Computer Networking: Principles, Protocols and Practice

Part 3: Transport Layer

Olivier Bonaventure
http://inl.info.ucl.ac.be/
Module 3 : Transport Layer

Basics

Building a reliable transport layer
  Reliable data transmission
  Connection establishment
  Connection release

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol
The transport layer

Goals
Improves the service provided by the network layer to allow it to be useable by applications
reliability
multiplexing

Transport layer services
Unreliable connectionless service
Reliable connection-oriented service
The network layer

Network layer service in Internet
Unreliable connectionless service
- Packets can be lost
- Packets can suffer from transmission errors
- Packet ordering is not preserved
- Packet can be duplicated
- Packet size is limited to about 64 KBytes

How to build a service useable by applications?
The transport layer

Problems to be solved by transport layer

Transport layer must allow two applications to exchange information. This requires a method to identify the applications.

The transport layer service must be useable by applications:
- detection of transmission errors
- correction of transmission errors
- recovery from packet losses and packet duplications
- different types of services:
  - connectionless
  - connection-oriented
  - request-response
Internal organisation
The transport layer uses the service provided by the network layer
Two transport layer entities exchange segments

The transport layer (2)
Module 3 : Transport layer

Basics

Building a reliable transport layer
  Reliable data transmission
  Connection establishment
  Connection release

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol
Transport layer protocols

How can we provide a reliable service in the transport layer

Hypotheses
1. The application sends small SDUs
2. The network layer provides a perfect service
   1. There are no transmission errors inside the packets
   2. No packet is lost
   3. There is no packet reordering
   4. There are no duplications of packets
3. Data transmission is unidirectional
Transport layer protocols (2)

Reference environment

Notations

`data.req` and `data.ind` primitives for application/transport interactions

`recv()` and `send()` for interactions between transport entity and network layer
Protocol 1 : Basics

Principle
Upon reception of \texttt{data.request(SDU)}, the transport entity sends a segment containing this SDU through the network layer (\texttt{send(SDU)})
Upon reception of the contents of one packet from the network layer (\texttt{recv(SDU)}), transport entity delivers the SDU found in the packet to its user by using \texttt{data.ind(SDU)}
Protocol 1 : Basics

Principle
Upon reception of \texttt{data.request(SDU)}, the transport entity sends a segment containing this SDU through the network layer (\texttt{send(SDU)})

Upon reception of the contents of one packet from the network layer (\texttt{recv(SDU)}), transport entity delivers the SDU found in the packet to its user by using \texttt{data.ind(SDU)}
Protocol 1 : Basics

Principle
Upon reception of `data.request(SDU)` , the transport entity sends a segment containing this SDU through the network layer (`send(SDU)`).
Upon reception of the contents of one packet from the network layer (`recv(SDU)`), transport entity delivers the SDU found in the packet to its user by using `data.ind(SDU)`.
Protocol 1: Basics

Principle
Upon reception of data.request(SDU), the transport entity sends a segment containing this SDU through the network layer (send(SDU)).
Upon reception of the contents of one packet from the network layer (recv(SDU)), transport entity delivers the SDU found in the packet to its user by using data.ind(SDU).
Protocol 1 as a FSM

Sender

Wait for SDU

Receiver

Wait for segment

Data.req(SDU)

send(SDU)

recv(SDU)

Data.ind(SDU)
Protocol 1: Example

Issue
What happens if the receiver is much slower than the sender?
  e.g. receiver can process one segment per second while sender is producing 10 segments per second?
Protocol 1: Example

Issue
What happens if the receiver is much slower than the sender?
e.g. receiver can process one segment per second while sender is producing 10 segments per second?
Protocol 1 : Example

Issue
What happens if the receiver is much slower than the sender?

- e.g. receiver can process one segment per second while sender is producing 10 segments per second?
Protocol 1 : Example

Issue

What happens if the receiver is much slower than the sender?
e.g. receiver can process one segment per second while sender is producing 10 segments per second?
Protocol 2

Principle
Use a control segment (OK) that is sent by the receiver after having processed the received segment creates a feedback loop between sender and receiver.

Consequences
Two types of segments
- Data segment containing on SDU
  Notation: D(SDU)
- Control segment
  Notation: C(OK)

Segment format
At least one bit in the segment header is used to indicate the type of segment.
Protocol 2 (cont.)

Sender

Data.req(SDU) \[\xrightarrow{\text{send}}\] D(SDU)

Receiver

Recvd(C(OK))

Data.ind(SDU)

Send(C(OK))
The sender only sends segments when authorised by the receiver
Protocol 2 : Example

The sender only sends segments when authorised by the receiver
Protocol 2: Example

The sender only sends segments when authorised by the receiver.
The sender only sends segments when authorised by the receiver.
Protocol 2: Example

The sender only sends segments when authorised by the receiver.
Protocol 2: Example

The sender only sends segments when authorised by the receiver.
Protocol 3

How can we provide a reliable service in the transport layer

Hypotheses
1. The application sends small SDUs
2. The network layer provides a perfect service
   1. Transmission errors are possible
   2. No packet is lost
   3. There is no packet reordering
   4. There are no duplications of packets
3. Data transmission is unidirectional
Transmission errors

Which types of transmission errors do we need to consider in the transport layer?

Physical-layer transmission errors caused by nature

- **Random isolated error**
  - one bit is flipped in the segment

- **Random burst error**
  - a group of $n$ bits inside the segment is errored
  - most of the bits in the group are flipped
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors.
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors.
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors.
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors.
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors.
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors.

These attacks are dealt by using special security protocols and mechanisms outside the transport layer.
How to detect transmission errors?

**Principle**

**Type** + **Control**

Sender adds some control information inside the segment. Control information is computed over the entire segment and placed in the segment header or trailer.

Receiver checks that the received control information is correct by recomputing it.
Parity bits

Simple solution to detect transmission errors

Used on slow-speed serial lines
e.g. modems connected to the telephone network

Odd Parity
For each group of n bits, sender computes the n+1th bit so that the n+1 group contains an odd number of bits set to 1

Examples

00110100

11011100

Even Parity
Parity bits

Simple solution to detect transmission errors

Used on slow-speed serial lines
e.g. modems connected to the telephone network

Odd Parity
For each group of n bits, sender computes the n+1th bit so that the n+1 group contains an odd number of bits set to 1

Examples

0011010 0 1101100

Even Parity
Parity bits

Simple solution to detect transmission errors

Used on slow-speed serial lines
e.g. modems connected to the telephone network

Odd Parity
For each group of n bits, sender computes the n+1th bit so that the n+1 group contains an odd number of bits set to 1

Examples

0011010 0 1101100 1

Even Parity
Internet checksum

Motivation
Internet protocols are implemented in software and we would like to have efficient algorithms to detect transmission errors that are easy to implement.

Solution
Internet checksum
Sender computes for each segment and over the entire segment the 1s complement of the sum of all the 16 bits words in the segment.

Receiver recomputes the checksum over each received segment and verifies that it is correct. Otherwise, the
Internet checksum : example

Assume a segment composed of 48 bits

0110011001101100 0101010101010101 0000111100001111
Internet checksum : example

Assume a segment composed of 48 bits

0110011001101100 000011111000001111
0101010101010101
Internet checksum : example

Assume a segment composed of 48 bits

0110011001101100 0000111100001111
0101010101010101
1011101110111011
1011101110111011
Internet checksum : example

Assume a segment composed of 48 bits

0110011001101100
0101010101010101
0000111100001111
1011101110111011
0000111100001111
Internet checksum : example

Assume a segment composed of 48 bits

```
0110011001101100
0101010101010101
0000111100001111
1011101110111011
1100101011001010
```
Internet checksum : example

Assume a segment composed of 48 bits

0110011001101100
0101010101010101
0000111100001111
1011101110111011
1100101011001010
0011010100110101

0110011001101100
0101010101010101
0000111100001111
1011101110111011
1100101011001010
0011010100110101
**Cyclical Redundancy Check (CRC)**

**Principle**

Improve the performance of the Internet checksum by using polynomial codes

Sender and receiver agree on \( r+1 \) bits pattern called Generator (G)
Sender adds \( r \) bits of CRC to a \( d \) bits data segment such that the \( d+r \) bits pattern is exactly divisible by G using modulo 2 arithmetic

\[
D \times 2^r \text{ XOR } R = n\times G
\]

All computations are done in modulo 2 arithmetic by using XOR

\[
\begin{align*}
1011 + 0101 &= 1110 \\
1011 - 0101 &= 1110
\end{align*}
\]

\[
\begin{align*}
1001 + 1101 &= 0100 \\
1001 - 1101 &= 0100
\end{align*}
\]
Detection of transmission errors (2)

Behaviour of the receiver

If the checksum is correct
   Send an OK control segment to the sender to
       confirm the reception of the data segment
       allow the sender to send the next segment

If the checksum is incorrect
   The content of the segment is corrupted and must be discarded
   Send a special control segment (NAK) to the sender to
       ask it to retransmit the corrupted data segment
Protocol 3a : Sender

Sender

Wait for SDU  \( \xrightarrow{\text{Data.req(}SDU\text{)}} \) \( \xrightarrow{\text{send(}D\text{(SDU, CRC)}\text{)}} \)  
\[ \text{Recvd(}C\text{(OK)}\text{)} \]

Wait for OK/NAK  \( \xrightarrow{\text{Recvd(}C\text{(NAK)}\text{)}} \) \( \xrightarrow{\text{send(}D\text{(SDU, CRC)}\text{)}} \)
Protocol 3a : Receiver

Receiver

\( \text{Recvd}(D(SDU,CRC)) \)

AND \( \text{IsOK}(CRC,SDU) \)

Data.ind(SDU)

\( \rightarrow \) Process

segment OK

Wait

for segment

\( \rightarrow \) Process

segment KO

\( \neg \text{IsOK}(CRC,SDU) \)

\( \rightarrow \) Send(\(C(NAK)\))

\( \rightarrow \) Send(\(C(OK)\))

\( \rightarrow \) Send(\(C(NAK)\))

Recvd\(D(SDU,CRC))\)

AND \( \neg \text{IsOK}(CRC,SDU) \)
Protocol 3a : Example

A

B
Protocol 3a: Example

A
Data.req(a)

D(a)

B
Data.ind(a)
Protocol 3a : Example

A

Data.req(a)

Data.req(b)

B

Data.ind(a)

D(a)
Protocol 3a : Example

A
Data.req(a)

B
Data.ind(a)

C(OK)
D(a)
Protocol 3a : Example

A

Data.req(a)

Data.req(b)

D(a)

C(OK)

D(b')

Data.ind(a)

B

Transmission error

Invalid checksum
Protocol 3a : Example

A

Data.req(a)

D(a)

Data.req(b)

C(OK)

D(b')

C(NAK)

Transmittion error

B

Data.ind(a)

Invalid checksum
Protocol 3a: Example

A

Data.req(a)

Data.req(b)

Retransmission

B

Data.ind(a)

D(a)

C(OK)

D(b')

Invalid checksum

C(NAK)

Transmission error

© O. Bonaventure, 2008
Protocol 3a: Example

A
Data.req(a)

Data.req(b)

Retransmission

B
Data.ind(a)

Data.ind(b)

D(a)

D(b')

C(OK)

C(NAK)

D(b)

Invalid checksum

Transmission error

© O. Bonaventure, 2008
Protocol 3a: Example

A
Data.req(a)

Data.req(b)

Retransmission

B
Data.ind(a)

Data.ind(b)

D(a)

C(OK)

D(b')

Invalid checksum

Transmission error

C(NAK)

D(b)

C(OK)

Invalid checksum

© O. Bonaventure, 2008

CNP3/2008.3.
Protocol 3b

How can we provide a reliable service in the transport layer?

Hypotheses
1. The application sends small SDUs
2. The network layer provides a perfect service
   1. Transmission errors are possible
   2. Packets can be lost
   3. There is no packet reordering
   4. There are no duplications of packets
3. Data transmission is unidirectional

2. How to deal with these problems?
Protocol 3a and segment losses

How do segment losses affect protocol 3a?
Protocol 3a and segment losses

How do segment losses affect protocol 3a?
Protocol 3a and segment losses

How do segment losses affect protocol 3a?
Protocol 3a and segment losses

How do segment losses affect protocol 3a?
Protocol 3a and segment losses

How do segment losses affect protocol 3a?
Protocol 3a and segment losses

How do segment losses affect protocol 3a?

A Data.req(a)

B

D(a)

C(OK)

D(b)

Lost segment

DEADLOCK

A is waiting for a control segment

B is waiting for a data segment
Protocol 3b

Modification to the sender
Add a retransmission timer to retransmit the lost segment after some time

Data.req(\textbf{SDU})
\textbf{send}(D(\textbf{SDU}, \textbf{CRC}))
\textbf{start\_timer}()

Wait for SDU

\textbf{Recvd}(C(\textbf{OK}))
\textbf{cancel\_timer}()

Wait for OK/NAK

\textbf{Recvd}(C(\textbf{NAK}))
\textbf{send}(D(\textbf{SDU}, \textbf{CRC}))
\textbf{restart\_timer}()

\textbf{Timer expires}
\textbf{send}(D(\textbf{SDU}, \textbf{CRC}))
\textbf{restart\_timer}()

No modification to the receiver
Protocol 3b : Example

<table>
<thead>
<tr>
<th>A</th>
<th></th>
<th>B</th>
</tr>
</thead>
</table>
Protocol 3b : Example

A
Data.req(a)

D(a)
start timer

B

Data.ind(a)
Protocol 3b : Example

A → Data.req(a) → B

start timer

D(a)

C(OK)

cancel timer

Data.ind(a)
Protocol 3b: Example

A

Data.req(a)

Data.req(b)

B

Data.ind(a)

D(a)

C(OK)

© O. Bonaventure, 2008
Protocol 3b: Example

A

Data.req(a)

start timer

D(a)

Data.req(b)

cancel timer

C(OK)

B

Data.ind(a)
Protocol 3b: Example

- A
  - Data.req(a)
  - Data.req(b)
    - start timer
    - cancel timer

- B
  - D(a)
  - Data.ind(a)

- Lost segment
  - D(b)
Protocol 3b: Example

Data.req(a)

start timer

D(a)

C(OK)

data.ind(a)

Data.req(b)

cancel timer

D(b)

Lost segment

start timer

timer expires
Protocol 3b: Example

A

Data.req(a)

start timer

D(a)

Data.ind(a)

Data.req(b)

cancel timer

D(b)

C(OK)

Data.ind(a)

start timer

timer expires

D(b)

Lost segment

B

Data.ind(b)
Protocol 3b: Example

A

Data.req(a)

start timer

D(a)

cancel timer

C(OK)

Data.req(b)

start timer

D(b)

Lost segment

Data.ind(a)

Data.ind(b)

timer expires

C(OK)
Protocol 3b : Example

A

Data.req(a)

Data.req(b)

start timer

D(a)

cancel timer

C(OK)

D(b)

start timer

D(b)

C(OK)

timer expires

Lost segment

B

Data.ind(a)

Data.ind(b)

Does this protocol always work?
<table>
<thead>
<tr>
<th>A</th>
<th></th>
<th>B</th>
</tr>
</thead>
</table>
Protocol 3b : Example

A
Data.req(a)

B
Data.ind(a)

D(a)

C(OK)

start timer

cancel timer
Protocol 3b: Example

A

Data.req(a)

Data.req(b)

B

Data.ind(a)

Data.ind(b)

start timer

D(a)

C(OK)

D(b)

cancel timer
Protocol 3b : Example

A

Data.req(a)

start timer

D(a)

Data.ind(b)

cancel timer

D(b)

start timer

C(OK)

Lost segment

C(OK)

B

Data.ind(a)

Data.ind(b)
Protocol 3b: Example

A

Data.req(a)

Data.req(b)

B

Data.ind(a)

Data.ind(b)

Data.ind(b)

C(OK)

D(a)

C(OK)

D(b)

C(OK)

D(b)

start timer

cancel timer

Lost segment

start timer

timer expires
How to solve this problem?
Alternating bit protocol

Principles of the solution
Add **sequence numbers** to each data segment sent by sender.
By looking at the sequence number, the receiver can check whether it has already received this segment.

