number
- identifying the DatagramSocket: (@IPdest, numPortDest)
- receiving a datagram at a host:
  - verify its numPortDest
  - send to the socket corresponding to numPortDest
  - ∀ @IPsource, ∀ numPortSource

---

O. Fourmaux - T. Friedman (olivier.fourmaux@upmc.fr)  
Computer Networks (ComNet) 3/5 : Transport

pictures from Tanenbaum A. S. Computer Networks 3rd edition
Multiplexing in connection oriented mode

A connection associating two processes
- a StreamSocket is identified by a four-tuple:
  - source address: @IPsource
  - source port: numPortSource
  - destination address: @IPdest
  - destination port: numPortDest
- receiving a segment at a host:
  - verification of the four-tuple
  - forwarding to the corresponding socket
  - there can be many simultaneous connections to a web server

Demultiplexing in connection oriented mode

Multi-threaded webserver (one socket per connection)

ComNet: course 3/5 outline

1. Basic services
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol
2. Reliable service
   - Principles of reliable data transfer
   - TCP: a connection oriented protocol
   - TCP: reliability mechanisms
3. Congestion control
   - General principles
   - TCP mechanisms
   - SCTP and DCCP

UDP: a connectionless mode protocol

User Datagram Protocol [RFC 768]
- basic, unembellished, Internet transport protocol
- best effort service:
  - datagrams in transit can...
    - be lost
    - be duplicated
    - arrive out of order
- unconnected service:
  - no initial exchange
  - no state information at the end-hosts
  - each datagram is handled independently

UDP datagram
- 32 bits (4 octets) Source port Destination port
- 2 bytes (8 octets) Datagram length Checksum
- Message (application data)

UDP: source port
- 16 bits (65535 ports)
- multiplexing at the source
- identify the socket for an eventual reply
- fixed allocation, or dynamic (generally for clients)
- division of the port number space:
  - $0 \leq \text{numPort} \leq 1023$: available to the administrator
  - server sockets (generally)
  - $1024 \leq \text{numPort} < 65536$: available to users
  - client sockets (generally)
Basic services
Reliable service
Congestion control
Transport layer review
Multiplexing and demultiplexing
UDP: a connectionless mode protocol

**UDP: destination port**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datagram length</td>
<td>Checksum</td>
</tr>
</tbody>
</table>

- 16 bits (65535 ports)
- **demultiplexing** at the destination
- the receiver must listen on this port
- negotiation of well-known ports (reserved ports):

```bash
Unix> cat /etc/services |grep udp
discard 9/udp
tftp 69/udp
daytime 13/udp
chargen 19/udp
snmp 161/udp
snmp-trap 162/udp
ssh 22/udp
...```

**UDP: checksum**

- 16 bits
- error control **optional**
- sender:
  - add a **pseudo-header**
  - checksum = \(\sum\text{word_{16bits}}\)
  - Binary sum over 16 bits, carrying the overflow to the least significant bit
- receiver:
  - add a **pseudo-header**
  - recalculate the \(\sum\text{word_{16bits}}\)
  - \(= 0\): no error detected still possible...
  - \(\neq 0\): error (silently destroyed)

**UDP: arguments for connectionless transport**

Motivation for choosing connectionless transport:
- limited resources at the end-hosts
  - limited TCP/IP stack
  - lack of **state** in the hosts
  - limited processing power
- need for a rapid exchange
  - no connection **establishment**
- need for efficiency
  - **smaller header**
- temporal constraints
  - **retransmissions** not appropriate
  - no sending bandwidth **control**
- need for new functionalities
  - handled by the application layer...
UDP: example applications

- classical applications:
  - name resolution (DNS)
  - network administration (SNMP)
  - routing protocol (RIP)
  - clock synchronization protocol (NTP)
  - remote filesystem (NFS)
    - implicit reliability through temporal redundancy
    - explicit reliability through application layer mechanisms

- multicast applications

- multimedia applications
  - multimedia transmissions, audio or video streaming
  - voice over IP
  - video conferencing
    - temporal constraints
    - loss tolerance

UDP: socket interface

```c
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
// domain : PF_INET for IPv4 Internet Protocols
// type : SOCK_DGRAM Supports datagrams (connectionless, unreliable msg of a fixed max length)
// protocol : UDP (/etc/protocols)

int bind(int s, struct sockaddr *my_addr, socklen_t addrlen);

int sendto(int s, const void *msg, size_t len, int flags,
            const struct sockaddr *to, socklen_t tolen);
// Send an outgoing datagram to a destination address

int recvfrom(int s, void *buf, size_t len, int flags,
             struct sockaddr *from, socklen_t *fromlen);
// Receive the next incoming datagram and record it source address

int close(int s);
```

ComNet: course 3/5 outline

1. Basic services
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol

2. Reliable service
   - Principles of reliable data transfer
     - TCP: a connection oriented protocol
     - TCP: reliability mechanisms

3. Congestion control
   - General principles
   - TCP mechanisms
   - SCTP and DCCP

Transport layer and reliability (1)

Multilayer problem: application, transport and link layer
Transport layer and reliability (2)

Real channels are usually unreliable

Transport layer and reliability (3)

Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (RDT)

Reliable data transfer (RDT)

We’ll incrementally develop the reliable data transfer protocol (RDT)

- consider only unidirectional data transfer
- control info flow on both directions
- use finite state machines (FSM) to specify sender, receiver:
## RDT 1.0

Reliable transfer over a reliable channel
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender and receiver:

