UCL Université catholique de Louvain

# Computer Networks : Protocols and Practice

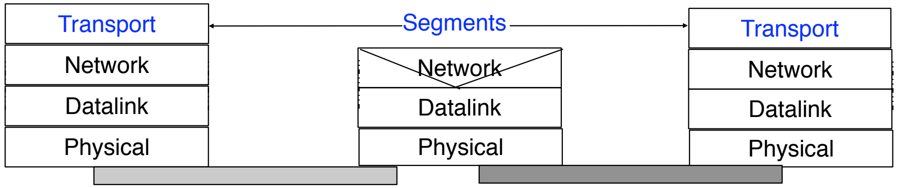
## Part 3 : Transport Layer

Olivier Bonaventure
http://inl.info.ucl.ac.be/

# Module 3 : Transport Layer

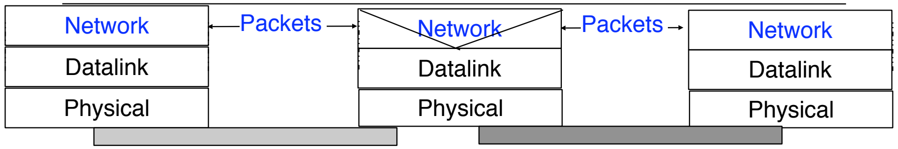
- Basics
- Building a reliable transport layer
  - Reliable data transmission
  - Connection establishment
  - Connection release
- UDP : a simple connectionless transport protocol
- TCP : a reliable connection oriented transport protocol

# 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



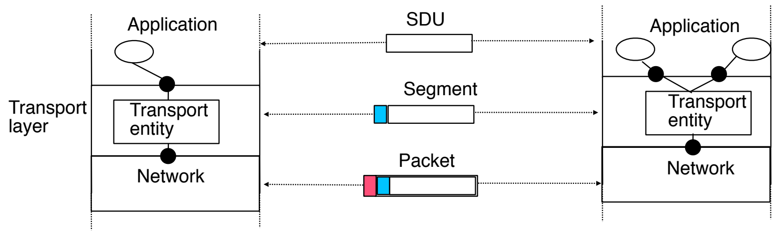

- Network layer service in Internet
  - Unreliable connectionless service
    - Packets can be lost
    - Packets can suffer from transmission errors
    - Packet ordering is not preserved
    - Packet can be duplicated
    - Packet size is limited to about 64 KBytes
  - How to build a service useable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

# The transport layer (2)

- Internal organisation
  - The transport layer uses the service provide by the network layer
  - Two transport layer entities exchanges segments



# Module 3 : Transport layer

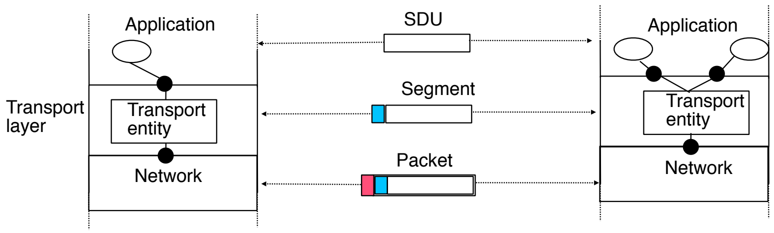
- Basics
- Building a reliable transport layer
  - Reliable data transmission
  - Connection establishment
  - Connection release
- UDP : a simple connectionless transport protocol
- TCP : a reliable connection oriented transport protocol

# 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

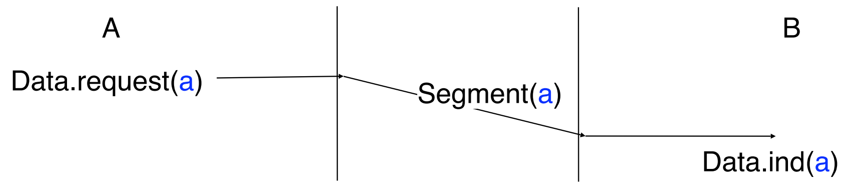
# Transport layer protocols (2)

- Reference environment




- Notations
  - `data.req` and `data.ind` primitives for application/ transport interactions
  - `recv()` and `send()` for interactions between transport entity and network layer

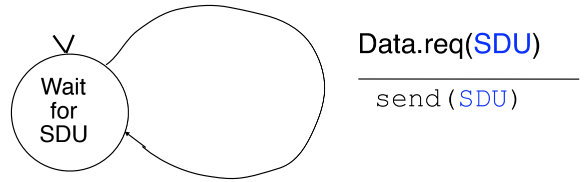
# Protocol 1 : Basics



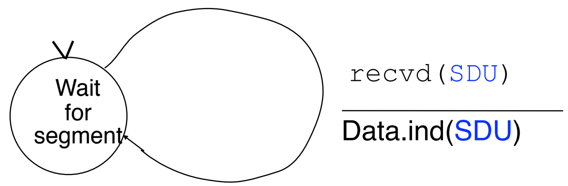

- Principle
  - Upon reception of `data.request(SDU)`, transport entity sends a segment containing this SDU through the network layer (`send(SDU)`)
  - Upon reception of the contents of one packet from the network layer (`recv(SDU)`), transport entity delivers the SDU found in the packet to its user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

# Protocol 1 as a FSM

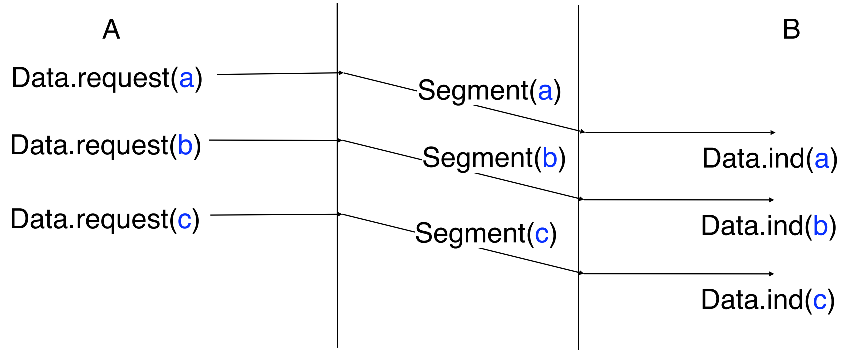
- Sender




- Receiver



# Protocol 1 : Example

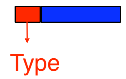



- Issue
  - What happens if the receiver is much slower than the sender ?
    - e.g. receiver can process one segment per second while sender is producing 10 segments per second ?

# 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

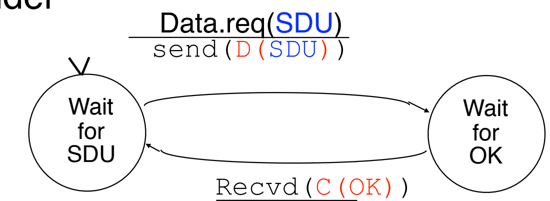
- Consequences
  - Two types of segments
    - Data segment containing on SDU
      - Notation : D(SDU)
    - Control segment
      - Notation : C(OK)



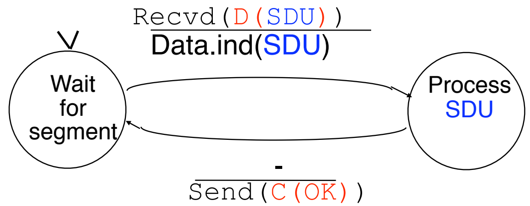

  - Segment format
    - At least one bit in the segment header is used to indicate the type of segment

# Protocol 2 (cont.)

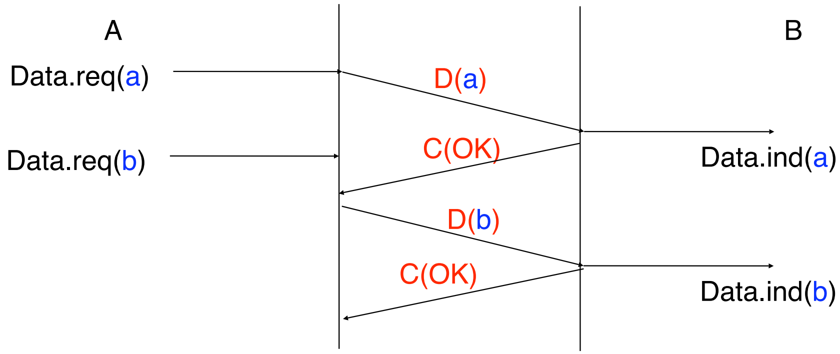
- Sender




- Receiver



# Protocol 2 : Example




The sender only sends segments when authorised by the receiver

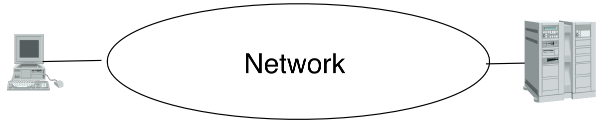
# Protocol 3

- 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

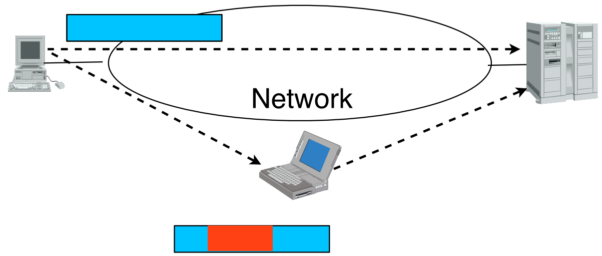
- Which types of transmission errors do we need to consider in the transport layer ?