Contents of each segment
Data segments
Control segments

How many bits do we need for the sequence number?
a single bit is enough.
Alternating bit protocol
Sender

Wait for D(0,...)

Data.req(SDU)
send(D(0, SDU, CRC))
start_timer()

Wait for D(1,...)

Recvd(C(NAK?)) or recvd(C(OK0)) or timer expires
send(D(0, SDU, CRC))
restart_timer()

Recvd(C(OK0))
cancel_timer()

Start timer()

Recvd(C(OK1))
cancel_timer()

Wait for D(0,...)

Recvd(C(OK1))
cancel_timer()

Data.req(SDU)
send(D(1, SDU, CRC))
start_timer()

Wait for OK0/NAK

Wait for OK1/NAK

Recvd(C(NAK?)) or Recvd(C(OK1)) or timer expires
send(D(0, SDU, CRC))
restart_timer()

All corrupted segments are discarded in all states
Alternating bit protocol
Receiver

Recvd(\texttt{D}(0, \texttt{SDU}, \texttt{CRC}) )
AND Is\texttt{OK}(\texttt{CRC}, \texttt{SDU})

\texttt{Send(C(OK0))}

Recvd(\texttt{D}(0, \texttt{SDU}, \texttt{CRC}) )
AND Is\texttt{OK}(\texttt{CRC}, \texttt{SDU})

\texttt{Data.ind(\texttt{SDU})}

\texttt{Send(C(OK0))}

\texttt{Recvd(D(1, SDU, CRC))}
AND Is\texttt{OK}(\texttt{CRC}, \texttt{SDU})

\texttt{Send(C(OK1))}

\texttt{Wait for D(0,...)}

\texttt{Recvd(D(1, SDU, CRC))}
AND Is\texttt{OK}(\texttt{CRC}, \texttt{SDU})

\texttt{Send(C(OK1))}

\texttt{Bad(D(?, SDU, CRC))}

\texttt{Send(C(NAK0))}

\texttt{Process SDU0 OK}

\texttt{Recvd(D(1, SDU, CRC))}
AND Is\texttt{OK}(\texttt{CRC}, \texttt{SDU})

\texttt{Data.ind(\texttt{SDU})}

\texttt{Send(C(NAK1))}

\texttt{Send(C(OK1))}

\texttt{Recvd(D(0, SDU, CRC))}
AND Is\texttt{OK}(\texttt{CRC}, \texttt{SDU})

\texttt{Send(C(OK0))}

\texttt{Wait for D(1,...)}

\texttt{Send(C(OK0))}

\texttt{Send(C(NAK0))}

\texttt{Process SDU0 KO}

\texttt{Process SDU1 OK}

\texttt{Process SDU1 KO}
# Alternating bit protocol

## Example

| A |   | B |
Alternating bit protocol

Example

A                              B
Data.req(a) --> D(0,a) --> Data.ind(a)
       \       |        
       C(OK0)
Alternating bit protocol
Example

A
Data.req(a)

Data.req(b)

Data.ind(b)

B
Data.ind(a)

Data.ind(b)

D(0,a)

C(OK0)

D(1,b)
Alternating bit protocol
Example

A
Data.req(a)
Data.req(b)

Retransmission timer

D(1,b) recv'd
D(1,b) recv'd

B
Data.ind(a)
Data.ind(b)

Duplicate detected

D(0,a)
C(OK0)
D(1,b)
D(1,b)
C(OK1)
C(OK1)
Alternating bit protocol
Example

A

Data.req(a)

Data.req(b)

Retransmission timer

Data.req(c)

B

Data.ind(a)

Data.ind(b)

Lost segment

Duplicate detected

D(0,a)

C(OK0)

D(1,b)

D(1,b) recvd

C(OK1)

D(1,b) recvd

D(0,c)
Alternating bit protocol
Example

A

Data.req(a)

Data.req(b)

Retransmission timer

D(1, b) recv'd

Data.req(c)

Retransmission timer

D(1, b) recv'd

D(0, a)

C(OK0)

D(1, b)

B

Data.ind(a)

Data.ind(b)

Duplicate detected

D(1, b)

C(OK1)

C(OK1)

D(0, c)

Lost segment
Alternating bit protocol
Example

A
Data.req(a)
Data.req(b)
Retransmission timer
D(1,b) recvd
Data.req(c)
Retransmission timer
D(1,b) recvd

B
Data.ind(a)
Data.ind(b)
Duplicate detected
Data.ind(c)
Lost segment

D(0,a)
C(OK0)
D(1,b)
D(0,c)
D(1,c)
C(OK1)
D(1,b) recvd
C(OK1)
Performance of the alternating bit protocol

What is the performance of the ABP in this case
One-way delay : 250 msec
Physical layer throughput : 50 kbps
segment size : 1000 bits
Performance of the alternating bit protocol

What is the performance of the ABP in this case
One-way delay : 250 msec
Physical layer throughput : 50 kbps
segment size : 1000 bits

\[
\frac{1000}{50000} = 20 \text{ msec}
\]

250 msec

520 msec
Performance of the alternating bit protocol

What is the performance of the ABP in this case?
One-way delay: 250 msec
Physical layer throughput: 50 kbps
Segment size: 1000 bits

\[
\frac{1000}{50000} = 0.02 \text{ sec} = 20 \text{ msec}
\]

\[
250 \text{ msec} + 520 \text{ msec} = 770 \text{ msec}
\]

\[
\text{Performance is function of } \text{bandwidth} \times \text{round-trip-time}
\]
How to improve the alternating bit protocol?

Use a pipeline

Principle
The sender should be allowed to send more than one segment while waiting for an acknowledgement from the receiver.
How to improve the alternating bit protocol?

Use a pipeline Principle
The sender should be allowed to send more than one segment while waiting for an acknowledgement from the receiver.

A
Data.req(a)
Data.req(b)
...
Data.req(e)

B
D(0,a)
...
D(4,e)
Data.ind(a)
...
Data.ind(e)
How to improve the alternating bit protocol? (2)

Modifications to alternating bit protocol

Sequence numbers inside each segment
  Each data segment contains its own sequence number
  Each control segment indicates the sequence number of
  the data segment being acknowledged (OK/NAK)

Sender
  Needs enough buffers to store the data segments that
  have not yet been acknowledged to be able to retransmit
  them if required

Receiver
  Needs enough buffers to store the out-of-sequence
  segments
How to improve the alternating bit protocol ? (2)

Modifications to alternating bit protocol

Sequence numbers inside each segment
Each data segment contains its own sequence number
Each control segment indicates the sequence number of the data segment being acknowledged (OK/NAK)

Sender
Needs enough buffers to store the data segments that have not yet been acknowledged to be able to retransmit them if required

Receiver
Needs enough buffers to store the out-of-sequence segments

How to avoid an overflow of the receiver’s buffers ?
Sliding window

Principle
Sender keeps a list of all the segments that it is allowed to send
\[ \text{sending\_window} \]
Receiver also maintains a receiving window with the list of acceptable sequence number
\[ \text{receiving\_window} \]
Sender and receiver must use compatible windows
\[ \text{sending\_window} \leq \text{receiving\_window} \]
For example, window size is a constant for a given protocol or negotiated during connection establishment phase
Sliding windows: example

Sending and receiving window: 3 segments

A
Sending window

0 1 2 3 4 5 6 7 8

B
Sliding windows: example

Sending and receiving window: 3 segments

A
Sending window
0 1 2 3 4 5 6 7 8
Data.req(a)

D(0,a)

B
Data.ind(a)
Sliding windows: example

Sending and receiving window: 3 segments

A

Sending window

012
3 4 5 6 7 8

Data.req(a)

012
3 4 5 6 7 8

Data.req(b)

012
3 4 5 6 7 8

D(0,a)

D(1,b)

B

Data.ind(a)

Data.ind(b)
Sliding windows : example

Sending and receiving window : 3 segments

Sending window

A

0 1 2 3 4 5 6 7 8

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

B

0 1 2 3 4 5 6 7 8

Data.ind(a)

Data.ind(b)

Data.ind(c)
## Sliding windows: example

### Sending and receiving window: 3 segments

<table>
<thead>
<tr>
<th>A</th>
<th>Sending window</th>
</tr>
</thead>
<tbody>
<tr>
<td>012</td>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>012</td>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>012</td>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>012</td>
<td>3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

**Data.req(a)**

**D(0,a)**

**Data.req(b)**

**D(1,b)**

**Data.req(c)**

**D(2,c)**

**C(OK0)**

<table>
<thead>
<tr>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data.ind(a)</td>
</tr>
<tr>
<td></td>
<td>Data.ind(b)</td>
</tr>
<tr>
<td></td>
<td>Data.ind(c)</td>
</tr>
</tbody>
</table>

© O. Bonaventure, 2008
Sliding windows : example

Sending and receiving window : 3 segments

Sending window

A

B

Data.req(a)

Data.req(b)

Data.req(c)

Data.ind(a)

Data.ind(b)

Data.ind(c)

C(OK0)

C(OK1)
Sliding windows : example

Sending and receiving window : 3 segments

A
Sending window

012 3 4 5 6 7 8
012 3 4 5 6 7 8
012 3 4 5 6 7 8
012 3 4 5 6 7 8

Data.req(a)
Data.req(b)
Data.req(c)

D(0,a)
D(1,b)
D(2,c)

0123 4 5 6 7 8
0123 4 5 6 7 8

C(OK0)
C(OK1)

B

Data.ind(a)
Data.ind(b)
Data.ind(c)
Sliding windows: example

Sending and receiving window: 3 segments

A
Sending window

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

Data.req(a)

Data.req(b)

Data.req(c)

Data.req(d)

B

Data.ind(a)

Data.ind(b)

Data.ind(c)

Data.ind(d)

D(0,a)

D(1,b)

D(2,c)

D(3,d)

C(OK0)

C(OK1)
Sliding windows : example

Sending and receiving window : 3 segments

A
Sending window

Data.req(a)

Data.ind(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

C(OK0)

C(OK1)

C(OK2)

Data.req(d)

D(3,d)

Data.ind(d)

B

Data.ind(a)

Data.ind(b)

Data.ind(c)

Data.ind(d)
Sliding windows: example

Sending and receiving window: 3 segments

Sending window

A

B

Data.req(a)

Data.ind(a)

Data.req(b)

Data.ind(b)

Data.req(c)

Data.ind(c)

Data.req(d)

Data.ind(d)

D(0,a)

D(1,b)

D(2,c)

D(3,d)

C(OK0)

C(OK1)

C(OK2)
Sliding windows: example

Sending and receiving window: 3 segments

A

Sending window

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

B

Data.req(a)

D(0,a)

Data.ind(a)

Data.req(b)

D(1,b)

Data.ind(b)

Data.req(c)

D(2,c)

Data.ind(c)

Data.req(d)

C(OK0)

C(OK1)

C(OK2)

Data.ind(d)

Data.req(e)

D(3,d)

D(4,e)

Data.ind(e)
Encoding sequence numbers

Problem
How many bits do we have in the segment header to encode the sequence number
N bits means $2^N$ different sequence numbers

Solution
place inside each transmitted segment its sequence number modulo $2^N$
The same sequence number will be used for several different segments
be careful, this could cause problems...