```
sender
  rdt_send(data)
  packet = make_pkt(data)
  udt_send(packet)
```

```
receiver
  rdt_rcv(packet)
```

## RDT 2.0

Reliable transfer on a channel with bit errors
- underlying channel may flip bits in packet
  - checksum to detect bit errors:
    - checksum: redundancy added in the packet
  - how to recover from errors?
    - acknowledgements (ACKs): receiver explicitly tells sender that packet received OK
    - negative acknowledgements (NAKs): receiver explicitly tells sender that packet had errors
      - sender retransmits packet on receipt of NAK
- new mechanisms in RDT 2.0:
  - error detection
    - notcorrupt(pkt): true if the pkt checksum is correct
    - corrupt(pkt): true if the pkt is not correct
  - receiver feedback (ACK and NAK control messages)

```
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```

```
ratio(rcvpkt) && isACK(rcvpkt)
```

```
ratio(rcvpkt) && isNAK(rcvpkt)
```

```
ratio(rcvpkt) & isACK(rcvpkt)
```

```
ratio(rcvpkt) & corrupt(rcvpkt)
```

```
ratio(rcvpkt) & notcorrupt(rcvpkt)
```

```
ratio(rcvpkt) & isACK(rcvpkt)
```

```
ratio(rcvpkt) & corrupt(rcvpkt)
```

```
ratio(rcvpkt) & notcorrupt(rcvpkt)
```
Basic services
Reliable service
Congestion control

Principles of reliable data transfer
TCP: a connection oriented protocol
TCP: reliability mechanisms

RDT 2.0: NAK

Reliable transfer during error scenario:

- `rdt_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `wait for call from above`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `wait for call 0 from above`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(sndpkt)`
- `wait for call 1 from above`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `sndpkt = make_pkt(NAK, checksum)`
- `udt_send(sndpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK, checksum)`
- `udt_send(sndpkt)`

RDT v2.0: discussion

RDT 2.0 is a "stop and wait" protocol:
- sender sends one packet, then waits for receiver response
- poor performance

RDT 2.0 has a fatal flaw!
- What happens if ACK/NAK corrupted?
  - sender doesn’t know what happened at receiver!
  - retransmission alone not sufficient: possible duplicates
- Handling duplicates:
  - sender retransmits current packet if ACK/NAK garbled
  - sender adds sequence number to each packet
  - receiver discards duplicate packet
- include in RDT 2.1

RDT 2.1: sender

- `rdt_send(data)`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `wait for call 0 from above`
- `rdt_rcv(rcvpkt) && (isNAK(rcvpkt) || isACK(rcvpkt))`  
  - `udt_send(sndpkt)`
- `wait for call 1 from above`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
  - `sndpkt = make_pkt(ACK, checksum)`
  - `udt_send(sndpkt)`

RDT 2.1: receiver

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK, checksum)`
  - `udt_send(sndpkt)`
- `wait for call 0 from above`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
  - `sndpkt = make_pkt(ACK, checksum)`
  - `udt_send(sndpkt)`
- `wait for call 1 from above`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
  - `sndpkt = make_pkt(ACK, checksum)`
  - `udt_send(sndpkt)`
**RDT 2.1: discussion**

End host behavior with RDT 2.1

- **Sender**
  - sequence number added to packet
  - two numbers will suffice (0 and 1)
  - must check if received ACK/NAK corrupted
  - twice as many states
- **Receiver**
  - must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected packet seqnum

*Could we remove the NAK?*

- instead of NAK, receiver sends ACK for last packet received OK
  - receiver must explicitly include seqnum of packet being ACKed
  - duplicate ACK at sender results in same action as NAK (retransmit current packet)
  - included in RDT 2.2

**RDT 2.2: sender fragment**

```plaintext
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
```

**RDT 2.2: receiver fragment**

```plaintext
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**

Reliable transmission on a channel with errors and losses

- underlying channel can also lose packets (data or ACKs)
  - checksum + seqnum + ACK + retransmission
  - not sufficient: a missing packet will stop the FSM!

**Retransmission Timeout**

- estimating a reasonable time to wait for an ACK
  - sender waits "reasonable" amount of time for ACK
    - start_timer
    - ACK before timeout ➔ nothing
    - stop_timer
    - no ACK before timeout ➔ retransmission
    - timeout
  - if ACK is only late...
    - retransmission = duplication
    - use of seqnum already handles this
RDT 3.0: sender