- 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




- These attacks are dealt by using special security protocols and mechanisms outside the transport layer

## How to detect transmission errors ?

- Principle
  - 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

      0011010 0         11011001
- 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

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 * 2^r XOR R = n*G

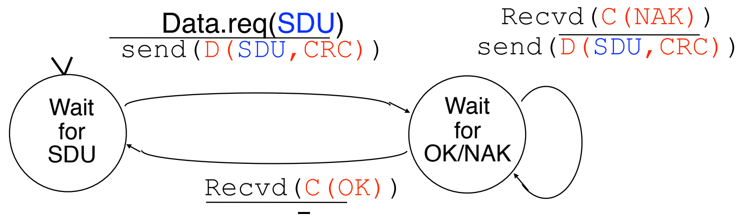

    - All computations are done in modulo 2 arithmetic by using XOR
      - 1011 + 0101 = 1110 1001 + 1101 = 0100
      - 1011 - 0101 = 1110 1001 - 1101 = 0100

# 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

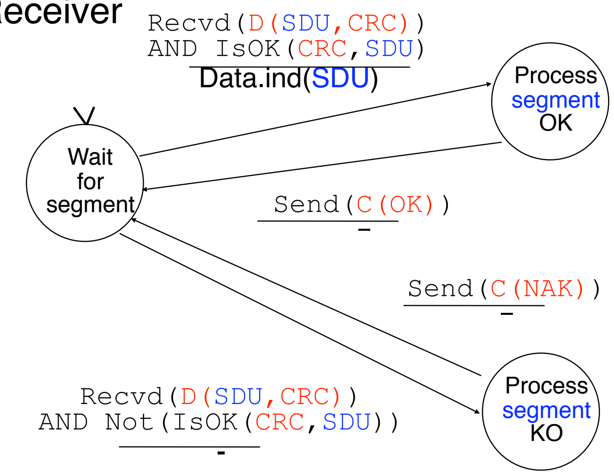
# Protocol 3a : Sender

- Sender

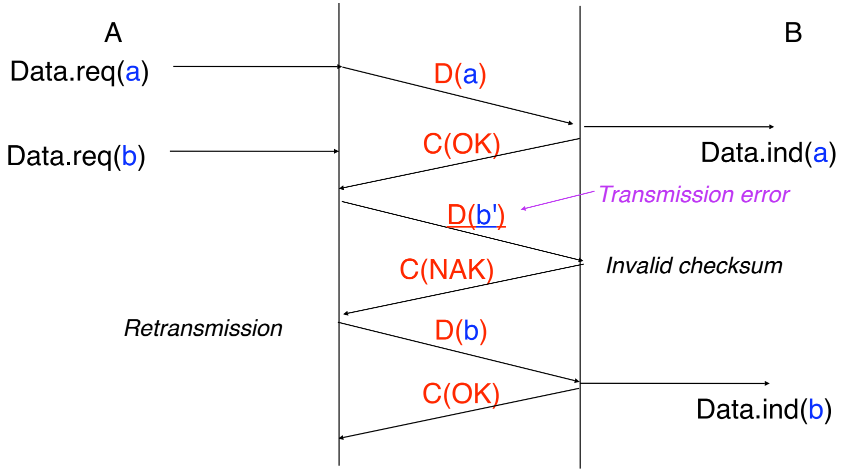


# Protocol 3a : Receiver

- Receiver



# Protocol 3a : Example



# Protocol 3b

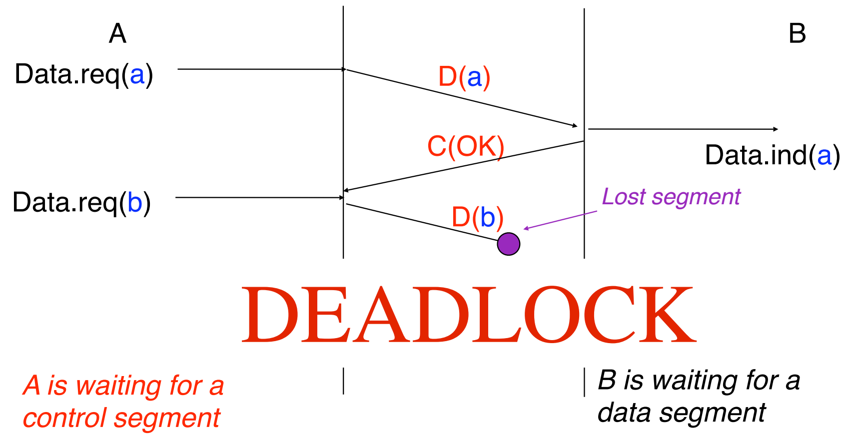
- How can we provide a reliable service in the transport layer ?

  - 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 ?

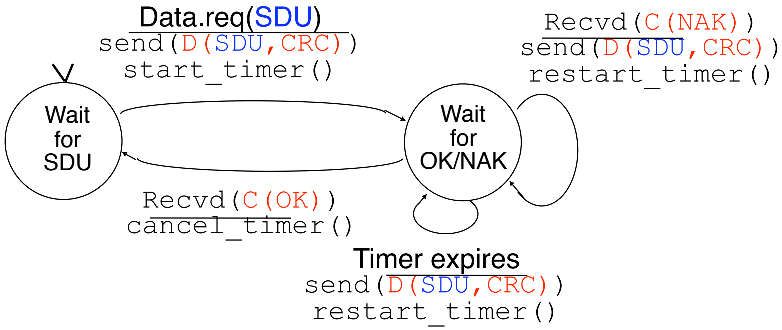
# Protocol 3a and segment losses

- How do segment losses affect protocol 3a ?



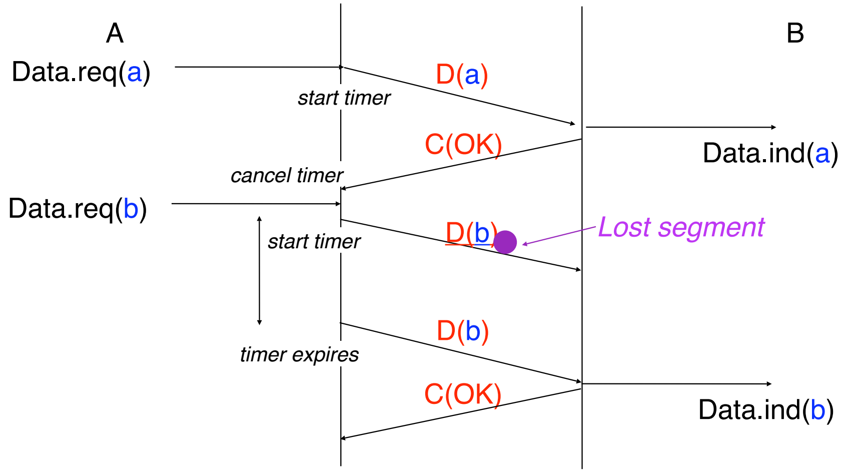
# Protocol 3b

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



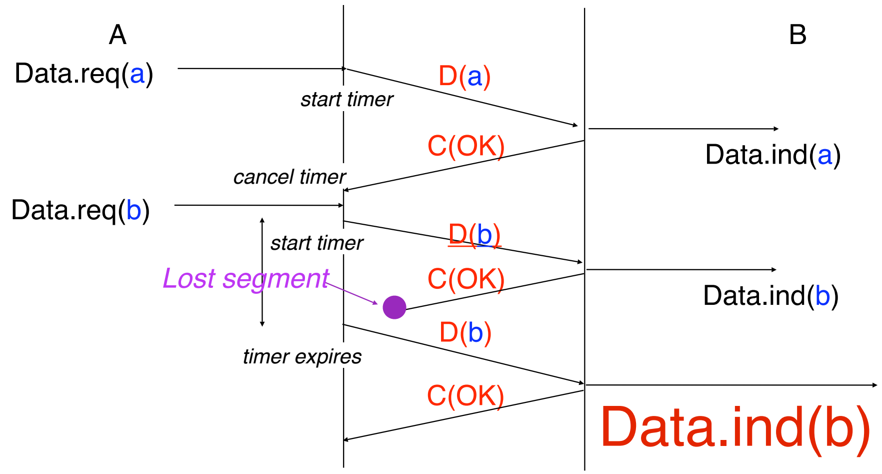

  - No modification to the receiver

# Protocol 3b : Example




- Does this protocol always work ?

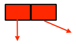
# Protocol 3b : Example




- How to solve this problem ?