Sliding window
List of consecutive sequence numbers (modulo $2^N$) that the sender is allowed to transmit
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

A
Sending window

0 1 2 3

B

OK(0)
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

Sending window

Data.req(a)

D(0,a)

D(1,b)

Data.ind(a)

Data.ind(b)

OK(0)
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

A
Sending window

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a)
Data.req(b)
Data.req(c)

D(0,a)
D(1,b)
D(2,c)

OK(0)

B

Data.ind(a)
Data.ind(b)
Data.ind(c)
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

A
Sending window

0 1 2
0 1 2
0 1 2
0 1 2

Data.req(a)
Data.req(b)
Data.req(c)

D(0,a)
D(1,b)
D(2,c)

C(OK0)

B

Data.ind(a)
Data.ind(b)
Data.ind(c)
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

Sending window

A

Data.req(a)
D(0,a)

Data.req(b)
D(1,b)

Data.req(c)
D(2,c)

C(OK0)

B

Data.ind(a)

Data.ind(b)

Data.ind(c)
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

Sending window

\begin{itemize}
  \item Data.req(a)
  \item Data.req(b)
  \item Data.req(c)
\end{itemize}

\begin{itemize}
  \item D(0,a)
  \item D(1,b)
  \item D(2,c)
\end{itemize}

\begin{itemize}
  \item C(OK0)
  \item C(OK1)
\end{itemize}

\begin{itemize}
  \item Data.ind(a)
  \item Data.ind(b)
  \item Data.ind(c)
\end{itemize}
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

Sending window

A

| 0 1 2 3 |
| 0 1 2 3 |
| 0 1 2 3 |
| 0 1 2 3 |

Data.req(a)

Data.req(b)

D(0,a)

D(0,a)

D(1,b)

D(1,b)

D(2,c)

C(OK0)

C(OK0)

C(OK1)

C(OK1)

Data.req(c)

Data.req(c)

Data.req(d)

Data.req(d)

D(2,c)

D(2,c)

D(3,d)

D(3,d)

OK(0)

C(OK0)

C(OK1)

C(OK1)

C(OK0)

C(OK1)

C(OK0)

C(OK1)

C(OK0)

C(OK1)

C(OK0)

C(OK1)

Data.ind(a)

Data.ind(a)

Data.ind(b)

Data.ind(b)

Data.ind(c)

Data.ind(c)

Data.ind(d)

Data.ind(d)
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

A
Sending window

B

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

C(OK0)

C(OK1)

Data.req(d)

D(3,d)

Data.ind(a)

Data.ind(b)

Data.ind(c)

Data.ind(d)
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

A
Sending window

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

0 1 2 3
0 1 2 3

Data.req(d)

C(OK0)

D(3,d)

B

Data.ind(a)

Data.ind(b)

Data.ind(c)

Data.ind(d)

C(OK1)

C(OK2)
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

A
Sending window

B

Data.req(a) → D(0,a)
Data.req(b) → D(1,b)
Data.req(c) → D(2,c)
Data.req(d) → C(OK0)
Data.req(e) → D(3,d)

Data.ind(a)
Data.ind(b)
Data.ind(c)

C(OK0)
C(OK1)
C(OK2)

Data.ind(d)
Data.ind(e)
Sliding window: second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field

Sending window

A

Data.req(a)
Data.req(b)
Data.req(c)
Data.req(d)
Data.req(e)

B

Data.ind(a)
Data.ind(b)
Data.ind(c)
Data.ind(d)

CNP3/2008.3.
Reliable transfer with a sliding window

How to provide a reliable data transfer with a sliding window
  How to react upon reception of a control segment?
  Sender’s and receiver’s behaviours

Basic solutions

Go-Back-N
  simple implementation, in particular on receiving side
  throughput will be limited when losses occur

Selective Repeat
  more difficult from an implementation viewpoint
  throughput can remain high when limited losses occur
GO-BACK-N

Principle
Receiver must be as simple as possible

Receiver
Only accepts consecutive in-sequence data segments
Meaning of control segments
Upon reception of data segment
  OKX means that all data segments, up to and including X have been received correctly
  NAKX means that the data segment whose sequence number is X contained an error or was lost

Sender
Relies on a retransmission timer to detect segment losses
Upon expiration of retransmission time or arrival of a NAK segment: retransmit all the unacknowledged data segments
the sender may thus retransmit a segment that was already received correctly but out-of-sequence at destination
Go-Back-N: Receiver

State variable

next : sequence number of expected data segment

Recvd(D(next, SDU, CRC))
AND NOT(IsOK(CRC, SDU))
\[\rightarrow\] discard(SDU);
\[\rightarrow\] send(C(NAK, next, CRC));

Wait

Recvd(D(next, SDU, CRC))
AND IsOK(CRC, SDU)
\[\rightarrow\] Data.ind(SDU)

\[\rightarrow\] Process SDU
\[\rightarrow\] OK

next = (next + 1);

\[\rightarrow\] send(C(OK, (next - 1), CRC));

Recvd(D(t<>next, SDU, CRC))
AND IsOK(CRC, SDU)
\[\rightarrow\] discard(SDU);
\[\rightarrow\] send(C(OK, (next - 1), CRC));
Go-Back-N: Sender

State variables

- **base**: sequence number of oldest data segment
- **seq**: first available sequence number
- **W**: size of sending window

### Code Snippet

```c
Recvd(C(?, ?, CRC))
and NOT(CRCOK(C(?, ?, CRC)))
```

```c
if (seq == base) { start_timer(); }
insert_in_buffer(SDU);
send(D(seq, SDU, CRC));
seq = seq + 1;
```

### State Transitions

1. **Wait**
   - **Recvd(C(OK, t, CRC))**
   - **base = (t + 1);**
   - **if (base == seq){ cancel_timer(); }**
   - **else{ restart_timer(); }**

### Data Request

```c
Data.req(SDU)
AND (seq < (base + w))
if (seq == base) { start_timer(); }
insert_in_buffer(SDU);
send(D(seq, SDU, CRC));
seq = seq + 1;
```

### NAK Reception

```c
Recvd(C(NAK, ?, CRC))
and CRCOK(C(NAK, ?, CRC))]
```

### Timer Expiration

```c
for (i = base; i < seq; i = i + 1)
{ send(D(i, SDU, CRC)); }
restart_timer();
```
Go-Back-N : Example

A
Sending window

B
Go-Back-N: Example

A
Sending window

0 1 2 3
0 1 2 3

Data.req(a)

D(0,a)

B

Data.ind(a)
Go-Back-N : Example

Sending window

Data.req(a)

Data.req(b)

D(0,a)

Transmission error

D(1,b)

Data.ind(a)

Invalid CRC, discarded
Go-Back-N: Example

Sending window:

A

Data.req(a)

Data.req(b)

Data.req(c)

D(0,a)

D(1,b)

D(2,c)

B

Data.ind(a)

Transmission error

Invalid CRC, discarded

Not expected seq num, discarded
Go-Back-N: Example

Sending window:

A

B

Data.req(a)  D(0,a)  C(OK,0)  Data.ind(a)

Data.req(b)  D(1,b)  D(2,c)  Transmission error

Data.req(c)

Transmission error

Invalid CRC, discarded

Not expected seq num, discarded
Go-Back-N : Example

Sending window

A

Data.req(a)

Data.ind(a)

D(0,a)

D(1,b)

D(2,c)

C(OK,0)

C(NAK,1)

Transmission error

Invalid CRC, discarded

Not expected seq num, discarded

B

Data.req(b)

Data.req(c)

Data.ind(a)
Go-Back-N : Example

Sending window

A

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

B

Data.ind(a)

C(OK,0)

C(NAK,1)

Transmission error

Invalid CRC, discarded

Not expected seq num, discarded
Go-Back-N : Example

Sending window

Data.req(a)
D(0,a)
Transmission error
Invalid CRC, discarded
Not expected seq num, discarded
Retransmission

Data.req(b)
D(1,b)

Data.req(c)
D(2,c)

Data.ind(a)

Data.ind(b)

Data.ind(c)
Go-Back-N: Example

Sending window

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

A

Data.req(a) → D(0,a)

Data.req(b) → D(1,b)

Data.req(c) → D(2,c)

Data.req(d) → Retransmission

Data.ind(a)

Data.ind(b) → Data.ind(c) → Data.ind(d)

Transmission error

Invalid CRC, discarded

Not expected seq num, discarded

Data.ind(c)

Data.ind(d)
Go-Back-N: Example

Sending window:

A

B

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

C(OK,0)

Data.req(c)

D(2,c)

C(NAK,1)

Data.req(d)

D(1,b)

C(OK,0)

Data.req(e)

D(2,c)

D(3,d)

Transmission error

Invalid CRC, discarded

Not expected seq num, discarded

Data.ind(b)

Data.ind(c)

Data.ind(d)

Data.ind(a)
Go-Back-N : Example

Sending window

A

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

B

- Data.ind(a)
- Data.ind(b)
- Data.ind(c)
- Data.ind(d)
- Data.ind(e)

Transmission error

Invalid CRC, discarded

Not expected seq num, discarded

Retransmission

Data.req(b)

D(0,a)

D(1,b)

D(2,c)

C(OK,0)

C(NAK,1)

C(OK,0)

D(1,b)

D(2,c)

D(3,d)

Sending window is full

Application blocked

e will be accepted and sent later
Go-Back-N : Example (2)

A
Sending window

B
Go-Back-N : Example (2)

Sending window

Data.req(a)

D(0,a)

Data.ind(a)
Go-Back-N : Example (2)

Sending window

A

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Segment lost

B

Data.ind(a)
Go-Back-N : Example (2)

Sending window

A

Data.req(a) → D(0,a)

Data.req(b) → D(1,b)

Data.req(c) → D(2,c)

B

Data.ind(a)

Segment lost

Not expected seq num, discarded
Go-Back-N : Example (2)

Sending window:

A

Segment lost

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

C(OK,0)

Not expected seq num, discarded

Data.ind(a)
**Go-Back-N : Example (2)**

**Sending window**

- **A**
  - Data req(a)
  - Data req(b)
  - Data req(c)

- **B**
  - Data ind(a)
  - Segment lost
  - Not expected seq num, discarded

- **C(OK,0)**
- **D(0,a)**
- **D(1,b)**
- **D(2,c)**

© O. Bonaventure, 2008

CNP3/2008.3.
Go-Back-N : Example (2)

Sending window

Data.req(a)
D(0,a)
C(OK,0)
Data.ind(a)

Data.req(b)
D(1,b)
C(OK,0)
Data.ind(b)

Data.req(c)
D(2,c)
C(OK,0)
Data.ind(c)

Segment lost

Retransmission timer expires

Not expected seq num, discarded
Go-Back-N : Example (2)

A

Sending window

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a)

Data.ind(a)

D(0,a)

Data.req(b)

D(1,b)

Segment lost

Data.req(c)

D(2,c)

C(OK,0)

Data.req(d)

D(1,b)

C(OK,0)

Data.ind(b)

D(2,c)

D(3,d)

Data.ind(c)

Data.ind(d)

B

Retransmission timer expires

Not expected seq num, discarded

CNP3/2008.3.
Go-Back-N : Example (2)

Sending window

Data.req(a)
D(0,a)
C(OK,0)
Data.ind(a)

Data.req(b)
D(1,b)
C(OK,0)
Data.ind(b)

Data.req(c)
D(2,c)
Data.ind(c)

Data.req(d)
D(1,b)
Data.ind(d)

Data.req(e)
D(2,c)
Data.ind(d)

Segment lost
Retransmission timer expires
Not expected seq num, discarded
Go-Back-N: Example (2)

Sending window

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a) → D(0,a)

Data.req(b) → D(1,b)

Data.req(c) → D(2,c)

Data.req(d) → D(3,d)

Data.req(e) → D(1,b)

Segment lost

C(OK,0) → D(2,c)

Not expected seq num, discarded

C(OK,0) → D(1,b)

Sending window is full

Application blocked

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

e will be accepted and sent later
Selective Repeat

Receiver
Uses a buffer to store the segments received out of sequence and reorder their content

Receiving window

Acknowledged segments
Out of sequence segments
Acceptable segments
Segments outside receiving window
Selective Repeat

Receiver
Uses a buffer to store the segments received out of sequence and reorder their content
Receiving window

Acknowledged segments → Out of sequence segments → Acceptable segments

... 0 1 2 3 4 5 → 6 7 8 9 10 → 11 12 13 14 15 ....

Semantics of the control segments

OKX
The segments up to and including sequence number X have been received

NAKX
The segment with sequence number X was errored

Sender
Upon detection of an errored or lost segment, sender retransmits only this segment
may require one retransmission timer per segment
Selective-Repeat: Receiver

State variable

- **next**: sequence number of expected data segment
- **Last**: last received in-sequence segment

```plaintext
Recvd(D(t, SDU, CRC))
AND NOT(IsOK(CRC, SDU))

discard(SDU);
send(C(NAK, t, CRC));
```

```plaintext
Recvd(D(t, SDU, CRC))
AND IsOK(CRC, SDU)

insert_in_buffer(SDU);
```

For all in sequence segments inside buffer

- **Data.ind(SDU)**;
- slide the sliding window;
- update next and last
- send(C(OK, (next-1)));

```
wait
```
Selective Repeat: Sender

State variables

- base: sequence number of oldest unacknowledged segment
- seq: first free sequence number
- $W$: size of sending window

Data.req($SDU$)

AND (window not full)

start_timer(seq);

insert_in_buffer($SDU$);

send(D(seq, $SDU$, CRC));

seq = (seq + 1);

[ Recvd(C(NAK,t,CRC))
  and CRCOK(C(NAK,t,CRC)) ]

or timer (t) expires

send(D(t,$SDU$,CRC));

} }

restart_timer(t);

Recvd(C(OK,t,CRC))

and CRCOK(C(OK,t,CRC))

for all segments $i \leq t$

cancel_timer(t);

slide sliding window to the right;

Recvd(C(?,?,CRC))

and NOT(CRCOK(C(?,?,CRC)))

Wait
Selective Repeat: Example

A
Sending window

B
Rec. window

0 1 2 3

0 1 2 3
Selective Repeat: Example

Sending window:

```
0 1 2 3
```

Rec. window:

```
0 1 2 3
```

Data.req(a)

```
D(0,a)
```

Data.ind(a)

```
0 1 2 3
```
Selective Repeat: Example

Sending window

Data.req(a)

Data.req(b)

D(0,a)

Transmission error

D(1,b)

Data.ind(a)

Rec. window

- Discard segment

A

B

© O. Bonaventure, 2008
Selective Repeat: Example

Sending window:

A

Data.req(a)

Data.req(b)

Data.req(c)

Transmission error

Rec. window:

B

Data.ind(a)

Discard segment

© O. Bonaventure, 2008
Selective Repeat: Example

A

Sending window

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

Transmission error

Data.ind(a)

Rec. window

0 1 2 3

Discard segment

Segment stored

0 1 2 3

B

0 1 2 3

0 1 2 3

0 1 2 3

© O. Bonaventure, 2008
Selective Repeat: Example

Sending window:
- Data.req(a)
- Data.req(b) → D(0,a)
- Data.req(c) → D(1,b), D(2,c)

Transmission error:
- Discard segment D(1,b)
- Segment stored

Rec. window:
- Data.ind(a) → C(OK,0)
- Discard segment 0123
- Segment stored 0123
Selective Repeat: Example

Sending window:

A

Data.req(a) → D(0,a)

Data.req(b) → D(1,b)

Data.req(c) → D(2,c)

Rec. window:

B

Data.ind(a) → C(OK,0)

Transmission error

C(NAK,1)

Discard segment

Segment stored
Selective Repeat : Example

A

Sending window

B

Rec. window

Transmission error

Data.req(a)

Data.req(b)

Data.req(c)

Data.ind(a)

Discard segment

Segment stored

C(OK,0)

C(NAK,1)

C(OK,0)
Selective Repeat: Example

Sending window

Data.ind(b)
D(0,a)
C(OK,0)
C(NAK,1)
C(OK,0)

Transmission error

Data.req(b)
D(1,b)

Segment stored

Data.req(c)
D(2,c)

Discard segment

Retransmission

Data.ind(a)

Data.ind(b)

Data.ind(c)

Rec. window

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

© O. Bonaventure, 2008
Selective Repeat: Example

Sending window: 0 1 2 3

Data.req(a)

Data.req(b)

Data.req(c)

Data.ind(a)