```
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

Wait for ACK0

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer

rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

Wait for ACK1

rdt_rcv(rcvpkt)
```

RDT 3.0: no loss

```
sender
send pkt0
rcv pkt0
send ACK0
rcv ACK0
send pkt1
rcv pkt1
send ACK1
rcvACK1
send pkt0
rcv pkt0
send ACK0
```

RDT 3.0: lost packet

```
sender
send pkt0
rcv pkt0
send ACK0
rcv ACK0
send pkt1
send ACK1
rcvACK1
send pkt0
```

RDT 3.0: lost ACK

```
sender
send pkt0
rcv pkt0
send ACK0
rcv ACK0
send pkt1
rcv pkt1
send ACK1
```

O. Fourmaux - T. Friedman (olivier.fourmaux@upmc.fr)
Computer Networks (ComNet) 3/5 : Transport

O. Fourmaux - T. Friedman (olivier.fourmaux@upmc.fr)
Computer Networks (ComNet) 3/5 : Transport

O. Fourmaux - T. Friedman (olivier.fourmaux@upmc.fr)
Computer Networks (ComNet) 3/5 : Transport

O. Fourmaux - T. Friedman (olivier.fourmaux@upmc.fr)
Computer Networks (ComNet) 3/5 : Transport
### RDT 3.0: premature timeout

- **sender**
  - send pkt0
- **receiver**
  - rcv pkt0
  - send ACK0
- **timeout**
  - resend pkt1
  - rcv ACK1
  - send pkt0
  - rcv pkt1
  - send ACK1

### RDT 3.0: performance

RDT 3.0 works, but poor performance...

- **transmission example:**
  - link transmission rate: $R = 1$ Gbps,
  - end to end propagation delay: $d = 40$ ms ($RTT = 80$ ms)
  - packet length: $L = 1000$ B, $B = 8000$ b
  - $T_{\text{transmit}} = L/R = 8.10^3/10^9 = 8 \mu s$

- **Usage ratio ($U_{\text{sender}}$):**
  - $U_{\text{sender}} = L/R + L/R = 8.10^{-6} + 8.10^{-6} = \frac{1}{10000}$

- **$R_{\text{average}}$:**
  - $L/RTT = 8.10^7/8.10 = 100$ Kbps (over 1 Gbps link)
  - network protocol limits use of physical resources!

---

### Pipelined protocols

- sender allows multiple ("in-flight"), yet-to-be-acknowledged packets
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver
  - Two generic forms of pipelined protocol: **Go-Back-N** and **Selective repeat**
Pipelining performance

Sender with Go-Back-N management
- packet headers with k bit sequence numbers
- cumulative acknowledgements
  - ACK(n) acknowledges all packets up to sequence number n
- window of at most N unacknowledged packets:
  - send_base
  - nextseqnum
  - send ACK
- timer for packets still (in-flight)
  - timeout(n): retransmission of packet n and all those with higher sequence numbers

Receiver with Go-Back-N management.
- ACKs only:
  - always send an ACK with the highest seqnum of valid in-order packets received
  - can generate duplicate ACKs
  - only expectedseqnum is stored
- out-of-order packets:
  - discard out-of-order packets
  - no receiver-side buffer
  - resend ACK with the highest seqnum of valid in-order packets received
Basic services
Reliable service
Congestion control
Principles of reliable data transfer
TCP: a connection oriented protocol
TCP: reliability mechanisms

RDT 4.0: receiver

Sender with selective repeat management (receiver individually acknowledges all correctly received packets)
- sender only resends packets for which ACK not received
- sender window limited to $N$ consecutive seqnums
- algorithm:
  - `rdt_send(data)`
  - if next available seqnum in window, send packet
  - `timeout(n)`
  - resend packet $n$
  - `init_timer(n)`
  - ACK($n$) in $[send_base,send_base+N]$:
    - mark packet $n$ as received
    - if $n$ smallest unACKed packet, advance window base to next unACKed seqnum

RDT 4.0: example

Receiver with selective repeat management:
- receiver individually acknowledges all correctly received packets
- out-of-order: buffer
- algorithm:
  - `rdt_rcv(n)` with $n$ in $[send_base,send_base+N-1]$
    - ACK($n$)
    - out-of-order: buffer
    - in-order: `send_app(data)`, advance window to next not-yet-received packet
  - `rdt_rcv(n)` with $n$ in $[send_base-N,send_base-1]$
    - ACK($n$)
  - otherwise
    - ignore
Selective repeat: overview

(a) sender view of sequence numbers

send_base
nextseqnum
window size
N
already ack’ed
sent, not yet ack’ed
usable, not yet sent
not usable

(b) receiver view of sequence numbers

window size
N
cov_base

Selective repeat: example

TCP

SYN+ACK
ACK
DATA

ComNet: course 3/5 outline

1. Basic services
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol

2. Reliable service
   - Principles of reliable data transfer
   - TCP: a connection oriented protocol
   - TCP: reliability mechanisms

3. Congestion control
   - General principles
   - TCP mechanisms
   - SCTP and DCCP
TCP (Transmission Control Protocol)

- **Basic services**
  - Reliable service
  - Congestion control

- **Principles of reliable data transfer**
  - TCP: a connection oriented protocol
  - TCP: reliability mechanisms

TCP (Transmission Control Protocol)

[RFCs: 793, 1122-1123, 2474, 3168, 3260, 4379, 5462, and 5681]

- **reliable** service
- **point-to-point**
  - two processes (generally a client and a server)
  - no message boundaries
- **connection oriented**
  - three-way handshake for opening connections
  - state created at the end-points prior to the data exchange
  - graceful or abrupt connection closure
- **bidirectional** (full duplex) connection
  - data streamed in both directions
  - MSS (Maximum Segment Size)
- **pipeline**
  - send and receive buffers
  - windows for flow and connection control

TCP: a connection oriented protocol

TCP: reliability mechanisms

**TCP: source port**

- 16 bits (65,535 ports)
- Multiplexing at the source
- Partial socket identification (local half-association)
- Generally dynamically allocated (clients)
- Port number space allocation:
  - \(0 \leq \text{numPort} \leq 1023\):
    - available to the administrator
    - typical server sockets
  - \(1024 \leq \text{numPort}\):
    - available to users
    - typical client sockets

**TCP: destination port**

- 16 bits (65,535 ports)
- Demultiplexing at the destination
- Partial identification of the socket (remote half-association)
- Destination listens on the port upon its creation
- Port negotiation or well-known (reserved):