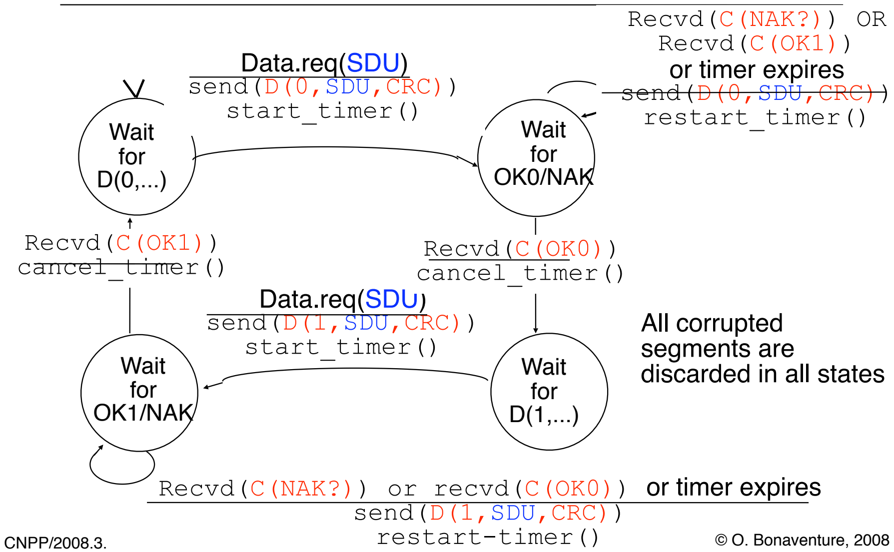
# Alternating bit protocol

- 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

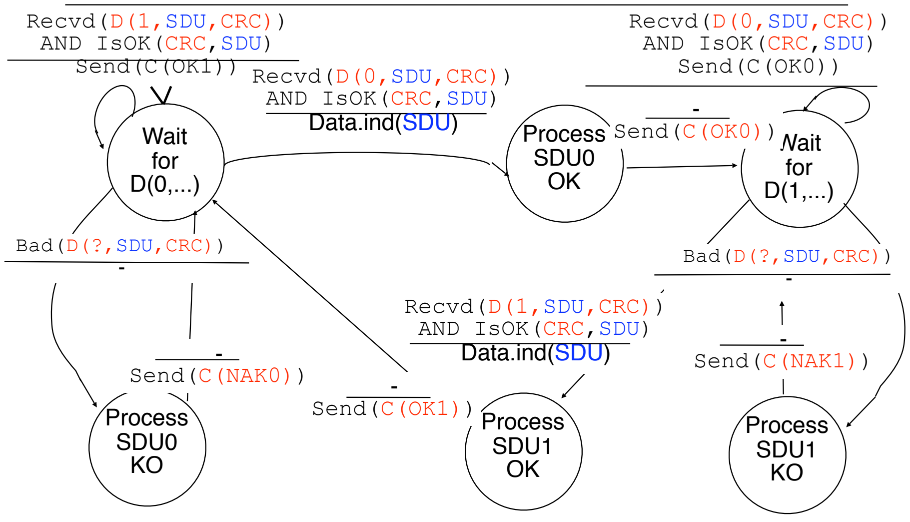




  - How many bits do we need for the sequence number?
    - a single bit is enough

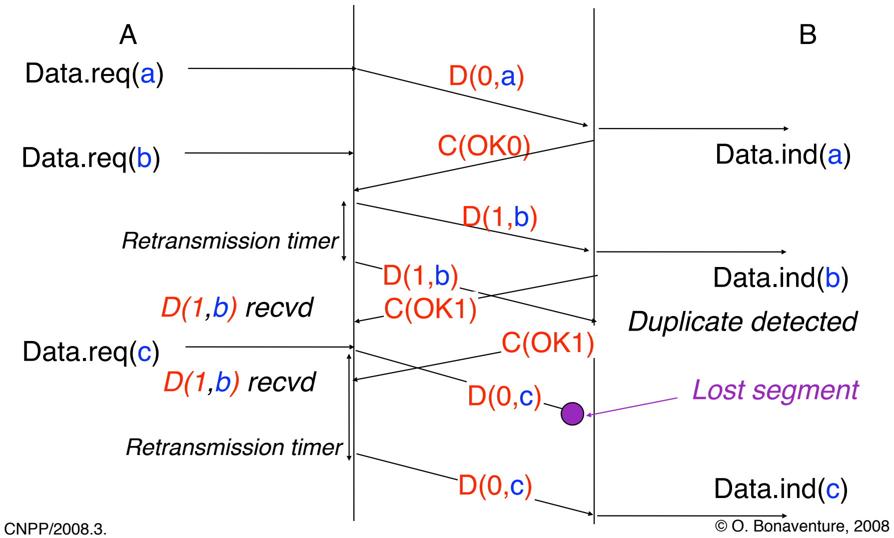
# Alternating bit protocol
# Sender



# Alternating bit protocol Receiver

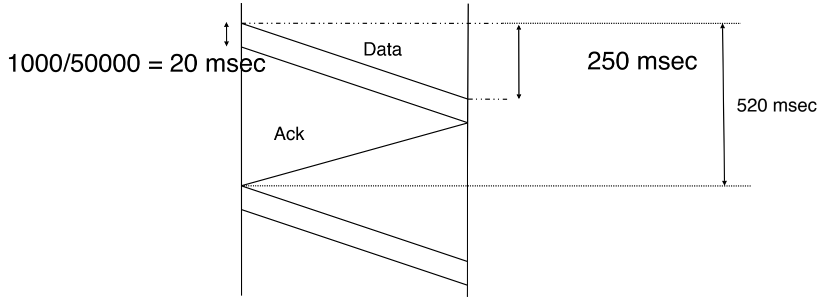


# Alternating bit protocol Example



# Performance of the alternating bit protocol

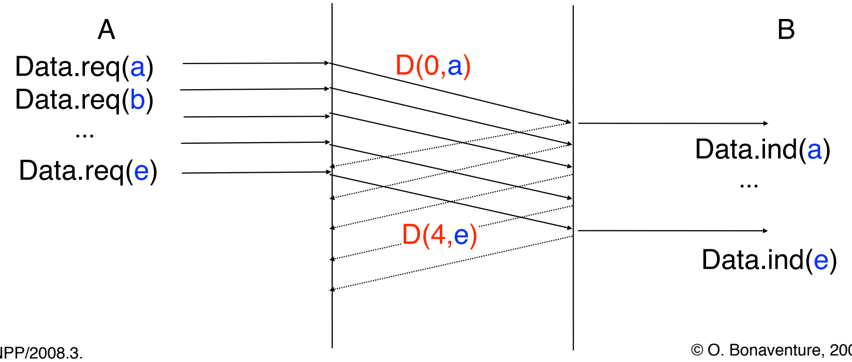
- What is the performance of the ABP in this case
  - One-way delay : 250 msec
  - Physical layer throughput : 50 kbps
  - segment size : 1000 bits




- -> Performance is function of *bandwidth * 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



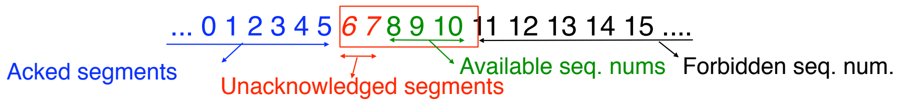
# How to improve the alternating bit protocol ? (2)

- 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
    - `sending_window`

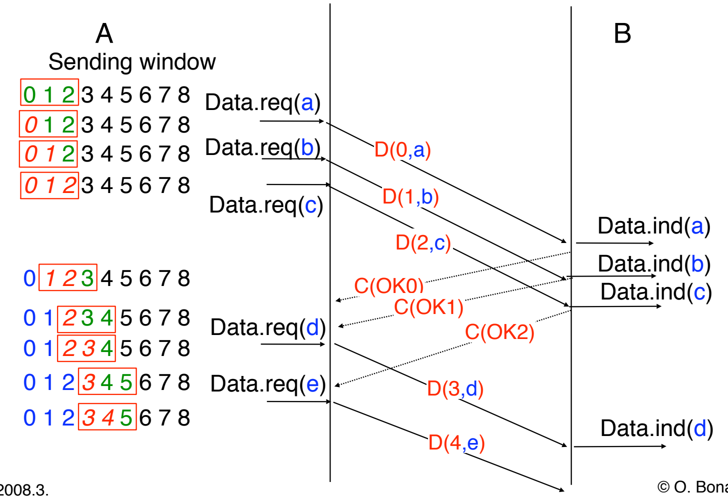



  - Receiver also maintains a receiving window with the list of acceptable sequence number
    - receiving_window

  - Sender and receiver must use compatible windows
    - sending_window ≤ 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

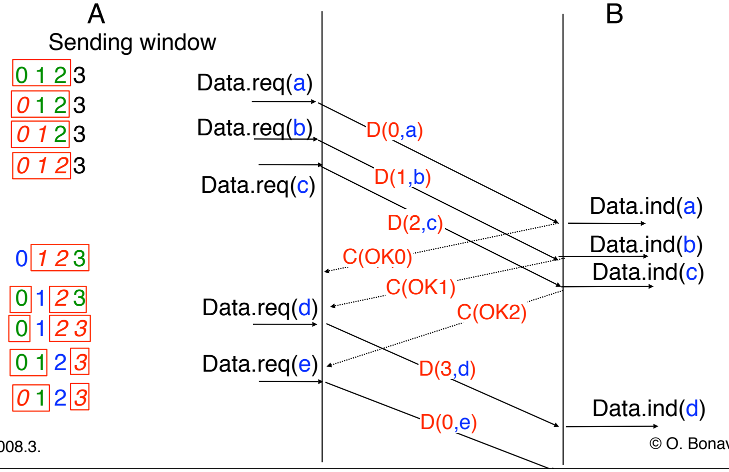


# Encoding sequence numbers

- 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 seqgment 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




CNPP/2008.3.

# Reliable transfer with a sliding window

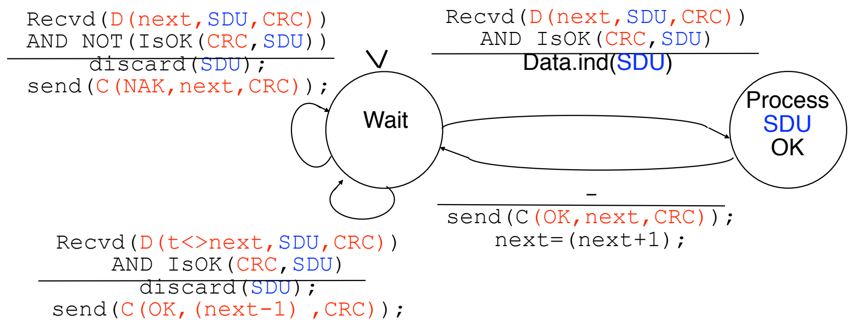
- How to provide a reliable data transfer with a sliding window
  - How to react upon reception of a control segment ?
  - Sender's and receiver's behaviours
- Basic solutions
  - Go-Back-N
    - simple implementation, in particular on receiving side
    - throughput will be limited when losses occur
  - 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




CNPP/2008.3.

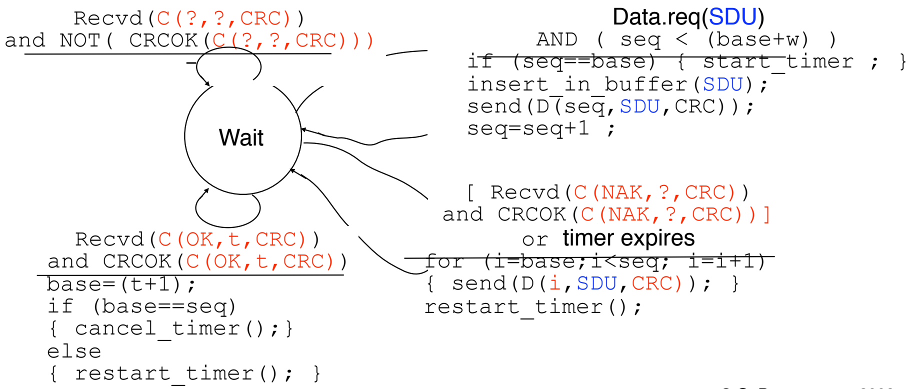




To avoid overloading the slide, we assume that sequence numbers are integers. Of course, all computations must be performed modulo $2^N$.