Data.ind(b)

Data.ind(c)

Transmission error

Discard segment

Segment stored

C(OK,0)

C(NAK,1)

C(OK,2)

C(OK,0)

D(0,a)

D(1,b)

D(2,c)

D(1,b)

C(OK,2)

Rec. window: 0 1 2 3

CNP3/2008.3.
Selective Repeat: Example

A

Sending window

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a)

Data.req(b)

Data.req(c)

Data.req(d)

Retransmission

D(0,a)

D(1,b)

D(2,c)

C(OK,0)

C(NAK,1)

C(OK,0)

D(1,b)

D(3,d)

C(OK,2)

B

Rec. window

0 1 2 3

Data.ind(a)

Data.ind(b)

Data.ind(c)

Data.ind(d)

Transmission error

Discard segment

Segment stored

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

© O. Bonaventure, 2008
Selective Repeat: Example

Sending window:

\begin{align*}
&0 1 2 3 \\
&0 1 2 3 \\
&0 1 2 3 \\
&0 1 2 3 \\
&0 1 2 3 \\
&0 1 2 3 \\
\end{align*}

Rec. window:

\begin{align*}
&0 1 2 3 \\
&0 1 2 3 \\
&0 1 2 3 \\
\end{align*}

Data.ind(a)

Data.req(b)

D(0,a)

Data.req(c)

D(1,b)

D(2,c)

Data.ind(a)

Data.ind(b)

Data.ind(c)

Data.ind(d)

C(OK,0)

C(NAK,1)

C(OK,0)

C(OK,2)

D(1,b)

D(3,d)

C(OK,0)

Transmission error

Discard segment

Segment stored

Retransmission
Selective Repeat: Example

Sending window

Data.ind(b)
Data.req(a)
Data.req(b)
Data.req(c)
Data.req(d)

Transmission error

Retransmission

C(OK,0)
C(NAK,1)
C(OK,0)
C(OK,2)
C(OK,3)

Transmission window

Rec. window

Data.ind(a)
Data.ind(b)
Data.ind(c)
Data.ind(d)

Discard segment
Segment stored

© O. Bonaventure, 2008
Selective Repeat : Example (2)

A
Sending window
0 1 2 3

B
Rec. window
0 1 2 3

© O. Bonaventure, 2008
Selective Repeat: Example (2)

A
Sending window

0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a)

D(0,a)

B
Rec. window

0 1 2 3

Data.ind(a)

0 1 2 3
Selective Repeat : Example (2)

Sending window

Data.req(a)

D(0,a)

Lost segment

Data.req(b)

D(1,b)

Data.ind(a)

Rec. window

© O. Bonaventure, 2008
Selective Repeat: Example (2)

Data.req(a) → D(0,a) → B

Data.req(b) → D(1,b) → B

Data.req(c) → D(2,c) → B

Lost segment

Sending window

0 1 2 3

0 1 2 3

Rec. window

0 1 2 3

Data.ind(a) → 0 1 2 3
Selective Repeat: Example (2)

Sending window:

A

Data.req(a)

Data.req(b)

Data.req(c)

Lost segment

D(0,a)

D(1,b)

D(2,c)

B

Rec. window

Data.ind(a)

Segment stored

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

© O. Bonaventure, 2008
Selective Repeat: Example (2)

Sending window:

A

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Lost segment

Data.req(c)

D(2,c)

C(OK,0)

Segment stored

Rec. window:

B

Data.ind(a)

Selective Repeat: Example (2)

A

Sending window

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

Lost segment

B

Rec. window

0 1 2 3

Data.ind(a)

0 1 2 3

Segment stored

C(OK,0)

Segment stored

C(OK,0)
Selective Repeat : Example (2)
Selective Repeat: Example (2)

A
Sending window

0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Data.req(a)

D(0,a)

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

C(OK,0)

C(OK,0)

Lost segment

Retransmission timer expires

Data.ind(a)

0 1 2 3

Data.req(b)

D(1,b)

Data.ind(c)

0 1 2 3

Data.ind(b)

0 1 2 3

Data.ind(c)

0 1 2 3

B
Rec. window

0 1 2 3

Segment stored

0 1 2 3

0 1 2 3

0 1 2 3

C(OK,2)
Selective Repeat: Example (2)

A

Sending window

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

Data.req(a)

D(0,a)

Lost segment

Data.req(b)

D(1,b)

Data.req(c)

D(2,c)

Retransmission timer expires

Data.req(d)

Recovery window

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

Data.ind(a)

D(1,b)

C(OK,0)

Segment stored

Data.ind(b)

D(3,d)

Data.ind(c)

C(OK,2)

Data.ind(d)

Data.ind(a)

D(1,b)

C(OK,0)

Data.ind(b)

D(3,d)

Data.ind(c)

C(OK,2)

Data.ind(d)

C(OK,0)
Selective Repeat: Example (2)

Sending window

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Data.req(\(a\))

D(0,\(a\))

Lost segment

Data.req(\(b\))

D(1,\(b\))

Segment stored

Retransmission timer expires

Data.req(\(c\))

D(2,\(c\))

C(OK,0)

C(OK,0)

Data.req(\(d\))

D(3,\(d\))

C(OK,2)

Rec. window

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Data.ind(\(a\))

Data.ind(\(b\))

Data.ind(\(c\))

Data.ind(\(d\))

© O. Bonaventure, 2008
Selective Repeat: Example (2)

Sending window:

A

Rec. window:

B

Data.ind(b)

Lost segment

Segment stored

Retransmission timer expires

C(OK,0)

Data.req(b)

D(0,a)

D(1,b)

D(2,c)

C(OK,0)

D(1,b)

D(3,d)

C(OK,2)

C(OK,3)

Data.ind(c)

Data.req(c)

Data.req(a)

Data.ind(a)

Data.ind(d)

Data.ind(c)

Data.ind(b)
Buffer management

Problem

A transport entity may support many transport connections at the same time. How can we share the available buffer among these connections? The number of connections changes with time. Some connections require large buffers while others can easily use smaller ones. *ftp versus telnet*
Buffer management (2)

Principle
Adjust the size of the receiving window according to the amount of buffering available on the receiver
Allow the receiver to advertise its current receiving window size to the sender

New information carried in control segments
\( \text{win} \) indicates the current receiving window’s size

Changes to sender
Sending window: \( \text{swin} \) (function of available memory)
Keep in a state variable the receiving window advertised by the receiver: \( \text{rwin} \)
At any time, the sender is only allowed to send data segments whose sequence number fits inside \( \min(\text{rwin}, \text{swin}) \)
### Buffer management (3)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Swin=3, rwin=1</td>
<td>Rwin=1</td>
</tr>
<tr>
<td>0</td>
<td>1 2 3</td>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

Buffer management (3)

A

Swin=3, rwin=1

Data.req(a)

D(0,a)

B

Rwin=1

Data.ind(a)

Swin=3, rwin=1
Buffer management (3)

A

Swin=3, rwin=1

0 1 2 3

Data.req(a)

D(0,a)

C(OK,0, w=1)

B

Rwin=1

0 1 2 3

Data.ind(a)

0 1 2 3
Buffer management (3)

A
Swin=3, rwin=1

0 1 2 3

Data.req(a)

D(0,a)

C(OK,0, w=1)

B
Rwin=1

0 1 2 3

Data.ind(a)
Buffer management (3)

A

Swin=3, rwin=1

Data.req(a)

D(0,a)

C(OK,0, w=1)

D(1,b)

Data.ind(a)

Data.ind(b)

B

Rwin=1

0 1 2 3

Swin=3, rwin=1

0 1 2 3

0 1 2 3

Swin=3, rwin=1

0 1 2 3

0 1 2 3
Buffer management (3)

A

Swin=3, rwin=1

Data.req(a)

D(0,a)

C(OK,0, w=1)

C(OK,0,w=3)

2 new buffers become available

Data.req(b)

C(OK,0, w=1)

D(1,b)

B

Rwin=1

Data.ind(a)

Data.ind(b)

Swin=3, rwin=1

Swin=3, rwin=3

Swin=3, rwin=3

© O. Bonaventure, 2008
Buffer management (3)

A

Swin=3, rwin=1

\[0123]\]

Data.req(a)

\[D(0,a)\]

C(OK,0, w=1)

\[0\]

\[1\]

\[2\]

\[3\]

\[C(OK,1,w=3)\]

\[D(1,b)\]

\[0\]

\[1\]

\[2\]

\[3\]

2 new buffers become available

\[Rwin=1\]

\[0\]

\[1\]

\[2\]

\[3\]

\[Data.ind(b)\]

\[Data.ind(a)\]

\[0\]

\[1\]

\[2\]

\[3\]

\[Swin=3, rwin=1\]

\[0\]

\[1\]

\[2\]

\[3\]

\[Swin=3, rwin=3\]

\[0\]

\[1\]

\[2\]

\[3\]
Buffer management (3)

A

Swin=3, rwin=1

Data.req(a)

D(0,a)

Data.ind(a)

D(1,b)

C(OK,0, w=1)

D(2,c)

C(OK,0, w=3)

D(3,d)

C(OK,1, w=3)

2 new buffers become available

B

Rwin=1

Data.ind(a)

0 1 2 3

Data.ind(b)

0 1 2 3

Swin=3, rwin=1

0 1 2 3

Swin=3, rwin=3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3
Buffer management (4)

A
Swin=3, rwin=1

0 1 2 3

B
Rwin=1

Receiver cannot handle segment immediately

Data.ind(a)

OK(0)
Buffer management (4)

Receiver cannot handle segment immediately
Buffer management (4)

A

\[ \text{Swin}=3, \text{rwin}=1 \]

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c} 0 & 1 & 2 & 3 \\ \end{array} \]

Data.req(a)

D(0, a)

C(OK, 0, w=0)

B

\[ \text{Rwin}=1 \]

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c} 0 & 1 & 2 & 3 \\ \end{array} \]

Receiver cannot handle segment immediately

Data.ind(a)
Buffer management (4)

A
Swin=3, rwin=1

Data.req(a)
D(0,a)
C(OK,0, w=0)

Data.req(b)

B
Rwin=1

Receiver cannot handle segment immediately
Data.ind(a)

Swin=3, rwin=0
Window blocked
No transmission possible
Buffer management (4)

A

Swin=3, rwin=1

0 1 2 3

Data.req(a)

D(0,a)

C(OK,0, w=3)

Receiver cannot handle segment immediately

Data.ind(a)

B

Rwin=1

0 1 2 3

Swin=3, rwin=0

0 1 2 3

Window blocked
No transmission possible

Lost segment

C(OK,0, w=0)

No transmission possible
Buffer management (4)

A

- Swin=3, rwin=1
- Data.req(a)
- D(0,a)
- C(OK,0,w=0)
- Receiver cannot handle segment immediately
- Data.ind(a)
- Swin=3, rwin=0
- Window blocked
- No transmission possible
- Lost segment

B

- Rwin=1
- 2 new buffers are available
- C(OK,0,w=3)
Buffer management (4)

A

Swin=3, rwin=1

Data.req(a)

D(0,a)

C(OK,0, w=0)

Receiver cannot handle segment immediately

Data.ind(a)

B

Rwin=1

2 new buffers are available

Swin=3, rwin=0

Lost segment

Window blocked

No transmission possible

Waits for control segment

Waits for data segment

Lost segment

Swin=3, rwin=0

0 1 2 3

Window blocked

No transmission possible

Waits for control segment

C(OK,0,w=3)

0 1 2 3

Waits for data segment

C(OK,0,w=3)
Buffer management (4)

A

Swin=3, rwin=1

Data.req(a)

D(0,a)

Swin=3, rwin=0

Data.req(b)

Lost segment

Window blocked
No transmission possible

Waits for control segment

C(OK,0, w=0)

C(OK,0, w=3)

Rwin=1

Data.ind(a)

Receiver cannot handle segment immediately

2 new buffers are available

Waits for data segment

How to recover from deadlock?
Persistance timer on receiver, resend control segment after timer expiration
Duplication and reordering

How can we provide a reliable service in the transport layer?

Hypotheses
1. The application sends small SDUs
2. The network layer provides a perfect service
   1. Transmission errors are possible
   2. Packets can be lost
   3. Packet reordering is possible
   4. Packets can be duplicated
3. Data transmission is unidirectional

2. How to deal with these problems?
Duplication and reordering (2)

Problem
A late segment could be confused with a valid segment
Duplication and reordering (2)

Problem
A late segment could be confused with a valid segment

A
\[\text{Data.req}(a)\]

\[\text{D}(0,a)\]

\[\text{C}(\text{OK},0)\]

B
\[\text{Data.ind}(a)\]
Problem
A late segment could be confused with a valid segment

A

Data.req(a)

D(0,a)

D(1,b)

D(3,d)

C(OK,0)

C(OK,0)

B

Data.ind(a)
Problem
A late segment could be confused with a valid segment

Timer expiration
Retransmission

A: Data.req(a)

B: Data.ind(a)

D(0,a)

C(OK,0)

D(1,b)

C(OK,0)

D(3,d)

C(OK,0)

D(1,b)

C(OK,3)

Data.ind(b)

Data.ind(e)
Duplication and reordering (2)

Problem
A late segment could be confused with a valid segment

Timer expiration
Retransmission

Data.req(a)
D(1,b)  D(0,a)
C(OK,0)
D(1,b)  D(3,d)
C(OK,0)
D(1,b)
C(OK,3)
D(0,e)
C(OK,0)

Data.ind(a)
Data.ind(b)
Data.ind(e)
Data.ind(e)
Duplication and reordering (2)

Problem
A late segment could be confused with a valid segment

Data.req(a) → D(1,b) → C(OK,0)
D(0,a) → D(1,b) → C(OK,0)
D(3,d) → C(OK,0)
D(0,e) → C(OK,0)
C(OK,3)
C(OK,0)
C(OK,0)
C(OK,1)

Data.ind(a) → B
Data.ind(b) → C(OK,0)
Data.ind(e) → C(OK,3)
Data.ind(e) → C(OK,0)
Data.ind(b) → C(OK,1)

Timer expiration
Retransmission

© O. Bonaventure, 2008
Duplication and reordering (3)

How to deal with duplication and reordering?

Possible provided that segments do not remain forever inside the network

Constraint on network layer
  A packet cannot remain inside the network for more than MSL seconds

Principle of the solution
  Only one segment carrying sequence number $x$ can be transmitted during MSL seconds
  upper bound on maximum throughput
Bidirectional flow

How can we allow both hosts to transmit data?