```
Unix> cat /etc/services|grep tcp
tcpmux 1/tcp
ftp-data 20/tcp
ftp 21/tcp
ssh 22/tcp...
```

- **O. Fourmaux - T. Friedman (olivier.fourmaux@upmc.fr)**
  - Computer Networks (ComNet) 3/5 : Transport
**TCP: sequence number (1)**

- 32 bits
- associated with each byte (and not a segment)
  - number the first byte of data
  - implicit numbering of the following bytes
  - loop after 4 GB
- loss detection
- ordering

**TCP: acknowledgement number (1)**

- 32 bits
- piggybacking
- indicates the number of the next byte expected
- cumulative, indicating the first non-received byte (other, higher sequence number, bytes might have been received)
TCP: acknowledgement number (3)

**Piggybacking**

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq=4000 Ack=11200</td>
<td></td>
</tr>
<tr>
<td>ACK 200</td>
<td>Seq=11200 Ack=4200</td>
</tr>
<tr>
<td>ACK 200</td>
<td>Seq=11300 Ack=4400</td>
</tr>
<tr>
<td>ACK 100</td>
<td>Seq=11400 Ack=4600</td>
</tr>
</tbody>
</table>

TCP: TELNET example (1)

**Sending a key stroke and server reply for display**

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq=80 Ack=210</td>
<td></td>
</tr>
<tr>
<td>the user type &quot;a&quot;</td>
<td>ACK data=&quot;a&quot;</td>
</tr>
<tr>
<td>Seq=81 Ack=211</td>
<td>Seq=210 Ack=81</td>
</tr>
<tr>
<td>the server receive &quot;a&quot;</td>
<td>ACK</td>
</tr>
<tr>
<td>Seq=81 Ack=211</td>
<td>Seq=211 Ack=281</td>
</tr>
<tr>
<td>... and return &quot;a&quot;</td>
<td></td>
</tr>
</tbody>
</table>

TCP: TELNET example (2)

**ACKs can be more rapid than the application**

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq=80 Ack=210</td>
<td>Seq=21000 Ack=677</td>
</tr>
<tr>
<td>the user type &quot;a&quot;</td>
<td></td>
</tr>
<tr>
<td>Seq=81 Ack=211</td>
<td>Seq=22000 Ack=677</td>
</tr>
<tr>
<td>Seq=210 Ack=81</td>
<td>Seq=23000 Ack=677</td>
</tr>
<tr>
<td>the server receive &quot;a&quot;</td>
<td>ACK</td>
</tr>
<tr>
<td>Seq=210 Ack=81</td>
<td>Seq=24000 Ack=677</td>
</tr>
<tr>
<td>... and return &quot;a&quot;</td>
<td></td>
</tr>
<tr>
<td>Seq=24000 Ack=677</td>
<td></td>
</tr>
</tbody>
</table>

TCP: delayed ACK

**waiting for two segments or max 500 ms**

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq=667 Ack=2600</td>
<td>Seq=667 Ack=2200</td>
</tr>
<tr>
<td></td>
<td>ACK 1000 octets</td>
</tr>
<tr>
<td></td>
<td>Seq=667 Ack=24000</td>
</tr>
<tr>
<td></td>
<td>ACK 1000 octets</td>
</tr>
<tr>
<td></td>
<td>Seq=667 Ack=26000</td>
</tr>
</tbody>
</table>
TCP: header length

- 4 bits (15 possible values)
- number of 32-bit words in the TCP header
- necessary because the option field is of variable length
  - value of 5...
    - no options
    - minimum TCP header length: 20 bytes
  - ... to 15
    - 10 words of options
    - maximum 40 bytes of options
    - maximum TCP header length: 60 bytes

TCP: receiver window size

- 16 bits
  - the receiver can announce up to 64 KB
  - piggybacking
  - flow control
    - indicates the number of bytes available in the receiver buffer
    - dimension the size of the sender’s sliding window

TCP: segment checksum

- 16 bits
- same as for UDP
- sender:
  - ajout pseudo-header
  - checksum = \sum_{word}^{16bits}
- receiver:
  - adds pseudo-header
  - recalculates \sum_{word}^{16bits}
    - = 0: Ok
    - \neq 0: destruction
TCP: urgent pointer

- 16 bits
- for sending special data (not out-of-band)
- designates data for priority treatment
- points to the end of the urgent data
  - application-specific interpretation of these data and their role

TCP: options

Options are in Type, Length (bytes), Value format:

- END: end of the option list (T=0, not required)
- NOOP: no operation (T=1, padding)
- MSS: MSS negotiation (T=2, L=4, V=MSS)
- WSIZE: scale window by a factor of $2^N$ (T=3, L=3, V=N)
- SACK: request selective acknowledgement (T=4, L=2, upon open)
- SACK: selective acknowledgement of $n$ blocks (T=5, L=2 + 8n, 2n sequence numbers) ...

TCP: three-way handshake (1)

Opening exchange, with three segments

1. client → server: TCP segment with the SYN flag
   - indicates the client’s initial sequence number (ISN)
   - sending the SYN increments the future sequence number
   - no data

2. server → client: TCP segment with the SYN + ACK flags
   - receipt of a SYN has incremented the acknowledgement number
   - indicates the server’s initial sequence number (ISN)
   - sending the SYN increments the future sequence number
   - server buffer allocated

3. client → server: TCP segment with the ACK flag
   - receipt of a SYN has incremented the acknowledgement number