# Go-Back-N : Sender

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



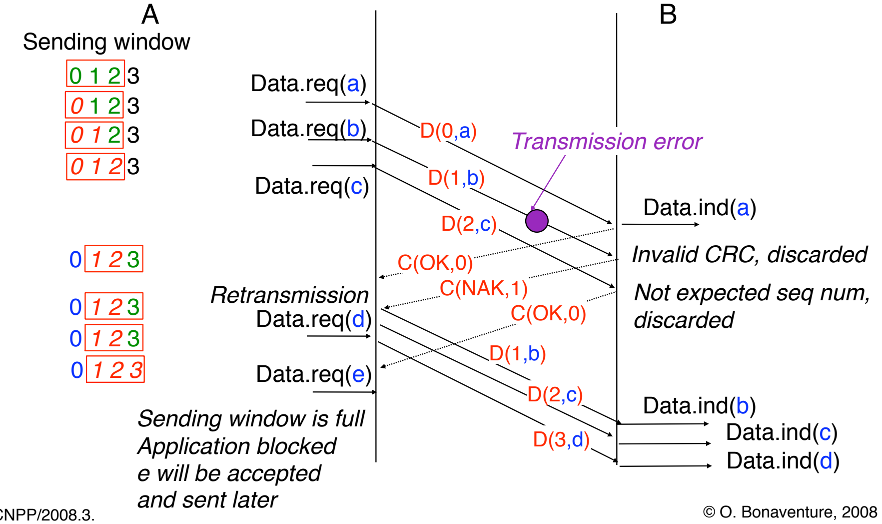

CNP



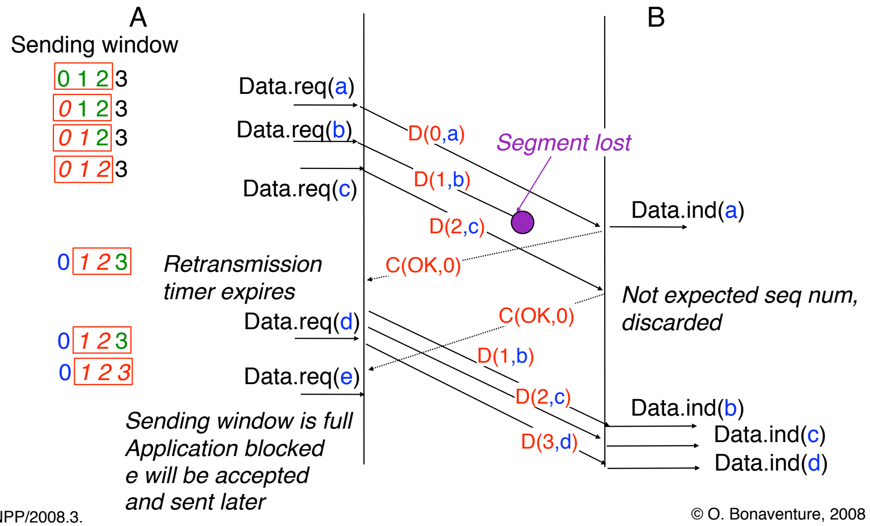


To avoid overloading the slide, we assume that sequence numbers are integers. Of course, all computations must be performed modulo $2^N$.

# Go-Back-N : Example



# Go-Back-N : Example (2)



# Selective Repeat

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

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

Acknowledged segments

Out of sequence segments

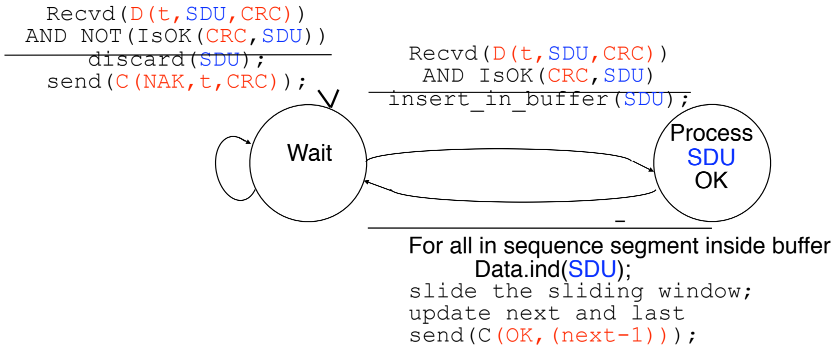
Acceptable segments

Segments outside receiving window

- 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





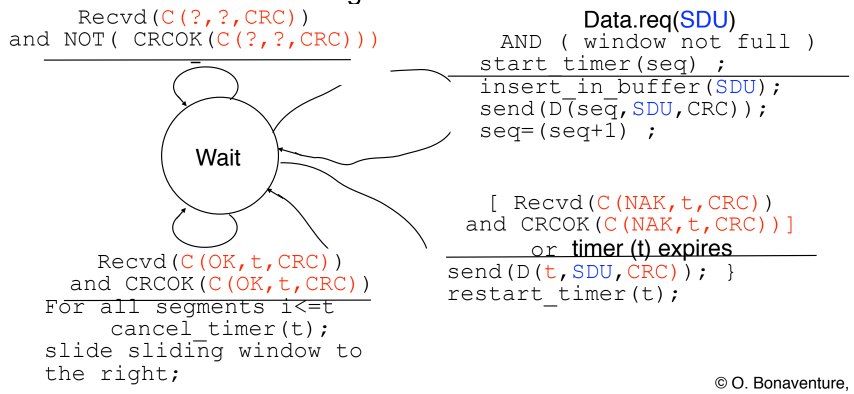




To avoid overloading the slide, we assume that sequence numbers are integers. Of course, all computations must be performed modulo $2^N$.

# Selective Repeat : Sender

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

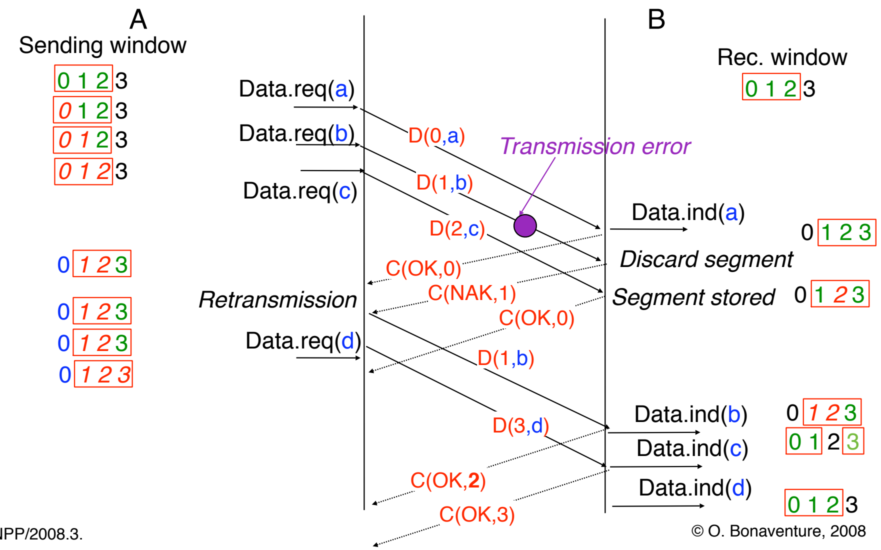




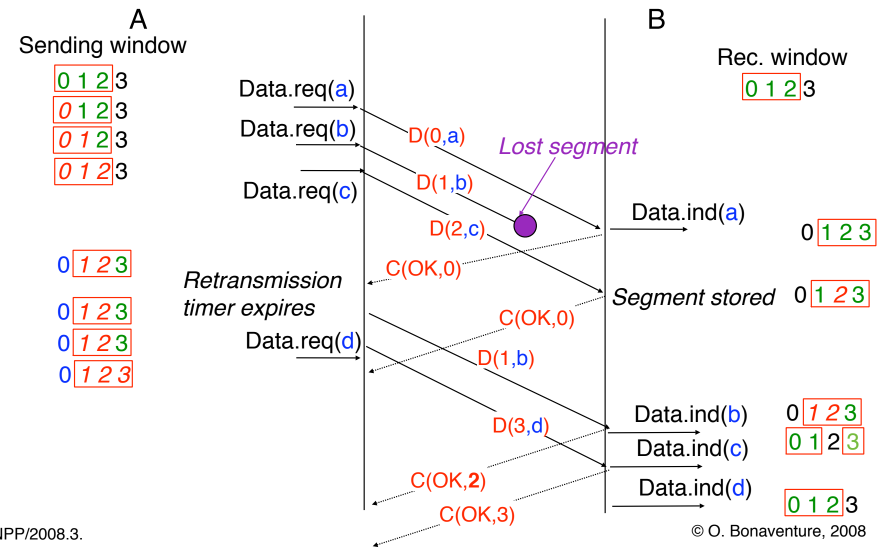



To avoid overloading the slide, we assume that sequence numbers are integers. Of course, all computations must be performed modulo $2^N$.
Other retransmission strategies are possible at the sender.

# Selective Repeat : Example



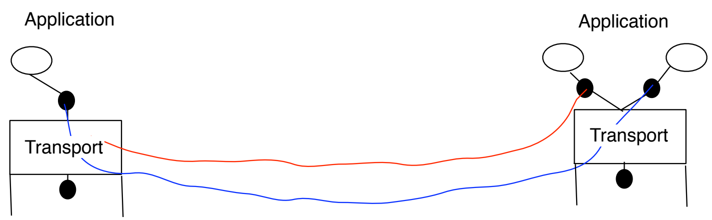
# Selective Repeat : Example (2)




CNPP/2008.3.