Principle
Each host sends both control and data segments

Piggybacking
Place control fields inside the data segments as well (e.g. window, ack number) so that data segments also carry control information
Reduces the transmission overhead

Type: D or C
Seq: segment’s sequence number
Ack: sequence number of the last received in-order segment
Bidirectional flow
Example

A

OK(0)

B
Bidirectional flow
Example

A

Data.req(a)

D(0,0,a)

OK(0)

B

Data.ind(a)
Bidirectional flow
Example

A

Data.req(a)
D(0,0,a)

Data.ind(w)
D(5,0,w)

Data.ind(a)

D(5,0,w) acks D(0,0,a)

B

Data.req(w)
Bidirectional flow
Example

Data.req(a)
Data.req(b)
Data.ind(w)

D(0,0,a)
D(1,0,b)
D(5,0,w)

Error

D(5,0,w) acks D(0,0,a)

Data.req(w)
Data.ind(a)
Discarded
Bidirectional flow Example

A

Data.req(a)

Data.req(b)

Data.req(c)

Data.ind(w)

D(5,0,w) acks D(0,0,a)

B

Data.req(w)

Data.ind(a)

D(5,0,w) -> Segment -> buffer

Error

D(0,0,a)

D(1,0,b)

D(2,0,c)

D(5,0,w)
Bidirectional flow
Example

Data.req(a)
D(0,0,a)

Data.req(b)
D(1,0,b)

Data.req(c)
D(2,0,c)

Data.ind(a)
D(1,5,b)

C(NAK,1)

D(5,0,w)

Data.ind(w)
D(0,0,a)

Data.ind(b)
C(OK,2)

Data.ind(c)
C(OK,3)

C(OK,2)

D(5,0,w) acks D(0,0,a)

Retransmission

Error

Discarded

Segment -> buffer
Bidirectional flow Example

A

Data.req(a)

Data.req(b)

Data.req(c)

Data.ind(w)

D(5,0,w) acks D(0,0,a)

Retransmission

Data.ind(x)

D(5,0,w) -> buffer

Data.ind(w)

D(1,0,b)

Data.ind(a)

Data.req(w)

Discarded

Data.ind(b)

Data.ind(c)

C(OK,2)

Data.ind(x)

D(1,5,b)

C(NAK,1)

D(2,0,c)

C(OK,3)

D(0,0,a)

Error

D(6,0,x)
Bidirectional flow
Example

A

Data.req(a)
Data.req(b)
Data.req(c)
Data.ind(w)

D(5,0,w) acks D(0,0,a)
Retransmission

Data.ind(x)

B

Data.req(w)
Data.ind(a)
Discarded

Data.req(x)
Segment -> buffer

Data.ind(b)
Data.ind(c)
Data.ind(d)

D(1,5,b)
D(2,0,c)
D(0,0,a)
D(1,0,b)
D(5,0,w)
C(NAK,1)
D(6,0,x)
D(3,6,d)

C(OK,2)
C(OK,3)

© O. Bonaventure, 2008
Byte stream service

How to provide a byte stream service?

Principle

Sender splits the byte stream in segments
Receiver delivers the payload of the received in-sequence segments to its user
Usually each octet of the byte stream has its own sequence number and the segment header contains the sequence number of the first byte of the payload
   In this case, window sizes are often also expressed in bytes
Byte stream service (2)

A

| OK(0) |

B
Byte stream service (2)

Data.req(abcdef) → B

D(0,ab) → C(OK,1) → B

Data.ind(ab) → B

CNP3/2008.3.
Byte stream service (2)

Data.req(abcdef)

D(0,ab)

D(2,cd)

Lost segment

C(OK,1)

Data.ind(ab)

OK(0)

C(OK,1)
Byte stream service (2)

A

Data.req(abcdef)

D(0,ab) → B
D(2,cd)
D(4,ef)

Lost segment

C(OK,1)

C(OK,1)

Placed in buffer

B

Data.ind(ab)

Placed in buffer
Byte stream service (2)

A

Data.req(abcdef)

B

Data.ind(ab)

Expiration timer
Retransmission

Lost segment

D(0,ab)

D(2,cd)

D(4,ef)

C(OK,1)

C(OK,1)

D(2,cd)

C(OK,5)

Placed in buffer

Data.ind(cdef)

C(OK,5)
Byte stream service (2)

A

Data.req(abcdef)

Expiration timer
Retransmission

Data.req(ijkl)

Data.req(mnop)

B

Lost segment

Data.ind(ab)

Placed in buffer

Data.ind(cdef)

D(0,ab)

D(2,cd)

D(4,ef)

C(OK,1)

C(OK,1)

C(OK,5)
Byte stream service (2)

A

Data.req(abcdef)

Expiration timer
Retransmission

Data.req(ijkl)

Data.req(mnop)

B

Data.ind(ab)

Placed in buffer

D(0,ab)

Lost segment

D(2,cd)

C(OK,1)

D(4,ef)

C(OK,1)

D(2,cd)

D(6,ijklmnop)

C(OK,5)

C(OK,1)

Data.ind(cdef)

Data.ind(ijklmnop)

Data.ind(ijklmnop)
Byte stream service (2)

Data.req(abcdef)

Data.req(ijkl)

Data.req(mnoph)

Expiration timer
Retransmission

Lost segment

Data.ind(ab)

Placed in buffer

Data.ind(cdef)

Data.ind(ijklmnop)

C(OK,1)

C(OK,1)

C(OK,5)

C(OK,13)

OK(0)

D(0,ab)

D(2,cd)

D(4,ef)

D(2,cd)

D(6,ijklmnop)

C(OK,1)
Module 3 : Transport Layer

Basics

Building a reliable transport layer
- Reliable data transmission
- Connection establishment
- Connection release

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol
Transport connection establishment

How to open a transport connection between two transport entities?

The transport layer uses the imperfect network layer service
  Transmission errors are possible
  Segments can get lost
  Segments can get reordered
  Segments can be duplicated

Hypothesis
  We will first assume that a single transport connection needs to be established between the two transport entities
Simple solution

---

Principle

2 control segments

- CR is used to request a connection establishment
- CA is used to acknowledge a connection establishment

Is this sufficient with an imperfect network layer service?
Simple solution

Principle
2 control segments
CR is used to request a connection establishment
CA is used to acknowledge a connection establishment

Is this sufficient with an imperfect network layer service?
Simple solution

2 control segments
- CR is used to request a connection establishment
- CA is used to acknowledge a connection establishment

Is this sufficient with an imperfect network layer service?
Simple solution

Principle
2 control segments
- CR is used to request a connection establishment
- CA is used to acknowledge a connection establishment

Is this sufficient with an imperfect network layer service?
Simple solution (2)

How to deal with losses and transmission errors?
Control segments must be protected by CRC or checksum
Retransmission timer is used to protect against segment losses segments
Simple solution (2)

How to deal with losses and transmission errors?
Control segments must be protected by CRC or checksum
Retransmission timer is used to protect against segment losses segments

Connect.req() → CR
Simple solution (2)

How to deal with losses and transmission errors?
Control segments must be protected by CRC or checksum
Retransmission timer is used to protect against segment losses segments

Connect.req() → CR
Retransmission timer expires

CR → CR

→ Connect.ind()
How to deal with losses and transmission errors?
Control segments must be protected by CRC or checksum
Retransmission timer is used to protect against segment losses segments
Simple solution (2)

How to deal with losses and transmission errors?
Control segments must be protected by CRC or checksum.
Retransmission timer is used to protect against segment losses segments.

Connect.req() \rightarrow CR
Retransmission timer expires

CR \rightarrow Connect.ind()

Connect.conf() \rightarrow CA

Connection established

Connect.resp
Simple solution (2)

How to deal with losses and transmission errors?

Control segments must be protected by CRC or checksum.
Retransmission timer is used to protect against segment losses segments.
Connection establishment

How to deal with duplicated or delayed packets?
A duplicated CR should not lead to the establishment of two transport connections instead of a single one.
Connection establishment

How to deal with duplicated or delayed packets?
A duplicated CR should not lead to the establishment of two transport connections instead of a single one.
How to deal with duplicated or delayed packets?
A duplicated CR should not lead to the establishment of two transport connections instead of a single one.
Connection establishment

How to deal with duplicated or delayed packets?
A duplicated CR should not lead to the establishment of two transport connections instead of a single one.

First connection established
Connect.req()
Connect.conf()
CR
CA

First connection stopped
Connect.ind()
Connect.resp

First connection stopped

© O. Bonaventure, 2008
Connection establishment

How to deal with duplicated or delayed packets?
A duplicated CR should not lead to the establishment of two transport connections instead of a single one.
How to deal with duplicated or delayed packets?

A duplicated CR should not lead to the establishment of two transport connections instead of a single one.

Connect.req() \(\rightarrow\) CR

Connect.conf() \(\rightarrow\) CA

Old previous CR

Connect.ind() \(\rightarrow\) Connect.resp

First connection established

First connection stopped

First connection established

First connection stopped

© O. Bonaventure, 2008
Connection establishment

How to deal with duplicated or delayed packets?
A duplicated CR should not lead to the establishment of two transport connections instead of a single one.
Connection establishment

How to deal with duplicated or delayed packets?
A duplicated CR should not lead to the establishment of two transport connections instead of a single one.
Connection establishment

How to deal with duplicated or delayed packets?
A duplicated CR should not lead to the establishment of two transport connections instead of a single one.

Connect.req() → CR
Connect.conf() → CA
First connection established

First connection stopped

Old previous CR

CR → CA
CR

Connect.ind() → Connect.resp
First connection established
First connection stopped

How to detect duplicates?

D
Connection establishment (2)

How to detect duplicates?

Principles
The network layer guarantees by its protocols and internal organisation that a packet and its duplicates will not live forever inside the network.
- No packet will survive more than MSL seconds inside the network.
- Transport entities rely on a local clock to detect duplicated connection establishment requests.
Connection establishment (3)

Transport clock

Maintained by each transport entity
usually implemented as a k-bits counter

\[ 2^k \times \text{clock cycle} \gg MSL \]

Must continue to count even if the transport entity stops
or reboots

Transport clocks are not synchronised
neither with other transport clocks nor with realtime
Connection establishment (3)

Transport clock

Maintained by each transport entity, usually implemented as a k-bits counter:
\[ 2^k \times \text{clock cycle} \gg \text{MSL} \]

Must continue to count even if the transport entity stops or reboots.

Transport clocks are not synchronised neither with other transport clocks nor with realtime.
Three way handshake

Host A

Host B
Three way handshake

Host A

Sequence number $x$ read from local transport clock

**Local state:**
- Connection to B:
  - Wait for ack for CR ($x$)
  - Start retransmission timer

CR ($\text{seq}=x$)

Host B
Three way handshake

Sequence number $x$ read from local transport clock

**Local state:**
Connection to B:
- Wait for ack for CR ($x$)
- Start retransmission timer

Host A

CR (seq=$x$)

CA (seq=$y$, ack=$x$)

Host B

Sequence number $y$ read from local transport clock
CA sent to ack CR

**Local state:**
Connection to A:
- Wait for ack for CA($y$)
Three way handshake

Host A

Sequence number $x$ read from local transport clock

**Local state:**
- Connection to B:
  - Wait for ack for CR ($x$)
  - Start retransmission timer

Received CA acknowledges CR
Send CA to ack received CA

**Local state:**
- Connection to B:
  - established
  - current_seq = $x$

*Connection established*

Host B

Sequence number $y$ read from local transport clock
CA sent to ack CR

**Local state:**
- Connection to A:
  - Wait for ack for CA($y$)
Three way handshake

Host A

Sequence number \( x \) read from local transport clock

Local state:
- Connection to B:
  - Wait for ack for CR \((x)\)
  - Start retransmission timer

Received CA acknowledges CR
Send CA to ack received CA

Local state:
- Connection to B:
  - established
  - current_seq = \( x \)

Connection established

Host B

Sequence number \( y \) read from local transport clock
CA sent to ack CR

Local state:
- Connection to A:
  - Wait for ack for CA(\(y\))

Local state:
- Connection to A:
  - established
  - current_seq = \( y \)

Connection established
Three way handshake

Host A

Sequence number x read from local transport clock

**Local state:**
Connection to B:
- Wait for ack for CR (x)
- Start retransmission timer

Received CA acknowledges CR
Send CA to ack received CA

**Local state:**
Connection to B:
- established
- current_seq = x

*Connection established*

The sequence numbers used for the data segments will start from x

Host B

Sequence number y read from local transport clock
CA sent to ack CR

**Local state:**
Connection to A:
- Wait for ack for CA(y)

**Local state:**
Connection to A:
- established
- current_seq=y

*Connection established*
Three way handshake

Host A

Sequence number \( x \) read from local transport clock

**Local state:**
- Connection to B :
  - Wait for ack for CR (\( x \))
  - Start retransmission timer

Received CA acknowledges CR
Send CA to ack received CA

**Local state:**
- Connection to B :
  - established
  - current_seq = \( x \)

**Connection established**

The sequence numbers used for the data segments will start from \( x \)

Host B

Sequence number \( y \) read from local transport clock
CA sent to ack CR

**Local state:**
- Connection to A :
  - Wait for ack for CA(\( y \))

**Local state:**
- Connection to A :
  - established
  - current_seq = \( y \)

**Connection established**

The sequence numbers used for the data segments will start from \( y \)
Three way handshake (2)

What happens with duplicates
Duplicate CR

Host A          |          | Host B
What happens with duplicates
Duplicate CR

Host A  | CR (seq=Z) | Host B
Three way handshake (2)

What happens with duplicates

Duplicate CR

Host A

CR (seq=z)

Host B

Sequence number y read from local transport clock
Acknowledges CR segment

**Local state:**
Connection to A:
- Wait for ack for CA(y)
Three way handshake (2)

What happens with duplicates
Duplicate CR

Host A

CR (seq=z)

CA (seq=y, ack=z)

Host B

Sequence number y read from local transport clock
Acknowledges CR segment
Local state:
Connection to A:
- Wait for ack for CA(y)
Three way handshake (2)

What happens with duplicates
Duplicate CR

Host A
Local state:
No connection to B
Send REJECT to cancel connection establishment

CR (seq=z)

CA (seq=y, ack=z)

Host B
Sequence number y read from local transport clock
Acknowledges CR segment
Local state:
Connection to A:
- Wait for ack for CA(y)
Three way handshake (2)

What happens with duplicates
Duplicate CR

Host A

CR (seq=z)

CA (seq=y, ack=z)

REJECT (ack=y)

Host B

Sequence number y read from local transport clock
Acknowledges CR segment

Local state :
Connection to A :
- Wait for ack for CA(y)

Connection cancelled

Local state :
No connection to B
Send REJECT to cancel connection establishment

No connection is established
Three way handshake (3)