   - may contain data

Opening the connection prior to data exchange:

- initialize TCP parameters
  - synchronize sequence numbers
  - allocate buffers
  - initialize flow control
- client: initiates the connection
- server: waits for a connection request

Closing the connection following the data exchange:

- wait, or not, for remaining data to be sent
- free the buffers
TCP: three-way handshake (2)

Initial exchange based on three segments

Host A

\[ S=3000 \]

SYN

\[ S=3001 \ A=7001 \]

SYN + ACK

\[ 200 \]

ACK

\[ 300 \]

ACK

\[ 100 \]

Host B

\[ S=7000 \ A=3001 \]

ACK

\[ S=7001 \ A=3201 \]

ACK

\[ S=7301 \ A=3201 \]

ACK

TCP: three-way handshake (3)

Handling simultaneous connection opening

(a) (b)

TCP: graceful release (1)

1. The client sends a TCP segment with \textit{FIN}
   - sending the FIN increments the future sequence number
   - may contain data
2. The server receives the FIN segment
   - receipt of the FIN increments the sequence number
   - sends a TCP \textit{ACK} segment
   - closes the connection (\textit{sends remaining data})
   - sends a TCP \textit{FIN} segment
   - sending the FIN increments the future sequence number
3. The client receives the FIN segment
   - receipt of the FIN increments the sequence number
   - sends a TCP \textit{ACK} segment
   - closes the connection
     - set a timer (duplicate FINs)
4. The server receives the FIN segment

TCP: graceful release (2)

Disconnection: graceful release

Host A

\[
S=4000 \ A=11200
\]

\[
S=4200 \ A=11200
\]

release

\[
S=4301 \ A=11700
\]

\[
S=4301 \ A=12200
\]

\[
S=4301 \ A=12701
\]

Host B

\[
S=11200 \ A=4301
\]

\[
S=11700 \ A=4301
\]

\[
S=12200 \ A=4301
\]

release

\[
S=12700 \ A=4301
\]

closed
TCP: shutdown

Disconnection: unilateral close (for all abnormal or undesired circumstances)

- **Case 1**
  - Host A: S=4000, A=4000
  - Host B: S=0
  - Host A sends SYN and RST
  - Host B sends ACK + RST

- **Case 2**
  - Host A: S=2200, A=7300
  - Host B: S=7300
  - Host A sends ACK
  - Host B sends RST

TCP: finite state machine

TCP is a protocol for reliable transfer over the unreliable IP service

- base mechanisms:
  - **pipeline**
  - **cumulative** ACKs
  - **single** retransmission timer
  - retransmissions triggered by:
    - timeouts
    - duplicate ACKs
  - furthermore...
    - simplified TCP sender:
      - no duplicate ACKs
      - no flow control
      - no congestion control

ComNet: course 3/5 outline

1. **Basic services**
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol
2. **Reliable service**
   - Principles of reliable data transfer
   - TCP: a connection oriented protocol
   - TCP: reliability mechanisms
3. **Congestion control**
   - General principles
   - TCP mechanisms
   - SCTP and DCCP

O. Fourmaux - T. Friedman (olivier.fourmaux@upmc.fr)  Computer Networks (ComNet) 3/5 : Transport
TCP: calculating RTT

RTT = Round Trip Time

- Estimating the retransmission timeout:
  - greater than the RTT... but RTT varies!
  - too small: unnecessary retransmissions
  - too large: slow reaction to losses

- Estimating the RTT:
  - $RTT_{measured} = \Delta$ (sending of a segment to reception of the ACK)
  - $RTT_{measured}$ can vary rapidly → smoothing
  - $RTT = \alpha RTT_{measured} + (1 - \alpha)RTT_{old}$
    with typical $\alpha = 1/8$
  - exponentially weighted moving average

TCP: example of RTT calculation

TCP: Timers

Managing multiple timers:

- retransmission timer (detects losses)
  - $RTO = RTT + 6D$
  - $D = \beta (|RTT_{measured} - RTT_{old}|) + (1 - \beta)D_{old}$
  - mean deviation calculation with typical $\beta = 1/4$

- Karn’s algorithm
  - do not take into account retransmitted packets and double the $RTO$ with each failure (exponential backoff)

- persistence timer (avoid blockage)
  - send an acknowledgement with a window of 0

- keep alive timer (verify that the other end-host is still there)

- closing timer

TCP: sender events

- data received from the layer above
  - creation of a segment with numSeq
    - numSeq is the number, in the data stream, of the segment’s first byte
  - start the timer if it is not already set
    - the timer is for the oldest non-acknowledged segment

- timeout
  - retransmit the segment associated with the timer
  - restart the timer

- acknowledgement received (ACK)
  - if it acknowledges as-yet unacknowledged segments:
    - update the base of the transmission window (base_emis)
    - restart the timer if waiting on other ACKs