# 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 oones
    - `ftp` versus `telnet`

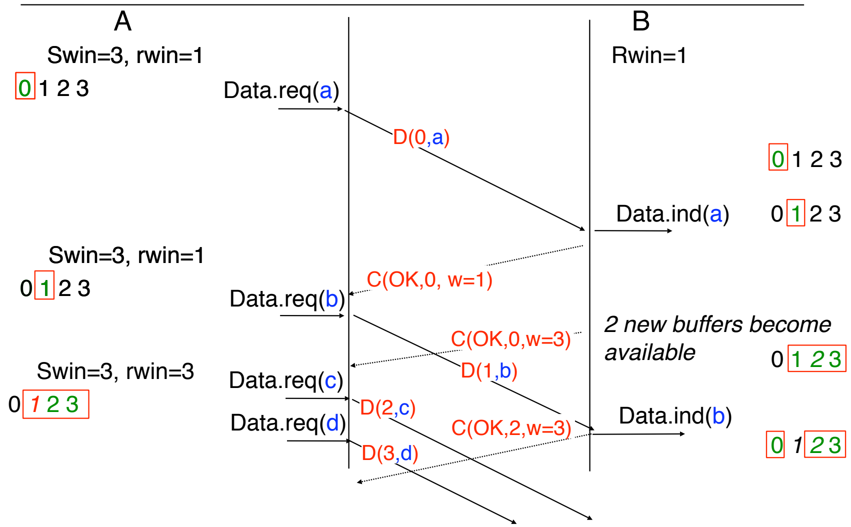
# Buffer management (2)

- 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

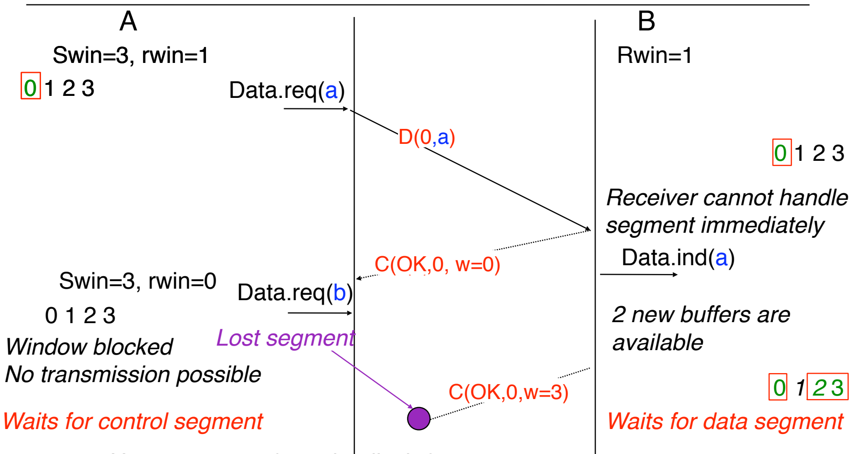
  - New information carried in control segments
    - `win` indicates the current receiving window's size

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

# Buffer management (3)



# Buffer management (4)




How to recover from deadlock ?
Persitence timer on receiver, resend control segment after timer expiration

## Duplication and reordering

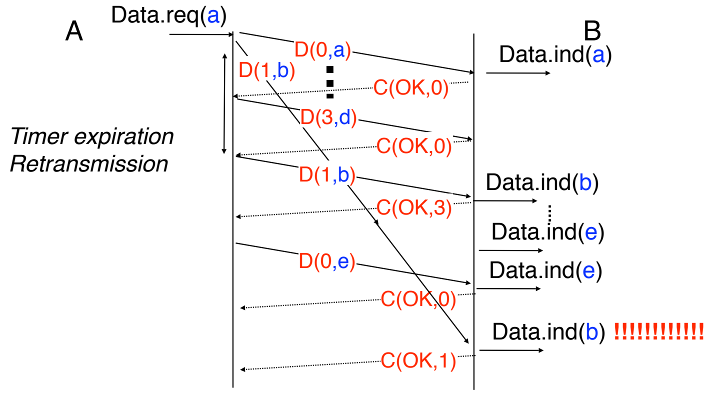
- How can we provide a reliable service in the transport layer ?

- 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 (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



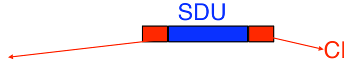



MSL means Maximum Segment Lifetime

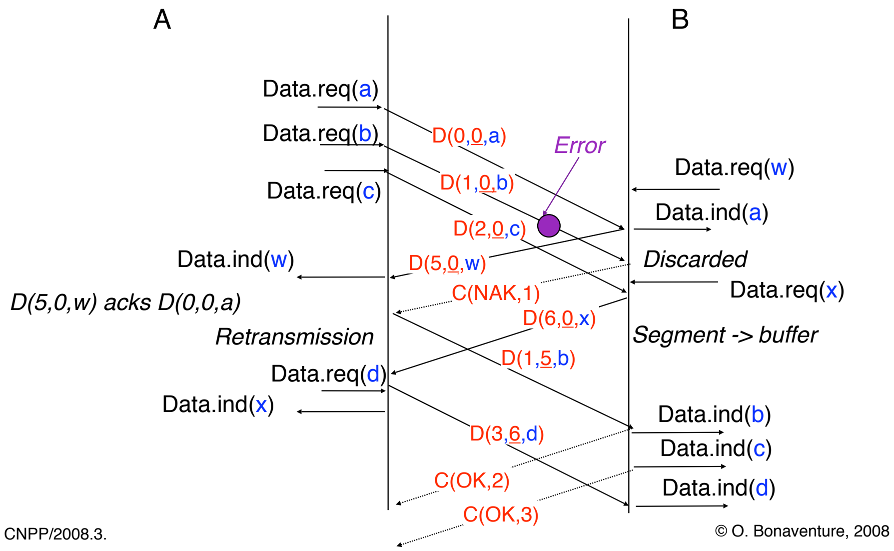
# Bidirectional flow

- 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



# Bidirectional flow
# Example

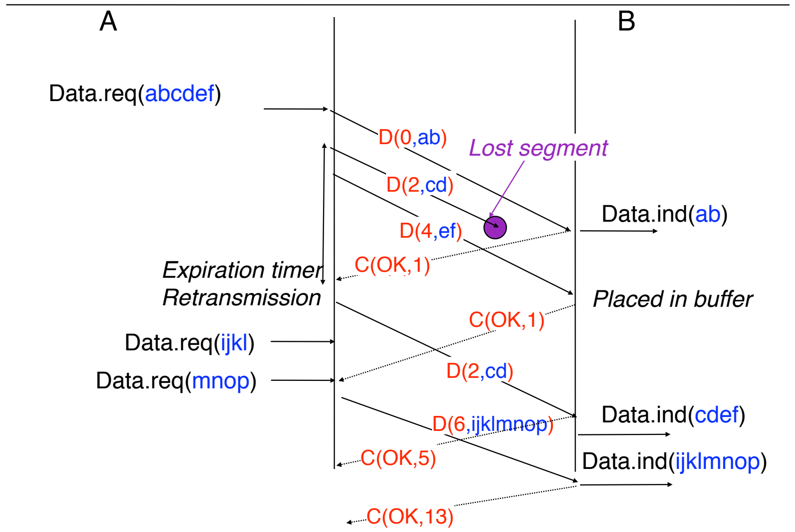



CNPP/2008.3.

# 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)



# 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

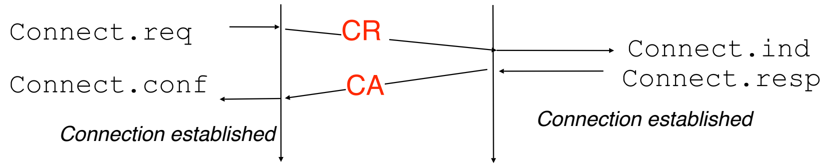
## Transport connection establishment

- 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

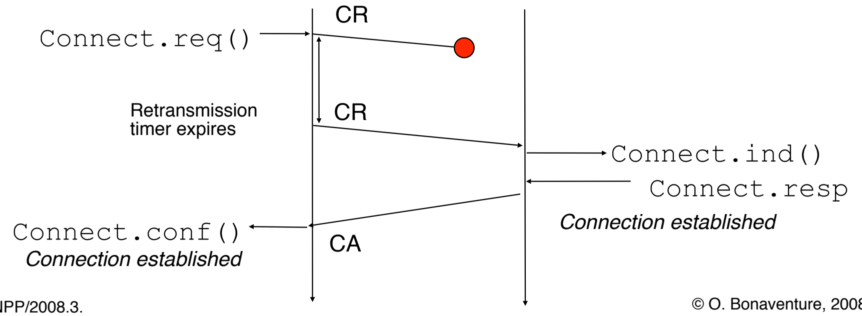
# Simple solution




- 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 ?

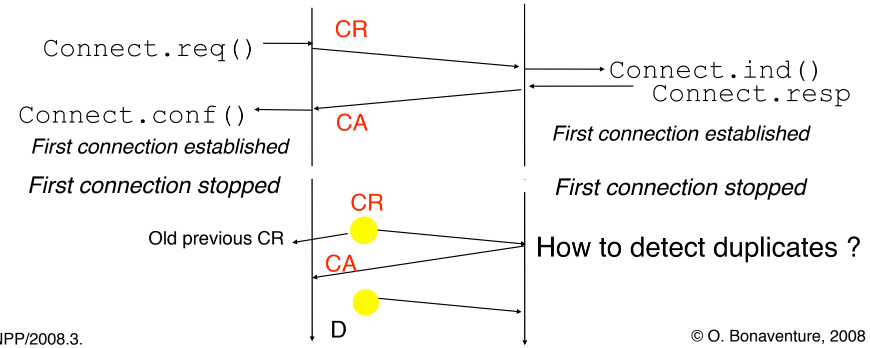
## Simple solution (2)

- 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









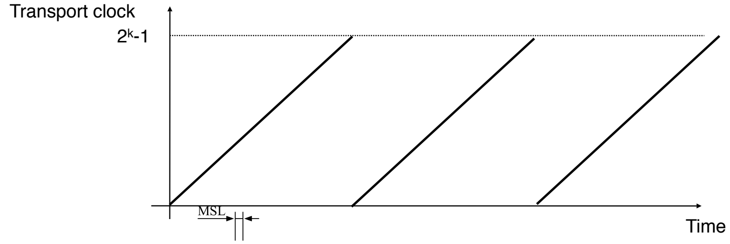

In this example, the duplicate CR is likely to be a previous retransmission of the CR that was delayed in the network.