Host A

Current state does not contain a CR with seq=x

Host B
Three way handshake (3)

Host A

Sequence number $z$ read from local transport clock

**Local state:**
- Connection to B:
  - Wait for ack for CR ($z$)
  - Start retransmission timer

Current state does not contain a CR with seq=$x$

Host B

CR (seq=$z$)
Three way handshake (3)

Host A

Sequence number \( z \) read from local transport clock

**Local state:**
Connection to B:
- Wait for ack for CR (\( z \))
- Start retransmission timer

Current state does not contain a CR with seq=\( x \)

---

Host B

CR (seq=\( z \))

CA (seq=\( y \), ack=\( x \))
Three way handshake (3)

Host A

Sequence number $z$ read from local transport clock

Local state:
Connection to B:
- Wait for ack for CR ($z$)
- Start retransmission timer

Current state does not contain a CR with seq=$x$

Host B

Current state does not contain a segment with seq=$y$

REJECT ignored

CR (seq=$z$)

CA (seq=$y$, ack=$x$)

REJECT (ack=$y$)
Three way handshake (3)

Host A

Sequence number $z$ read from local transport clock

**Local state:**
- Connection to B:
  - Wait for ack for CR ($z$)
  - Start retransmission timer

Current state does not contain a CR with seq=$x$

Retransmission timer expires

Host B

Current state does not contain a segment with seq=$y$
REJECT ignored

Sequence number $w$ read from local transport clock
CA sent to ack CR

**Local state:**
- Connection to A:
  - Wait for ack for CA($w$)

CR (seq=$z$)

CA (seq=$y$, ack=$x$)

REJECT (ack=$y$)

CA (seq=$w$, ack=$z$)
Three way handshake (3)

Host A

Sequence number \( z \) read from local transport clock

**Local state:**
- Connection to B:
  - Wait for ack for CR (\( z \))
  - Start retransmission timer

Current state does not contain a CR with seq=\( x \)

Retransmission timer expires

Received CA acknowledges CR
Send CA to ack received CA

**Local state:**
- Connection to B:
  - established
  - current_seq = \( z \)

Connection established

CR (seq=\( z \))

CA (seq=\( y \), ack=\( x \))

REJECT (ack=\( y \))

CR (seq=\( z \))

CA (seq=\( w \), ack=\( z \))

CA (seq=\( z \), ack=\( w \))

Host B

Current state does not contain a segment with seq=\( y \)
REJECT ignored

Sequence number \( w \) read from local transport clock
CA sent to ack CR

**Local state:**
- Connection to A:
  - Wait for ack for CA(\( w \))

Connection established

CNP3/2008.3.
Three way handshake (4)

Another scenario

Host A  |  Host B
Three way handshake (4)

Another scenario

Host A

CR (seq=z)

Host B
Another scenario

Host A

CA (seq=w, ack=z)

CR (seq=z)

Sequence number w read from local transport clock
Acknowledges CR segment
Local state:
- Connection to A:
  - Wait for ack for CA(w)

Host B
Three way handshake (4)

Another scenario

Host A

- Current state does not contain a CR with seq=z
- CA (seq=w, ack=z)
- REJECT (ack=w)

Host B

- Sequence number w read from local transport clock
- Acknowledges CR segment
- Local state:
  - Connection to A:
    - Wait for ack for CA(w)

Host A

Host B

© O. Bonaventure, 2008
Another scenario

Host A

- CA (seq=w, ack=z)
- REJECT (ack=w)
- CA (seq=z, ack=y)
- REJECT (ack=z)

Current state does not contain a CR with seq=z

No connection is established

Host B

Sequence number w read from local transport clock
Acknowledges CR segment

Local state:
- Connection to A:
  - Wait for ack for CA(w)

Invalid CA received from A
Send REJECT

No connection is established
Module 3 : Transport Layer

Basics

Building a reliable transport layer
  Reliable data transmission
  Connection establishment
  Connection release

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol
A transport connection can be used in both directions

Types of connection release

Abrupt connection release
One of the transport entities closes both directions of data transfer
can lead to losses of data

Graceful release
Each transport entity closes its own direction of data transfer
connection will be closed once all data has been correctly delivered
Abrupt release

Connection closed

Connection closed
Abrupt release

CR (seq=z)
CA (seq=w, ack=z)
CA (seq=z, ack=w)

Connection closed
Abrupt release

Data.req()

CA (seq=w, ack=z)

CR (seq=z)

CA (seq=z, ack=w)

D

Connection closed

Data.ind()

Connection closed
Abrupt release

Data.req()

Disc.req()

Connection closed

CR (seq=z)

CA (seq=w, ack=z)

CA (seq=z, ack=w)

D

Connection closed

Data.ind()

Disc.req()
Abrupt release

These segments will not be delivered!
Abrupt release (2)

A transport layer entity may itself be forced to release a transport connection

- the same data segment has been transmitted multiple times without receiving an acknowledgement
- the network layer reports that the destination host is not reachable anymore
- the transport layer entity does not have enough resources available to support this connection (e.g. not enough memory)

In this case, the transport layer entity will perform an abrupt disconnection
Graceful shutdown

Principle
Each entity closes its own direction of data transfer once all its data have been sent
Graceful shutdown

Principle
Each entity closes its own direction of data transfer once all its data have been sent.

DISCONNECT.req (A-B) \rightarrow DR(A-B) \rightarrow \text{DISCONNECT.ind(A-B)}
Graceful shutdown

Principle
Each entity closes its own direction of data transfer once all its data have been sent

DISCONNECT.req (A-B) → DR(A-B) → DISCONNECT.ind(A-B)

Incoming connection (A->B) closed
Graceful shutdown

Principle
Each entity closes its own direction of data transfer once all its data have been sent

DISCONNECT.req (A-B) → DR(A-B) → DISCONNECT.ind(A-B) → ACK

DISCONNECT.conf(A-B) ← Incoming connection (A->B) closed

Outgoing connection (A->B) closed
Graceful shutdown

Principle
Each entity closes its own direction of data transfer once all its data have been sent

Outgoing connection (A->B) closed

Incoming connection (B->A) closed
Reliability of the transport layer

Limitations
Transport layer provides a reliable data transfer during the lifetime of the transport connection. If a connection is gracefully shutdown, then all the data sent of this connection have been received correctly. Data transfer may be unreliable (e.g. loss of segments) if the connection is abruptly released.

Transport layer does not recover itself from abrupt connection releases

Possible solutions
- Application reopens the connection and restarts the data transfer
- Session Layer
- Transaction processing
Module 3 : Transport layer

Basics

Building a reliable transport layer

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol
A simple transport protocol

User Datagram Protocol (UDP)
The simplest transport protocol

Goal
Allow applications to exchange small SDUs by relying on the IP service on most operating systems, sending raw IP packets requires special privileges while any application can use directly the transport service

Constraint
The implementation of the UDP transport entity should remain as simple as possible
UDP : design choices

Which mechanisms inside UDP ?

Application identification
Several applications running on the same host must be able to use the UDP service

Solution
Source port to identify sending application
Destination port to identify receiving application
Each UDP segment contains both the source and the destination ports

Detection of transmission errors
UDP protocol

2 UDP entities exchange UDP segments

UDP segment format

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

Payload

32 bits

8 bytes
**UDP protocol**

2 UDP entities exchange UDP segments

**UDP segment format**

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

- **Source Port**: Used to identify the application that sent this segment on sending host.
- **Destination port**: 32 bits
- **UDP length**: 8 bytes
- **UDP Checksum**: 8 bytes

Payload
2 UDP entities exchange UDP segments

UDP segment format

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

Payload

8 bytes

32 bits

Used to identify the application that sent this segment on sending host

Used to identify the application that will receive this segment on destination host
UDP protocol

2 UDP entities exchange UDP segments

UDP segment format

- Source Port
- Destination port
- UDP length
- UDP Checksum
- Payload

8 bytes

32 bits

Checksum computed over the entire UDP segment and part of the IP header to detect transmission errors. 0 means that the sender did not compute a checksum.

Used to identify the application that will receive this segment on destination host.

Used to identify the application that sent this segment on sending host.
UDP protocol

2 UDP entities exchange UDP segments

UDP segment format

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

- **Checksum** computed over the entire UDP segment and part of the IP header to detect transmission errors. 0 means that the sender did not compute a checksum.

- **Used to identify the application that will receive this segment on destination host**

- **Constraint** Each UDP segment must fit inside a single IP packet
UDP Protocol (2)

Utilisation of the UDP ports

Request

Client

Source port : 1234
Destination port: 5678

Server
UDP Protocol (2)

Utilisation of the UDP ports

Request

Client

Source port: 1234
Destination port: 5678

Server

Source port: 5678
Destination port: 1234

Response
Limitations of the UDP service

Limitations

Maximum length of UDP SDUs depends on maximum size of IP packets

Unreliable connectionless service
   SDUs can get lost but transmission errors will be detected

UDP does not preserve ordering

UDP does not detect nor prevent duplication
Usage of UDP

Request-response applications where requests and responses are short and short delay is required or used in LAN environments
- DNS
- Remote Procedure Call
- NFS
- Games

Multimedia transfer were reliable delivery is not necessary and retransmissions would cause too long delays
- Voice over IP
- Video over IP
Module 3 : Transport Layer

Basics

Building a reliable transport layer

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol

TCP connection establishment
TCP connection release
Reliable data transfer
Congestion control
TCP

Transmission Control Protocol

Provides a reliable byte stream service

Characteristics of the TCP service

TCP connections
Data transfer is reliable
  no loss
  no errors
  no duplications
Data transfer is bidirectional
TCP relies on the IP service
TCP only supports unicast
TCP connection

How to identify a TCP connection

- Address of the source application
  - IP Address of the source host
  - TCP port number of the application on source host
- Address of the destination application
  - IP Address of the destination host
  - TCP port number of the application on destination host

Each TCP segment contains the identification of the connection it belongs to.
TCP connection (2)

Usage of TCP port numbers

Request

Client : C
Source Port : 1234
Destination Port: 5678

Server : S
Source Port : 5678
Destination Port: 1234

Response

Established TCP connections on client
Local IP  Remote IP  Local Port Remote Port
C        S          1234      5678

Established TCP connections on server
Local IP  Remote IP  Local Port Remote Port
S        C          5678      1234
TCP protocol

Single segment format

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>Reserved</td>
<td>Flags</td>
</tr>
<tr>
<td>THL</td>
<td>Window</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
</tbody>
</table>

Optional header extension

Payload

32 bits

20 bytes
TCP protocol

Single segment format

Flags:
- used to indicate the function of a segment
- SYN: used during establishment
- FIN: used during connection release
- RST: used in case of problems
- ACK: if true, means that the Acknowledgement number inside the segment is valid

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

<table>
<thead>
<tr>
<th>Sequence number</th>
<th>Acknowledgement number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>THL</th>
<th>Reserved</th>
<th>Flags</th>
<th>Window</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Optional header extension

Payload

32 bits

20 bytes
TCP protocol

Single segment format

Flags:
- used to indicate the function of a segment
  - SYN: used during establishment
  - FIN: used during connection release
  - RST: used in case of problems
  - ACK: if true, means that the Acknowledgement number inside the segment is valid

Segment header length

32 bits

Source port | Destination port

Sequence number

Acknowledgement number

THL | Reserved | Flags | Window

Checksum | Urgent pointer

Optional header extension

Payload

© O. Bonaventure, 2008
## TCP protocol

### Single segment format

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Source address of the sending host.</td>
</tr>
<tr>
<td>Destination port</td>
<td>Destination address of the receiving host.</td>
</tr>
<tr>
<td>Sequence number</td>
<td>Sequence number of the segment.</td>
</tr>
<tr>
<td>Acknowledgement number</td>
<td>Acknowledgement number of the segment.</td>
</tr>
<tr>
<td>Flags</td>
<td>Used to indicate the function of a segment.</td>
</tr>
<tr>
<td>SYN</td>
<td>Used during establishment.</td>
</tr>
<tr>
<td>FIN</td>
<td>Used during connection release.</td>
</tr>
<tr>
<td>RST</td>
<td>Used in case of problems.</td>
</tr>
<tr>
<td>ACK</td>
<td>If true, means that the Acknowledgement number inside the segment is valid.</td>
</tr>
<tr>
<td>Window</td>
<td>Window size.</td>
</tr>
<tr>
<td>Checksum</td>
<td>checked over the entire segment and part of the IP header.</td>
</tr>
<tr>
<td>Urgent pointer</td>
<td></td>
</tr>
<tr>
<td>Optional header extension</td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
</tbody>
</table>
TCP connection establishment

Three-way handshake
TCP connection establishment

Three-way handshake

CONNECT.req

Initial sequence number (x) read from TCP’s clock

SYN(seq=x)

CONNECT.ind
TCP connection establishment

Three-way handshake

CONNECT.req

Initial sequence number \( (x) \)
read from TCP’s clock

CONNECT.conf

\textit{Connection established}

SYN(seq=x)

SYN+ACK(ack=x+1,seq=y)

CONNECT.ind

CONNECT.resp

Initial sequence number \( (y) \)
read from TCP’s clock

Initial sequence number \( (x) \)
read from TCP’s clock

Connection established
TCP connection establishment

Three-way handshake

CONNECT.req

Initial sequence number \( (x) \)
read from TCP’s clock

CONNECT.conf

\textit{Connection established}

The sequence numbers of all segments \( A\rightarrow B \) will start at \( x+1 \)

\[ \text{SYN(seq}=x) \]

\[ \text{SYN+ACK(ack}=x+1,\text{seq}=y) \]

\[ \text{ACK(seq}=x+1,\text{ack}=y+1) \]

\[ \text{CONNECT.ind} \]

\[ \text{CONNECT.resp} \]

Initial sequence number \( (y) \)
read from TCP’s clock

\textit{Connection established}

The sequence numbers of all segments \( B\rightarrow A \) will start at \( y+1 \)

Remarks

Setting the SYN flag in a segment consumes one sequence number
The ACK flag is set only when the acknowledgement field contains a valid value
The default recommendation for the TCP clock is to be incremented by 1 at least after 4 microseconds and after each TCP connection establishment
TCP connection establishment (2)

Option negotiation
During the opening of a connection, it is possible to negotiate the utilisation of TCP extensions
Option encoded inside the optional part of TCP header
- Maximum segment size (MSS)
- RFC1323 timestamp extensions
- Selective Acknowledgments

MSS : 1460 bytes  MSS : 536 bytes (default)
TCP connection establishment (2)

Option negotiation
During the opening of a connection, it is possible to negotiate the utilisation of TCP extensions