### TCP: retransmission (1)

**Scenario with a lost ACK**

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>S=4000 A=1100</td>
<td>S=1100 A=4200</td>
</tr>
<tr>
<td>ACK 200</td>
<td>ACK</td>
</tr>
<tr>
<td>retrans. S=4000 A=1100</td>
<td></td>
</tr>
<tr>
<td>ACK 200</td>
<td></td>
</tr>
<tr>
<td>S=4200 A=1100</td>
<td>S=1100 A=4200</td>
</tr>
<tr>
<td>ACK 50</td>
<td></td>
</tr>
</tbody>
</table>

### TCP: retransmission (2)

**Scenario with an under-estimated timer**

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>S=4000 A=1100</td>
<td>S=1100 A=4200</td>
</tr>
<tr>
<td>ACK 200</td>
<td>ACK 200</td>
</tr>
<tr>
<td>S=4000 A=1100</td>
<td>S=1100 A=4400</td>
</tr>
<tr>
<td>ACK</td>
<td>ACK</td>
</tr>
</tbody>
</table>

### TCP: retransmission (3)

**Scenario with cumulative ACKs**

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>S=4000 A=1100</td>
<td>S=1100 A=4200</td>
</tr>
<tr>
<td>ACK 200</td>
<td>ACK</td>
</tr>
<tr>
<td>S=4200 A=1100</td>
<td>S=1100 A=4200</td>
</tr>
<tr>
<td>ACK 200</td>
<td></td>
</tr>
<tr>
<td>S=4400 A=1100</td>
<td>S=1100 A=4400</td>
</tr>
<tr>
<td>ACK 200</td>
<td></td>
</tr>
</tbody>
</table>

### TCP: receiver events

**Generating ACKs (receiver actions)**

- arrival of an in-order segment with the expected seqnum:
  - the prior segments have already been acknowledged
  - delayed ACK, wait up to 500 ms
  - if there are no other segments, send an ACK
  - another segment is waiting to be acknowledged
  - immediately send a cumulative ACK for these two in-order segments
- arrival of an out-of-order segment:
  - seqnum above what was expected (gap detected)
  - immediately send a duplicate ACK
  - reminder of the next expected seqnum
  - partly or wholly fills a gap
  - immediately send an ACK
  - new expected seqnum following the filling of the gap
TCP: fast retransmit (1)

Optimizing the retransmission mechanism

- timers are often set relatively high
  - long delays before retransmission
- detection of lost segments due to duplicate ACKs
  - segments typically arrive in groups
  - if a segment is lost ⇒ many duplicate ACKs
- if the sender receives 3 duplicate ACKs (4 identical ACKs)
  - TCP assumes that the segment following one being acknowledged has been lost
  - fast retransmit: retransmit the segment prior to timer expiration

TCP: receiver control

- flow control
  - the sender should not overflow the receiver's buffer
  - learning the available space in the receiver's buffer:
    - \( \text{RwndSize} = \text{BufferSize} - \text{LastByteReceived} + \text{LastByteRead} \)

TCP: sender limitation

Sliding window: the sender limits its sending of unacknowledged data
Basic services
Reliable service
Congestion control
Principles of reliable data transfer
TCP: a connection-oriented protocol
TCP: reliability mechanisms

TCP: flow control

TCP: flow control optimisation

Send-side silly window syndrome
- Nagle’s Algorithm (RFC 896)
  - aggregation of small packets (nagling)
  - waiting for an ACK or an MSS before sending a segment
    - TELNET: avoid sending one packet per typed character
    - can be deactivated with the TCP_NODELAY socket option

Receiver silly window syndrome
- Clark’s Algorithm
  - limit announcements of small windows
    - close the window while waiting for there to be sufficient space for an entire segment

TCP: re-opening receiver window timer

Persistence timer
- avoids having the window size remain at 0
  - possible if an ACK announcing a non-zero window is lost
  - avoided by sending a probe packet after a timer expiry
  - timer initialized to RTT, doubles with each expiry, up to 60 s (then stays at 60 s)
  - the probe packet contains one byte of data

TCP: usage examples

The following applications are typically based on TCP:
- remote login (TELNET, rlogin, and ssh)
- file transfer (FTP, rcp, scp, and sftp)
- inter-domain routing (BGP)
- instant messaging (IRC, ICQ, AIM, ...)
- web (HTTP)
  - new HTTP-based applications such as network access (allowing one to traverse firewalls)
TCP doit s’adapter à des flots de qqs bps à plusieurs Gbps:

- **LFN (Long Fat Network)**
  - network capacity = bandwidth * propagation delay
  - limited window size (FSIZE option, up to a factor of $2^{14}$)
  - sequence number wrapping (PAWS, Protect Against Wrapped Sequence, using the TIMESTAMP option)
  - selective ACKs to avoid too many unnecessary retransmissions (SACK option)
- satellites
- transoceanic fiber
- asymmetric networks (ADSL, cable)
- underutilization of the high capacity link

### TCP: special cases