## Connection establishment (2)

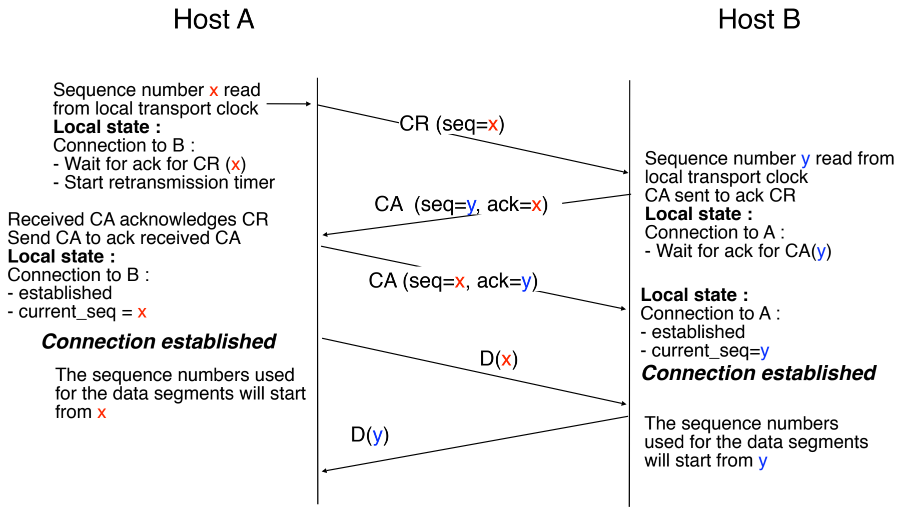
- How to detect duplicates ?
- Principles
  - The network layer guarantees by its protocols and internal organisation that a packet and its duplicates will not live forever inside the network
    - No packet will survive more than MSL seconds inside the network
  - Transport entities rely on a local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

# Connection establishment (3)

- Transport clock
  - Maintained by each transport entity
    - usually implemented as a k-bits counter
      - $2^k$ * clock cycle >> 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

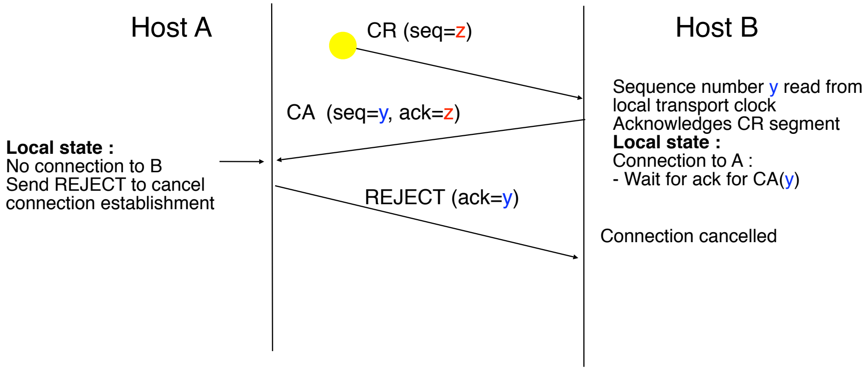


# Three way handshake



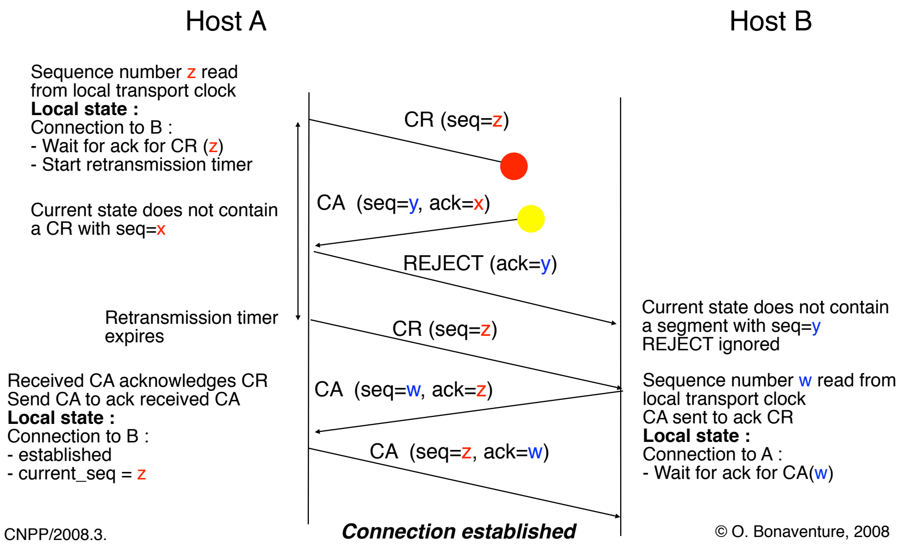
# Three way handshake (2)

- What happens with duplicates
  - Duplicated CR



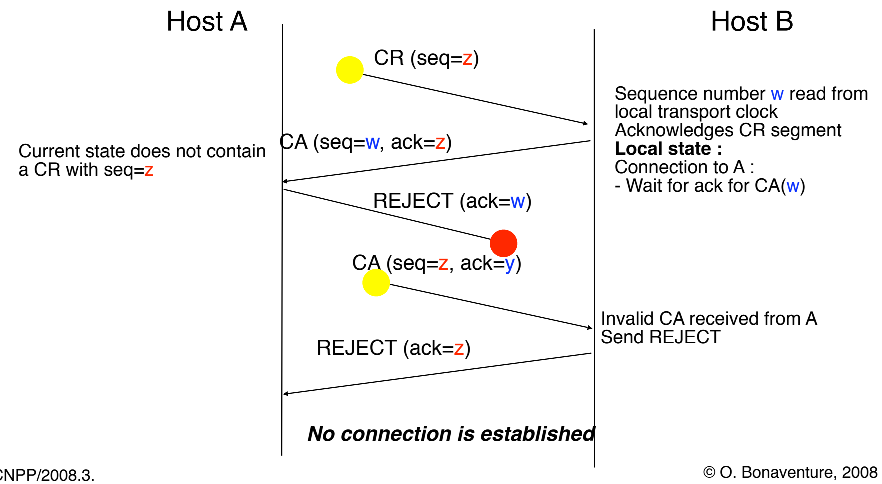

***No connection is established***

# Three way handshake (3)



# Three way handshake (4)

- Another scenario



# 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

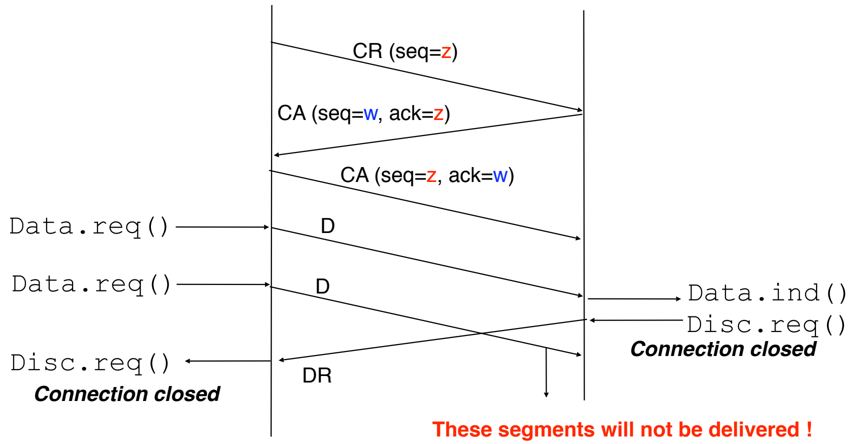
## Connection release

- 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 transfert
    - can lead to losses of data

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

# Abrupt shutdown

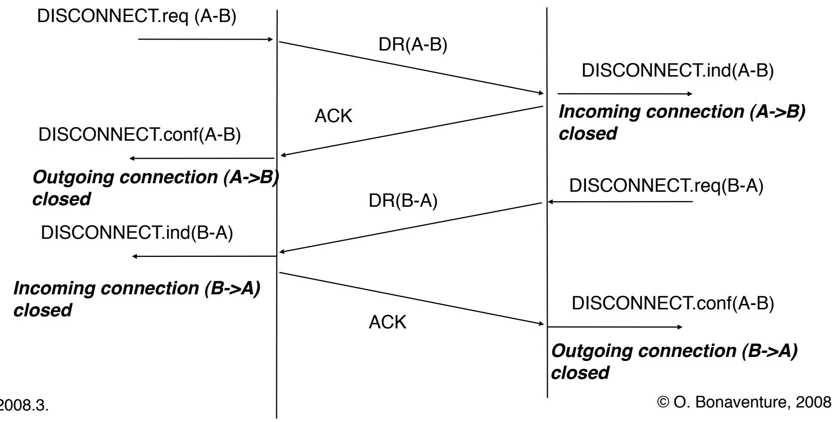


## Abrupt shutdown (2)

- 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



## 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

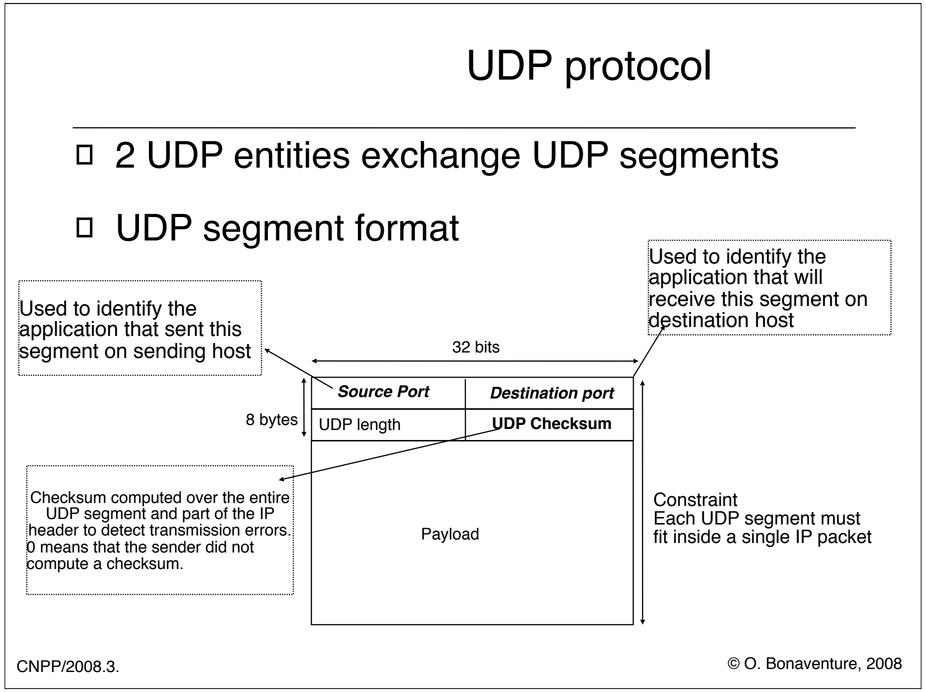
- Basics
- Building a reliable transport layer
- UDP : a simple connectionless transport protocol
- TCP : a reliable connection oriented transport protocol

# 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



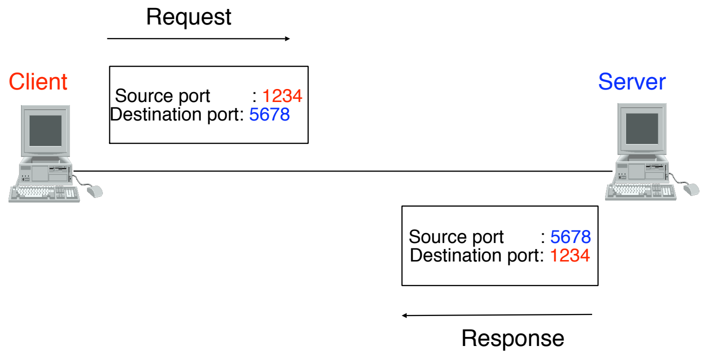



The computation of the UDP checksum is defined in :

R. Braden, D. Borman, C. Partridge, Computing the Internet Checksum, RFC1071, Septembre 1988

# UDP Protocol (2)

- Utilisation of the UDP ports



# 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

CNPP/2008.3.