Option encoded inside the optional part of TCP header
- Maximum segment size (MSS)
- RFC1323 timestamp extensions
- Selective Acknowledgments

CONNECT.req

MSS : 1460 bytes

SYN(seq=x)

MSS : 536 bytes (default)

CONNECT.ind

The remote host can only accept segments containing 1460 bytes of data in their payload
TCP connection establishment (2)

Option negotiation
During the opening of a connection, it is possible to negotiate the utilisation of TCP extensions

Option encoded inside the optional part of TCP header
- Maximum segment size (MSS)
- RFC1323 timestamp extensions
- Selective Acknowledgments

---

CONNECT.req

MSS : 1460 bytes

CONNECT.conf

The remote host can only accept segments containing 536 bytes of user data in their payload

Connection established

---

SYN(seq=x)

SYN+ACK(ack=x+1,seq=y)

MSS : 536 bytes (default)

CONNECT.ind
CONNECT.resp

The remote host can only accept segments containing 1460 bytes of data in their payload
TCP connection establishment (2)

Option negotiation
During the opening of a connection, it is possible to negotiate the utilisation of TCP extensions
Option encoded inside the optional part of TCP header
- Maximum segment size (MSS)
- RFC1323 timestamp extensions
- Selective Acknowledgments

CONNECT.req
- MSS : 1460 bytes

CONNECT.conf
- The remote host can only accept segments containing 536 bytes of user data in their payload

Connection established

SYN(seq=x)

SYN+ACK(ack=x+1, seq=y)

ACK(seq=x+1, ack=y+1)

Connection established

MSS : 536 bytes (default)

CONNECT.ind

CONNECT.resp
- The remote host can only accept segments containing 1460 bytes of data in their payload

Connection established
TCP connection establishment (3)

Rejection of connection establishment
TCP connection establishment (3)

Rejection of connection establishment

CONNECT.req → SYN(seq=x) → CONNECT.ind
TCP connection establishment (3)

Rejection of connection establishment

CONNECT.req

DISCONNECT.ind

Connection refused

SYN(seq=x)

RST+ACK(ack=x+1, seq=0)

CONNECT.ind

DISCONNECT.req
TCP connection establishment (3)

Rejection of connection establishment

A TCP entity should never send a RST segment upon reception of another RST segment
TCP connection establishment (4)

Simultaneous establishment
TCP connection establishment (4)

Simultaneous establishment

CONNECT.req → SYN(seq=x)
TCP connection establishment (4)

Simultaneous establishment

CONNECT.req → SYN(seq=x) → SYN(seq=y) → CONNECT.req

CONNECT.req
TCP connection establishment (4)

Simultaneous establishment

CONNECT.req

SYN(seq=x)

SYN(seq=y)

SYN+ACK(seq=x, ack=y+1)

CONNECT.conf

Connection established
TCP connection establishment (4)

Simultaneous establishment

CONNECT.req → SYN(seq=x) → SYN+ACK(seq=x, ack=y+1) → CONNECT.conf

CONNECT.req → SYN(seq=y) → SYN+ACK(seq=y, ack=x+1) → CONNECT.conf

Connection established → Connection established
TCP connection establishment (5)

Representation as a finite state machine

- Init
- SYN RCVD
- SYN Sent
- Established
TCP connection establishment (5)

Representation as a finite state machine

- Init
- SYN Sent
- SYN RCVD
- Established

Transition: !SYN
TCP connection establishment (5)

Representation as a finite state machine

- SYN RCVD
- Init
  - !SYN
  - SYN Sent
    - ?SYN+ACK / !ACK
    - Established
TCP connection establishment (5)

Representation as a finite state machine

Init

?SYN / !SYN+ACK
SYN RCVD

!SYN

SYN Sent

?SYN+ACK / !ACK

Established
TCP connection establishment (5)

Representation as a finite state machine

- **Init**
  - ?SYN / !SYN+ACK
  - !SYN
    - SYN Sent
    - ?SYN+ACK / !ACK
  - ?ACK
    - SYN RCVD
  - Established
TCP connection establishment (5)

Representation as a finite state machine

- Init
- SYN RCVD
- SYN Sent
- Established
TCP connection establishment (5)

Representation as a finite state machine

- SYN RCVD
- Init
- !SYN
- SYN Sent
- Established
TCP connection establishment (5)

Representation as a finite state machine

- Init
- SYN RCVD
- SYN Sent
- ?SYN / !SYN+ACK

Established
TCP connection establishment (5)

Representation as a finite state machine

- **Init**
  - !SYN
- **SYN Sent**
  - ?SYN / !SYN+ACK
  - !SYN
- **SYN RCVD**
  - ?SYN / !SYN+ACK
  - ?ACK
- **Established**
TCP connection establishment (6)

How to open several TCP connections at the same time?

Client: A

Server: S

Client: B

TCP connections on server

<table>
<thead>
<tr>
<th>IP local</th>
<th>IP remote</th>
<th>Port local</th>
<th>Port remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>A</td>
<td>80</td>
<td>1234</td>
</tr>
<tr>
<td>S</td>
<td>A</td>
<td>80</td>
<td>1235</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td>80</td>
<td>1235</td>
</tr>
</tbody>
</table>

TCP connections on host A

<table>
<thead>
<tr>
<th>IP locale</th>
<th>IP remote</th>
<th>Port local</th>
<th>Port remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S</td>
<td>1234</td>
<td>80</td>
</tr>
<tr>
<td>A</td>
<td>S</td>
<td>1235</td>
<td>80</td>
</tr>
</tbody>
</table>

TCP connections on host B

<table>
<thead>
<tr>
<th>IP locale</th>
<th>IP remote</th>
<th>Port local</th>
<th>Port remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>S</td>
<td>1235</td>
<td>80</td>
</tr>
</tbody>
</table>
Module 3 : Transport Layer

Basics

Building a reliable transport layer

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol
  TCP connection establishment
  TCP connection release
  Reliable data transfer
  Congestion control
TCP connection release

Graceful shutdown of a TCP connection
TCP connection release

Graceful shutdown of a TCP connection

1. DISCONNECT.req (A-B)
2. FIN(seq=x)
3. DISCONNECT.ind(A-B)
4. incoming connection closed
TCP connection release

Graceful shutdown of a TCP connection

DISCONNECT.req (A-B)  →  FIN(seq=x)  →  DISCONNECT.ind(A-B)  

DISCONNECT.conf(A-B)  →  ACK(ack=x+1)  

outgoing connection closed

incoming connection closed
TCP connection release

Graceful shutdown of a TCP connection

DISCONNECT.req (A-B) → FIN(seq=x)
→ DISCONNECT.ind(A-B)
→ ACK(ack=x+1)
→ DISCONNECT.conf(A-B)

outgoing connection closed

DISCONNECT.conf(A-B) → FIN(seq=x)

incoming connection closed

DISCONNECT.ind(B-A) → FIN(seq=y)

DISCONNECT.req(B-A) → incoming connection closed
TCP connection release

Graceful shutdown of a TCP connection

DISCONNECT.req (A-B)

FIN(seq=x)

ACK(ack=x+1)

DISCONNECT.ind(A-B)

incoming connection closed

DISCONNECT.conf(A-B)

outgoing connection closed

DISCONNECT.ind(B-A)

FIN(seq=y)

ACK(ack=y+1)

DISCONNECT.req(B-A)

incoming connection closed

DISCONNECT.conf(A-B)

outgoing connection closed
TCP connection release

Graceful shutdown of a TCP connection

1. DISCONNECT.req (A-B)
   - DISCONNECT.conf(A-B)
     - outgoing connection closed
2. FIN(seq=x)
   - ACK(ack=x+1)
     - DISCONNECT.ind(A-B)
       - incoming connection closed
3. ACK(ack=y+1)
   - DISCONNECT.conf(A-B)
4. DISCONNECT.req(B-A)
   - DISCONNECT.ind(B-A)
     - incoming connection closed
5. FIN(seq=y)
   - ACK(ack=y+1)
     - outgoing connection closed

Time WAIT
Maintain state for this connection during twice MSL to be able to retransmit ACK if a segment is received from the other entity

State can be removed
Abrupt TCP connection release
Abrupt TCP connection release

- DISCONNECT.req (abrupt)
- Connection closed
- RST(seq=x)
- DISCONNECT.ind(abrupt)
- Connection closed
Abrupt TCP connection release

DICOM format:

- DISCONNECT.req (abrupt)
- Connection closed
- State can be removed

- RST(seq=x)

- DISCONNECT.ind (abrupt)
- Connection closed
- State can be removed
Abrupt TCP connection release

Data segments can be lost during such an abrupt release.
No entity needs to wait in TIME_WAIT state after such a release anyway, any segment received when there is no state causes the transmission of a RST segment.
TCP connection release

SYN RCVD

!FIN

FIN Wait1

?FIN/!ACK

?ACK

FIN Wait2

?FIN/!ACK

TIME Wait

?ACK

Closing

Established

CLOSE Wait

LAST-ACK

Closed

?FIN/!ACK

!FIN

?ACK
TCP connection release

SYN RCVD

!FIN

FIN Wait1

?ACK

FIN Wait2

?FIN/!ACK

Closing

?ACK

TIME Wait

?FIN/!ACK

Established

CLOSE Wait

LAST-ACK

Closed

Timeout[2MSL]
TCP connection release

- **SYN RCVD**: !FIN
  - **FIN Wait1**: ?ACK, ?FIN/?ACK
    - **FIN Wait2**: ?FIN/?ACK
  - **TIME Wait**: ?ACK
    - **TIME Wait Closed**: !FIN, Timeout[2MSL]
- **Established**: !FIN
  - **CLOSE Wait**: !FIN
    - **LAST-ACK**: ?ACK
      - **Closed**
Module 3 : Transport Layer

Basics

Building a reliable transport layer

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol
  
  TCP connection establishment
  TCP connection release
  
  Reliable data transfer
  Congestion control
Reliable data transfer

Each TCP segment contains

- **16 bits checksum**
  - used to detect transmission errors affecting payload

- **32 bits sequence number** (one byte=one seq. number)
  - used by sender to delimitate sent segments
  - used by receiver to reorder received segments

- **32 bits acknowledgement number**
  - used (when ACK flag is 1) by receiver to advertise the sequence number of the next expected byte *(last byte received in sequence+1)*
Reliable data transfer

Each TCP segment contains
- **16 bits checksum**
  - used to detect transmission errors affecting payload
- **32 bits sequence number** (one byte = one seq. number)
  - used by sender to delimitate sent segments
  - used by receiver to reorder received segments
- **32 bits acknowledgement number**
  - used (when ACK flag is 1) by receiver to advertise the sequence number of the next expected byte (last byte received in sequence + 1)

DATA.req ("abcd") → (seq=123,"abcd") → DATA.ind("abcd")
Reliable data transfer

Each TCP segment contains
16 bits checksum
used to detect transmission errors affecting payload
32 bits sequence number (one byte=one seq. number)
used by sender to delimitate sent segments
used by receiver to reorder received segments
32 bits acknowledgement number
used (when ACK flag is 1) by receiver to advertise the sequence number of the next expected byte (last byte received in sequence+1)

DATA.req ("abcd")
(seq=123,"abcd")
(ack=127)
DATA.ind("abcd")
Each TCP segment contains
- 16 bits **checksum** used to detect transmission errors affecting payload
- **32 bits sequence number** (one byte=one seq. number) used by sender to delimitate sent segments used by receiver to reorder received segments
- **32 bits acknowledgement number** used (when ACK flag is 1) by receiver to advertise the sequence number of the next expected byte (last byte received in sequence+1)

```
DATA.req ("abcd")
DATA.ind("abcd")
(seq=123,"abcd")
(ack=127)
(seq=127,"efg")
DATA.req ("efg")
DATA.ind("abcd")
```
Reliable data transfer

Each TCP segment contains

- **16 bits checksum**
  - used to detect transmission errors affecting payload

- **32 bits sequence number** (one byte = one seq. number)
  - used by sender to delimitate sent segments
  - used by receiver to reorder received segments

- **32 bits acknowledgement number**
  - used (when ACK flag is 1) by receiver to advertise the sequence number of the next expected byte (last byte received in sequence + 1)

```
DATA.req ("abcd")
DATA.req ("efg")
DATA.req ("hi")
```

```
(seq=123,"abcd")
(ack=127)
(seq=127,"efg")
(seq=130,"hi")
```

```
DATA.ind("abcd")
```
Reliable data transfer

Each TCP segment contains

- **16 bits checksum**
  used to detect transmission errors affecting payload

- **32 bits sequence number** (one byte = one seq. number)
  used by sender to delimitate sent segments
  used by receiver to reorder received segments

- **32 bits acknowledgement number**
  used (when ACK flag is 1) by receiver to advertise the sequence
  number of the next expected byte (last byte received in sequence + 1)

DATA.req ("abcd")
DATA.ind("abcd")
(seq=123,"abcd")

DATA.req ("efg")
(seq=127,"efg")
(ack=127)

DATA.req ("hi")
(seq=130,"hi")

DATA.req ("jkl")
(seq=132,"jkl")

DATA.req ("jkl")
Each TCP segment contains:

- **16 bits checksum**
  - Used to detect transmission errors affecting payload

- **32 bits sequence number** (one byte = one seq. number)
  - Used by sender to delimitate sent segments
  - Used by receiver to reorder received segments

- **32 bits acknowledgement number**
  - Used (when ACK flag is 1) by receiver to advertise the sequence number of the next expected byte (last byte received in sequence + 1)

```
DATA.req ("abcd")
DATA.req ("efg")
DATA.req ("hi")
DATA.req ("jkl")
```

```
(seq=123,"abcd")
(ack=127)
(seq=127,"efg")
(seq=130,"hi")
(seq=132,"jkl")
(ack=135)
```

```
DATA.ind("abcd")
DATA.ind("efghijkl")
DATA.ind("efghijkl")
```
How to deal with segment losses?
TCP uses a retransmission timer.
If the retransmission timer expires, TCP performs go-back-n and retransmits all the unacknowledged segments usually a single retransmission timer is running at a given time.
Reliable data transfer

How to deal with segment losses?
TCP uses a retransmission timer
If the retransmission timer expires, TCP performs go-back-n and retransmits all the unacknowledged segments
usually a single retransmission timer is running at a given time

(seq=123,"abcd")
Reliable data transfer

How to deal with segment losses?
TCP uses a retransmission timer
If the retransmission timer expires, TCP performs go-back-n and retransmits all the unacknowledged segments usually a single retransmission timer is running at a given time

```
(seq=127,"ef")
(seq=123,"abcd")
(ack=123)
```

"ef" placed in buffer
How to deal with segment losses?  
TCP uses a retransmission timer  
If the retransmission timer expires, TCP performs go-back-n and retransmits all the unacknowledged segments  
usually a single retransmission timer is running at a given time.
Reliable data transfer

How to deal with segment losses?
TCP uses a retransmission timer
If the retransmission timer expires, TCP performs go-back-n and retransmits all the unacknowledged segments
usually a single retransmission timer is running at a given time

Retransmission timer

Retransmission of all unacked segments

(seq=123,"abcd")
(seq=127,"ef")
(ack=123)
(seq=123,"abcd")
(ack=129)

"ef" placed in buffer

"abcdef"
Reliable data transfer

How to deal with segment losses?
TCP uses a retransmission timer
If the retransmission timer expires, TCP performs go-back-n and retransmits all the unacknowledged segments usually a single retransmission timer is running at a given time

Retransmission timer
Retransmission of all unacked segments

(seq=123,"abcd")
(seq=127,"ef")
(ack=123)
(seq=123,"abcd")
(seq=127,"ef")
(ack=129)
(ack=129)

"ef" placed in buffer
"abcdef"
unnecessary retransmission
Retransmission timer

How to compute it?