```c
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
# domain : PF_INET for IPv4 Internet Protocols
# type : SOCK_STREAM Provides sequenced, reliable, 2-way, connection-based byte streams.
# protocol : TCP (/etc/protocols)

int bind(int s, struct sockaddr *my_addr, socklen_t addrlen);
# Server : passive queuing mode and connection acceptance
int listen(int s, int backlog);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
# Client : active connection
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);

int send(int s, const void *msg, size_t len, int flags);
int recv(int s, void *buf, size_t len, int flags);

int close(int s);
```

---

**ComNet: course 3/5 outline**

1. Basic services
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol
2. Reliable service
   - Principles of reliable data transfer
   - TCP: a connection oriented protocol
   - TCP: reliability mechanisms
3. Congestion control
   - General principles
   - TCP mechanisms
   - SCTP and DCCP
Congestion: scenario 1a

- 2 senders, 2 receivers
- 1 router
- infinite buffers
- no retransmissions

What happens as \( d_\text{in} \) grows?

Congestion: scenario 1b

- the cost of congestion:
  - maximum possible bandwidth
    - \( d_\text{in} = C/2 \)
    - high delay, close to the maximum
      - infinite buffer growth

Congestion: scenario 2a

- 2 senders, 2 receivers
- 1 router
  - finite buffers
  - retransmission of lost segments

What happens as \( d'_\text{in} \) grows?

Congestion: scenario 2b

- \( d_\text{in} = d_\text{out} \) (goodput)
- retransmission cost
  - useful retransmissions: losses only
    - \( d'_\text{in} \) greater than \( d_\text{out} \)
  - useless retransmissions: late segments
    - \( d'_\text{in} \) even greater than \( d_\text{out} \)

congestion cost:
  - much more traffic for a given \( d_\text{out} \)
  - duplications of useless segments
**Basic services**

- Reliable service
- Congestion control

**General principles**

- TCP mechanisms
- SCTP and DCCP

**Congestion control solutions**

- Two approaches:
  - congestion control by the network
    - routers inform the end-hosts
      - congestion notification bits (SNA, DECbit, ATM, TCP/IP ECN...)
      - explicit signalling of available bandwidth (ATM ABR, TCP/IP RSVP + IntServ...)
  - congestion control at the end-systems (end-to-end)
    - no explicit signalling from the network
    - inference based upon observations at the end-systems
      - losses
      - delays
    - approach taken by TCP

**ComNet**: course 3/5 outline

1. **Basic services**
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol

2. **Reliable service**
   - Principles of reliable data transfer
   - TCP: a connection oriented protocol
   - TCP: reliability mechanisms

3. **Congestion control**
   - General principles
   - TCP mechanisms
   - SCTP and DCCP

---

**Congestion: scenario 3a**

- 4 senders, 4 receivers
- 4 routers
- multi-hop paths
- finite buffers
- retransmission

What happens as $d'_\text{in}$ grows?

**Congestion: scenario 3b**

- additional costs associated with congestion:
  - when a packet is lost, all upstream capacity is wasted...
TCP: AIMD algorithms