UDP is mainly used for applications where either short messages are exchanged or losses or not a severe problem (either because they can be supported by the application or because they are used in LAN environment where there are almost no losses)

- Domain Name System, Network File System (NFS), Remote Procedure Call (RPC), jeux

Multimedia (conversational) applications such as VoIP or VideooverIP often use UDP. In this case, UDP is often combined with RTP

H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson.RTP: A Transport Protocol for Real-Time Applications. RFC1889, Jan 1996

## 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

CNPP/2008.3.





TCP was defined in

J. Postel. Transmission control protocol. Request for Comments 793, Internet Engineering Task Force, September 1981.

It has been heavily modified since

A more detailed description of TCP may be found in

W. Stevens. TCP/IP Illustrated, volume 1 : The protocols. Addison-Wesley, 1994.

# TCP

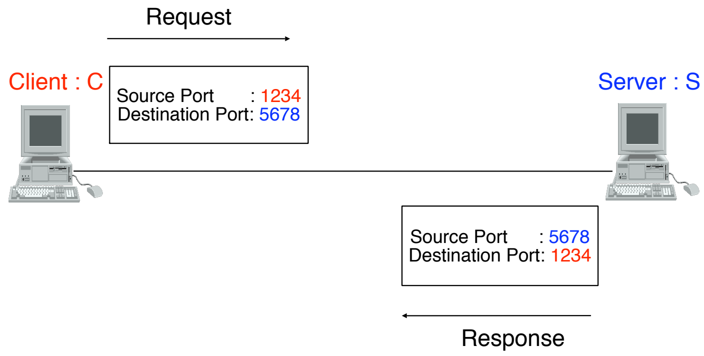
- Transmission Control Protocol
- Provides a reliable byte stream service
- Characteristics of the TCP service
  - TCP connections
  - Data transfer is reliable
    - no loss
    - no errors
    - no duplications
  - Data transfer is bidirectionnal
  - TCP relies on the IP service
  - TCP only supports 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

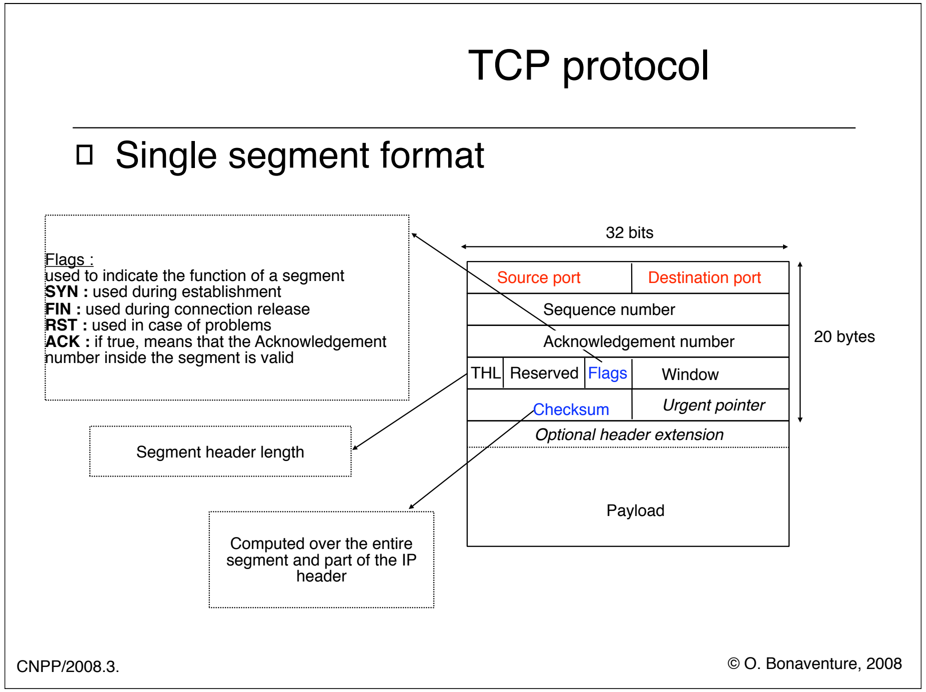



Established TCP connections on client

| 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 |



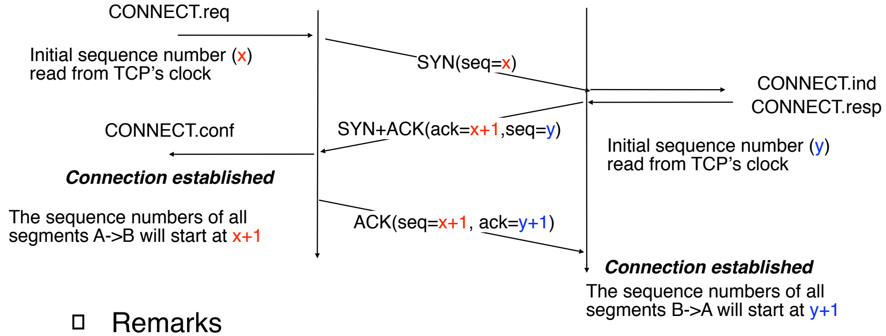



Urgent pointer is rarely used and will not be described.

The THL is indicated in blocs of 32 bits. The TCP header may contain options, these will be discussed later.

# TCP connection establishment

- Three-way handshake




- 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

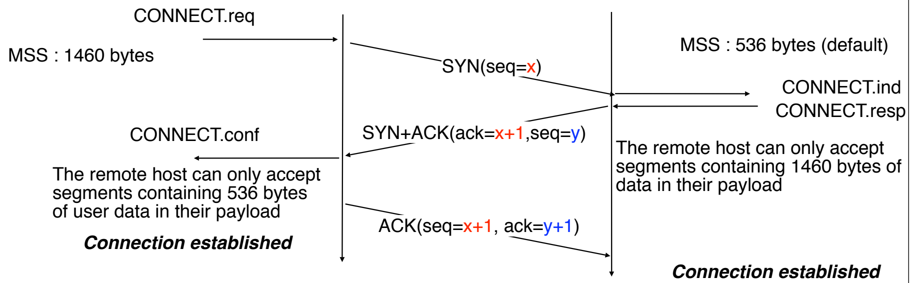





MSL in IP networks : 120 seconds

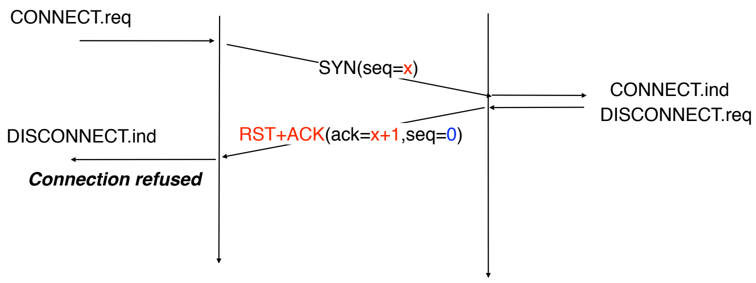
# TCP connection establishment (2)

- 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



# TCP connection establishment (3)

- Rejection of connection establishment

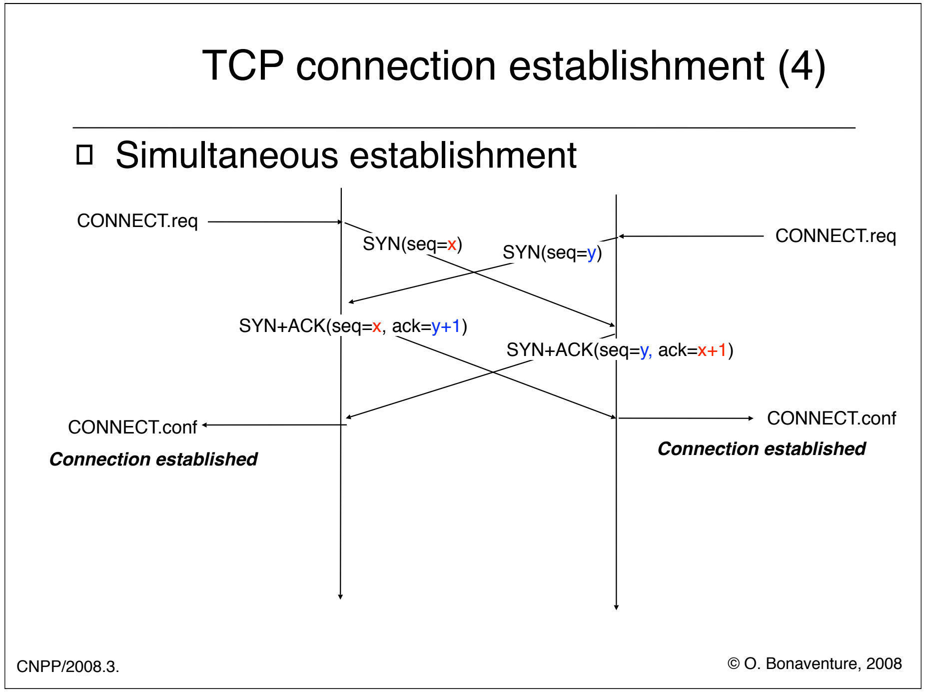



A TCP entity should never send a RST segment upon reception of another RST segment





Le segment TCP avec le flag RST est également utilisé envoyé lorsqu'une entité TCP reçoit un segment TCP n'ayant pas le bit SYN mis à vrai et qui est relatif à une connexion TCP n'existant pas à ce moment sur cette entité. Le segment RST peut également être utilisé pour informer l'autre extremité de la connexion TCP de problèmes de syntaxe dans un segment TCP reçu. Tout envoi d'un segment TCP avec le flag RST mis à vrai provoque la fin de la connexion TCP sur laquelle il est envoyé. Cependant, afin d'éviter des envois permanents de segments RST, une entité TCP n'enverra jamais de segment RST en réponse à un segment RST.