Issue
round-trip-time may change frequently during the lifetime of a TCP connection
Retransmission timer

TCP’s retransmission timer
One timer per connection

timer = mean(rtt) + 4*std_dev(rtt)

Estimation of the mean
est_mean(rtt) = (1-\alpha)\times est_mean(rtt) + \alpha \times rtt_{measured}

Estimation of the standard deviation of the rtt
est_std_dev = (1-\beta)\times est_std_dev + \beta \times |rtt_{measured} - est_mean(rtt)|

![Graph of RTT vs N with different lines for measured rtt, mean rtt, and timeout.](image-url)
Round-trip-time estimation

Problem
How to measure rtt after retransmissions?

Solution (Karn/Partridge)
1. Do not measure rtt of retransmitted segments
Round-trip-time estimation (2)

Improvement to Karn/Partridge
Add a timestamp in each segment sent
TS and TSEcho (RFC1323)

(seq=120, TS=1, TS echo=7, "xyz")
(ack=123, TS=12, TS echo=1)
(seq=123, TS=3, TS echo=12, "abcd")
(seq=123, TS=5, TS echo=12, "abcd")
(ack=127, TS=17, TS echo=3)
Improving the reliable data transfer

How to improve the reaction to segment losses?
TCP receiver should send an ack everytime an out-of-sequence segment is received

Heuristic: a segment is considered lost after three duplicate segments
Improving the reliable data transfer

How to improve the reaction to segment losses? TCP receiver should send an ack everytime an out-of-sequence segment is received.

**Heuristic**: a segment is considered lost after three duplicate segments.

(seq=120,"xyz")

(ack=123)
How to improve the reaction to segment losses?
TCP receiver should send an ack everytime an out-of-sequence segment is received.

**Heuristic**: a segment is considered lost after three duplicate segments:

- (seq=123, "abcd")
- (seq=120, "xyz")
- (ack=123)

Diagram: A sequence diagram showing the sequence of segments and acknowledgments.
Improving the reliable data transfer

How to improve the reaction to segment losses?
TCP receiver should send an ack every time an out-of-sequence segment is received.

Heuristic: A segment is considered lost after three duplicate segments.

First duplicate ack

(ack=123)

(seq=123, "abcd")

(seq=120, "xyz")

(seq=127, "ef")

Out of sequence
Improving the reliable data transfer

How to improve the reaction to segment losses?
TCP receiver should send an ack everytime an out-of-sequence segment is received.

Heuristic: a segment is considered lost after three duplicate segments.

```
(seq=120,"xyz")
(ack=123)
(seq=123,"abcd")
(seq=127,"ef")
(ack=123)
(seq=129,"gh")
(ack=123)
```

First duplicate ack
Out of sequence

Second duplicate ack
Out of sequence
Improving the reliable data transfer

How to improve the reaction to segment losses?
TCP receiver should send an ack everytime an out-of-sequence segment is received

Heuristic: a segment is considered lost after three duplicate segments

First duplicate ack
(ack=123)
(seq=120,"xyz")
(seq=123,"abcd")
(seq=127,"ef")
(ack=123)

Second duplicate ack
(ack=123)
(seq=129,"gh")
(ack=123)
(seq=131,"ij")
(ack=123)

Third duplicate ack
(ack=123)

Out of sequence
Out of sequence
Out of sequence
Improving the reliable data transfer

How to retransmit the lost segments

Upon reception of three duplicate acks, retransmit the first unacked segment

Fast retransmit, used by most TCP implementations
Improving the reliable data transfer

How to retransmit the lost segments
Upon reception of three duplicate acks, retransmit the first unacked segment
Fast retransmit, used by most TCP implementations

(ack=123)
How to retransmit the lost segments
Upon reception of three duplicate acks, retransmit the first unacked segment
Fast retransmit, used by most TCP implementations

(seq=123, "abcd")

(ack=123)
Improving the reliable data transfer

How to retransmit the lost segments
Upon reception of three duplicate acks, retransmit the first unacked segment
Fast retransmit, used by most TCP implementations
Improving the reliable data transfer

How to retransmit the lost segments
Upon reception of three duplicate acks, retransmit the first unacked segment
Fast retransmit, used by most TCP implementations

![Diagram showing sequence of acknowledgement and retransmission](image)
How to retransmit the lost segments
Upon reception of three duplicate acks, retransmit the first unacked segment
Fast retransmit, used by most TCP implementations
Improving the reliable data transfer

How to retransmit the lost segments
Upon reception of three duplicate acks, retransmit the first unacked segment
Fast retransmit, used by most TCP implementations
Improving the reliable data transfer

Selective acknowledgement

sack: [seq1-seq2]; [seq3-seq4]

(ack=123)

(ack=123, sack:127-128)

(ack=123, sack:127-130)

(ack=123, sack:127-132)

(ack=123)

only 123-126 must be retransmitted

"abcdefgij"

"abcdefghij"
Sending acknowledgements

When to send a pure ack?
upon reception of a data segment
inside data segments in the other direction
(piggyback)

TCP tradeoff
Insequence segment arrival
If there is no ack waiting to be sent, start ack timer (50 msec) and wait until
transmission of a data segment (piggyback)
expiration of acktimer
If there is already an ack waiting
send pure ack immediately

Out-of-sequence segment
send ack immediately
Flow control

Goal: protect the receiver’s buffers

Principle
- Advertise receiving window in all segments
- State variables maintained by each TCP entity
  - last_ack, swin, rwin

Last_ack=122, swin=100, rwin=4
To transmit: abcdefghijklm
Flow control

Goal: protect the receiver’s buffers

Principle
Advertise receiving window in all segments
State variables maintained by each TCP entity
last_ack, swin, rwin

Last_ack=122, swin=100, rwin=4
To transmit: abcdefghijklm

Last_ack=122, swin=96, rwin=0

Last_ack=126, swin=100, rwin=0
Flow control

Goal: protect the receiver’s buffers

Principle
Advertise receiving window in all segments
State variables maintained by each TCP entity
last_ack, swin, rwin

Last_ack=122, swin=100, rwin=4
To transmit: abcdefghijklm

Last_ack=122, swin=96, rwin=0

Last_ack=126, swin=100, rwin=0
Last_ack=126, swin=100, rwin=2
Last_ack=128, swin=100, rwin=20
Flow control

Goal: protect the receiver’s buffers

Principle
Advertise receiving window in all segments
State variables maintained by each TCP entity
last_ack, swin, rwin

Last_ack=122, swin=100, rwin=4
To transmit: abcdefghijklm

Last_ack=122, swin=96, rwin=0

Last_ack=126, swin=100, rwin=0
Last_ack=126, swin=100, rwin=2
Last_ack=126, swin=98, rwin=0

Last_ack=128, swin=100, rwin=20
Last_ack=128, swin=93, rwin=13
Last_ack=135, swin=100, rwin=20

© O. Bonaventure, 2008
Limitations

TCP uses a 16 bits window field in the segment header
- Maximum window size for normal TCP: 65535 bytes
- Extension RFC1323 for larger windows

After having transmitted a window full of data, TCP sender must remain idle waiting for ack

Maximum throughput on TPC connection
\[ \sim \text{window} / \text{round-trip-time} \]

<table>
<thead>
<tr>
<th></th>
<th>rtt 1 msec</th>
<th>10 msec</th>
<th>100 msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 Kbytes</td>
<td>65.6 Mbps</td>
<td>6.5 Mbps</td>
<td>0.66 Mbps</td>
</tr>
<tr>
<td>64 Kbytes</td>
<td>524.3 Mbps</td>
<td>52.4 Mbps</td>
<td>5.2 Mbps</td>
</tr>
</tbody>
</table>
Transmission of data and control segments

Nagle algorithm
A new data segment can be sent provided that
This is a maximum sized segment (MSS bytes)
There are currently no unacknowledged bytes

Consequence
Most TCP/IP packets are small or MSS-sized
Module 3 : Transport Layer

Basics

Building a reliable transport layer

UDP : a simple connectionless transport protocol

TCP : a reliable connection oriented transport protocol
  - TCP connection establishment
  - TCP connection release
  - Reliable data transfer
  - Congestion control
TCP/IP networks are heterogeneous
A can send at 10 Mbps to B
B can send at 2 Mbps to C

How to share the network among multiple hosts?
A and B send data to C at the same time
Congestion in TCP/IP networks

Possible solutions
The network indicates explicitly the bandwidth allocated to each host
network sends regularly control information to hosts
Available Bit Rate in ATM networks

Endhosts measure the state of the network and adapt their bandwidth to the network state
Endhosts must be able to measure the amount of congestion inside the network
Solution used by TCP in the Internet
Simple congestion
Simple congestion

Simplified model
Simple congestion

Infinite buffers

Simplified model
Simple congestion

10 Mbps

2 Mbps

TCP self-clocking
Simple congestion

TCP self-clocking
Can be sufficient when a single TCP connection uses a low bandwidth link if the intermediate buffer can store a window full of segments
What happens if several TCP connections need to share one link

Many senders $\rightarrow$ $10 \text{ Mbps}$ infinite buffers $\rightarrow$ $2 \text{ Mbps}$ Many receivers

Occupation du Buffer

$\rightarrow$ Congestion collapse!
TCP congestion control

How to adapt a TCP connection to the network state?

How to measure the current congestion state?
TCP uses segment losses in routers as an implicit indication of congestion.
This is valid in most environments besides some wireless networks where transmission errors can cause segment losses.

Adapt the bandwidth of the TCP connection
TCP adapts its transmission rate by using a new congestion window (cwnd) which is controlled by the sender based on the current congestion status.
A simple router

Output buffer

Input link1
Input link2
Input link3

Buffer acceptance algorithm
When a packet arrives in the output buffer, decides whether the packet is accepted or discarded

taildrop
the arriving packet is discarded if the buffer is full
**TCP congestion control**

Segment transmission
a new TCP segment can be sent if it is contained inside the current window:
\[
\text{[ last_ack, last_ack + min(cwnd, min(swin, rwin)) ]}
\]

Segment reception
send acks as usual and use TCP flow control

<table>
<thead>
<tr>
<th>Cwnd[B]</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>cwnd[A]</td>
<td>underused</td>
<td>underused</td>
<td>ok, but unfair</td>
</tr>
<tr>
<td>1</td>
<td>underused</td>
<td>ok, fair</td>
<td>congestion</td>
</tr>
<tr>
<td>2</td>
<td>ok, but unfair</td>
<td>congestion</td>
<td>congestion</td>
</tr>
</tbody>
</table>

How to dynamically update cwnd?
TCP congestion control

Additive Increase / Multiplicative Decrease

Segment transmission
a new TCP segment can be sent if it is contained inside the current window:
\[ \text{last_ack} , \text{last_ack} + \min(\text{cwnd}, \min(\text{swin}, \text{rwin})) \]

Ack reception: **Additive Increase**
// congestion avoidance
if (network not congested) // no segment losses
{ // increase slowly cwnd
   // increase cwnd by one mss every rtt
   \text{cwnd} = \text{cwnd} + \text{mss} \times \left( \frac{\text{mss}}{\text{cwnd}} \right);
}

Congestion: Segment loss: **Multiplicative decrease**
// retransmit lost segments
\text{ssthresh} = \max(\text{cwnd}/2, 2 \times \text{MSS});
\text{cwnd} = \text{MSS};

Segment reception
send acks as usual and use TCP flow control
TCP congestion control

How to select cwnd when connection starts?
Congestion avoidance increases cwnd slowly

Initialisation:
cwnd = MSS;
ssthresh = swin;

Ack reception:
if (network not congested) // no segment losses
{
  if (cwnd < ssthresh)
    // increase quickly cwnd
    // double cwnd every rtt
    cwnd = cwnd + MSS;
  else
    // increase slowly cwnd
    // increase cwnd by one mss every rtt
    cwnd = cwnd + mss * (mss / cwnd);
}

Slowstart

Congestion avoidance
TCP congestion control

Example

- **Slow-start**
  - exponential increase of cwnd

- **Congestion avoidance**
  - linear increase of cwnd

**Cwnd**

- Timer expiration
- Threshold

**Time**

© O. Bonaventure, 2008
TCP congestion control

How to react to segment losses?
Two different types of multiplicative decrease

Severe loss [several lost segments]
wait until expiration of retransmission timer
  sstresh = ssthresh/2
retransmit lost segments
slow-start (until cwnd = ssthresh)
congestion avoidance

Isolated loss [a single lost segment]
  fast retransmit can recover from lost segment
If a single segment was lost: fast recovery
  retransmit lost segment
  sstresh = cwnd / 2
  cwnd = ssthresh ; congestion avoidance
TCP congestion control

Simplified model
Assume that all segment losses are periodic and the every 1/p segment is lost

\[
\begin{align*}
\text{Cwnd(segments)} = & \ W \\
\text{time(rtt)} = & \ 0 \ 0 \ W/2 \ W \ 3W/2 \ 2W
\end{align*}
\]

Surface

\[
\left(\frac{W}{2}\right)^2 + \frac{1}{2} \left(\frac{W}{2}\right)^2 = \frac{1}{p}
\]

It can be shown that the throughput of a TCP connection can be approximated by:

\[
BW < \min\left[\frac{\text{Window}}{\text{RTT}}, \left(\frac{\text{MSS}}{\text{RTT}}\right)\frac{k}{\sqrt{p}}\right]
\]

Maximum throughput without losses

Throughput with losses/congestion

© O. Bonaventure, 2008