AIMD = Additive Increase, Multiplicative Decrease

- steady increase in sender bandwidth (cwnd) as long there are no losses
  - Additive Increase
    - increase cwnd by 1 MSS with each RTT as long as no losses have been detected
  - Multiplicative Decrease
    - divide cwnd by 2 following a loss
    - sawtooth behaviour

TCP: congestion control

- based on sender-side limitation
  - lastByteSent - lastByteAcked ≤ cwnd
  - approximate bandwidth:
    - \( d_{TCP} = \frac{cwnd}{RTT} \)
- cwnd = varies dynamically with detected congestion
  - congestion detection by the receiver:
    - timeout (RTO)
    - triple duplicate ACK
  - 3 mechanisms:
    - AIMD
    - Slow Start
    - caution following timer expiry

TCP: slow start

Slow start \( \to \) in fact grows rapidly!

- at the start of a connection
  - cwnd = 2 to 4 MSS
  - when restarting (after a loss or inactivity)
    - cwnd = 1 MSS (\( d_{init} = \frac{MSS}{RTT} \))
    - then exponential growth until the first loss
      - double cwnd / RTT
      - implemented by:
        - cwnd ++ / ACK
      - \( d_{potential} \gg \frac{MSS}{RTT} \)

TCP: optimisation

From exponential to linear growth

- cwnd ≥ old value of cwnd prior to the loss
  - implemented with a varying limit:
    - ssthresh = cwnd prior to loss \( / 2 \)
  - more precisely calculated over unacknowledged segments:
    - ssthresh = flightsize/2
TCP: inferring loss

Duplicate ACKs are not as bad as timeouts

- **3 duplicate ACKs:**
  - indicate that the network continues to transmit segments
    - cwnd divided by 2
    - cwnd then grows linearly
- **a timeout:**
  - indicates that the network is blocked
    - cwnd = 1 MSS
    - *Slow Start* (exponential growth)
    - to ssthresh = cwnd/2 (linear growth)

TCP congestion control: synthesis

RFC 5681

- when cwnd < ssthresh:
  - sender in the *Slow Start* phase
  - cwnd grows exponentially
- when cwnd ≥ ssthresh:
  - sender is in the *Congestion Avoidance* phase
  - cwnd grows linearly
- when there are 3 duplicate ACKs:
  - ssthresh = last cwnd / 2
  - cwnd = ssthresh
- when there is a timeout:
  - ssthresh = last cwnd / 2
  - cwnd = 1 MSS

TCP: fairness among flows

- oscillation of two congested flows

A trip to Nevada:

- **TCP Tahoe 1988**
  - *slow start + congestion avoidance + multiplicative decrease*
  - **fast retransmit** (retransmit a segment after 3 duplicate ACKs, before timeout)
  - as just described...problem when just 1 seg. lost
- **TCP Reno 1990 (RFC 2582)**
  - like TCP Tahoe, with...
  - **fast recovery** (no *slow start* after a *fast retransmit*)
- **TCP newReno 1996 (RFC 3782)**
  - like TCP Reno, with...
  - no *slow start* when first congested, with cwnd adjustment
  - **SACK** (RFC 2018)
ComNet: course 3/5 outline

1. Basic services
   - Transport layer review
   - Multiplexing and demultiplexing
   - UDP: a connectionless mode protocol

2. Reliable service
   - Principles of reliable data transfer
   - TCP: a connection oriented protocol
   - TCP: reliability mechanisms

3. Congestion control
   - General principles
   - TCP mechanisms
   - SCTP and DCCP

SCTP

Stream Control Transmission Protocol (RFC 4960)

DCCP

Datagram Congestion Control Protocol (RFC 4340)