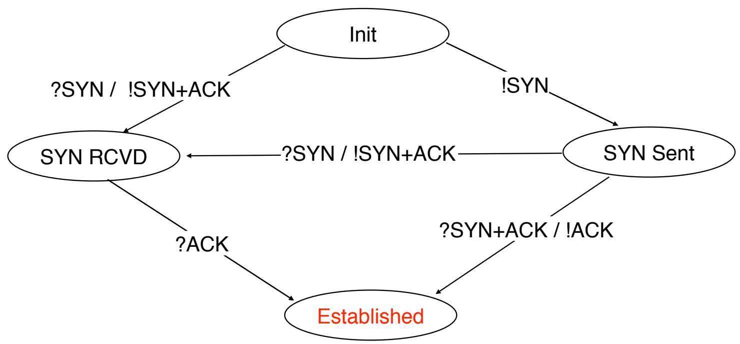

98

En pratique, cette ouverture simultannée est assez rare car souvent le client choisit un numéro de port TCP local ephémère pour contacter le serveur. Une ouverture simultanée ne se produira en pratique que si le client et le serveur utilisent chacun un numéro de port TCP « bien connu ».

Dans cette ouverture, le premier segment envoyé par chaque entité est un segment dont le bit SYN est mis à vrai. Ensuite, après avoir reçu le segment SYN, chaque entité va acquiter ce segment SYN en envoyant un segment dont le bit ACK est mis à vrai et en acquittant le numéro de séquence annoncé dans le segment SYN reçu. Cet acquittement prouve que l'entité qui l'envoie a bien reçu le segment SYN. Il faut noter que le bit SYN est mis à vrai dans le segment d'acquittement car le segment SYN n'a pas encore été acquitté. Lors de la réception de l'acquit, le three-way handshake se termine pour l'entité qui le reçoit car à ce moment son SYN a été acquitté et elle a acquité le SYN envoyé par l'autre entité.

## TCP connection establishment (5)

- Representation as a finite state machine




CNPP/2008.3.





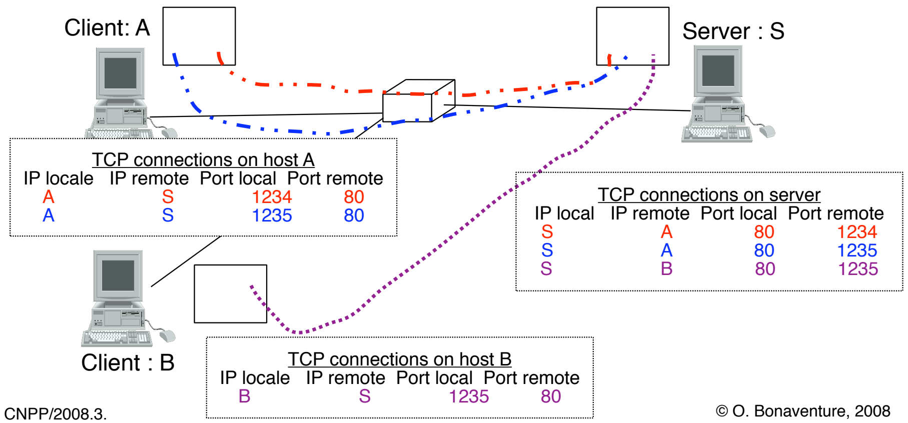
Cette machine ne prend pas en compte les retransmissions sur temporisateur.

Typiquement, un client va passer par les états Init, SynSent et Established. Un serveur passera lui par les états Init, SynRcvd et Established. Ce n'est que lors d'une ouverture simultanée que le chemin suivi est Init, SynSent, SynRcvd, Established.

Dans la transition entre l'état SYN-Sent et l'état SYN-RCV, le bit ACK est positionné pour acquitter le segment SYN reçu et le bit SYN est positionné car la première transmission n'a pas encore été acquittée. Lors d'une ouverture simultannée, la transition entre SYN-RCVD et Established sera fera lors de la réception d'un segment dont le bit ACK et le bit SYN seront positionnés.

# TCP connection establishment (6)

- How to open several TCP connections at the same time ?










Ces connexions doivent avoir des identificateurs différents
Adresse application destination

- choisie par l'application serveur, en général port fixe

Adresse application source

- en général l'application client laisse l'entité transport choisira le port source chaque fois qu'elle établi une connexion. En pratique, l'entité de transport maintiendra une liste des connexions TCP actives et lorsqu'un nouvelle connexion est demandée elle choisira le premier numéro de port TCP libre. Typiquement ces ports « éphémère » auront des valeurs entre 1024 et 5000.
- si il y a plusieurs connexions avec la même application serveur, l'entité transport du client choisira des valeurs différentes pour les ports source utilisés

Sur une machine Unix, il est possible de visualiser l'état des connexions TCP activant en utilisant la commande netstat (man netstat)

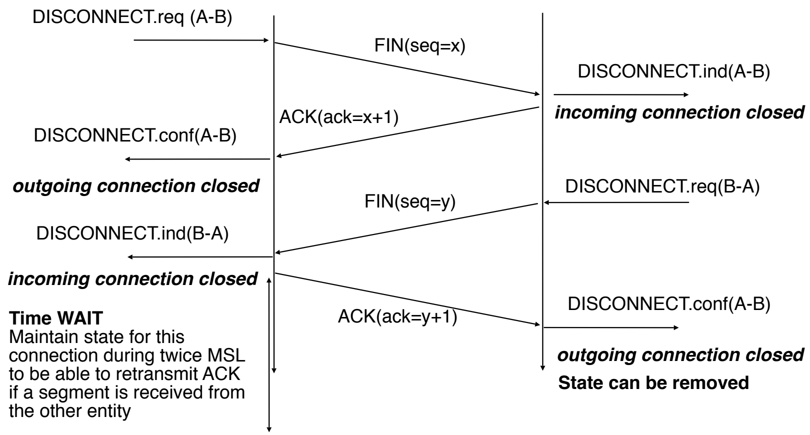
```
bismarck!obo [5] netstat -n | more

TCP
   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
130.104.229.58.1023  130.104.229.109.2049  8760      0  8760    116 ESTABLISHED
130.104.229.58.1021  130.104.229.51.2049  49640      0  8760      0 ESTABLISHED
130.104.229.58.997   130.104.229.33.2049  33176      0  8760      0 ESTABLISHED
...
```

# 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

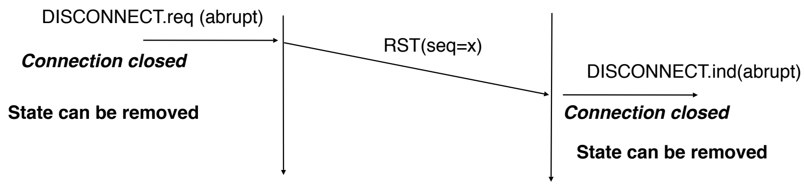
# TCP connection release

- Graceful shutdown of a TCP connection




CNPP/2008.3.

## Abrupt TCP connection released



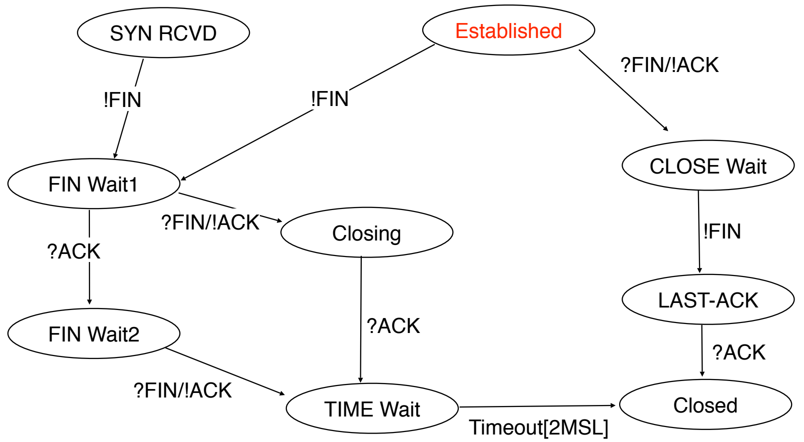

- Data segments can be lost during such an abrupt release
- No entity needs to wait in TIME_WAIT state after such a release
  - anyway, any segment received when there is no state causes the transmission of a RST segment






Some heavily loaded web servers, use abrupt release to close their connection to avoid maintaining state for 2*MSL seconds.

# TCP connection release

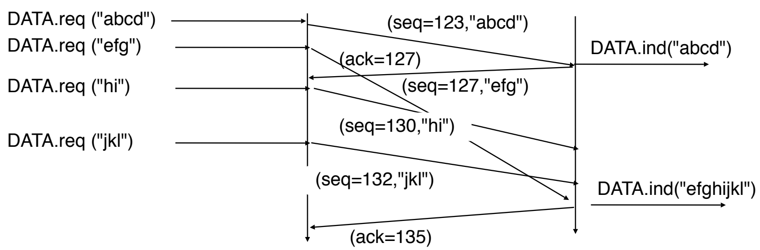


# Module 3 : Transport Layer

- 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

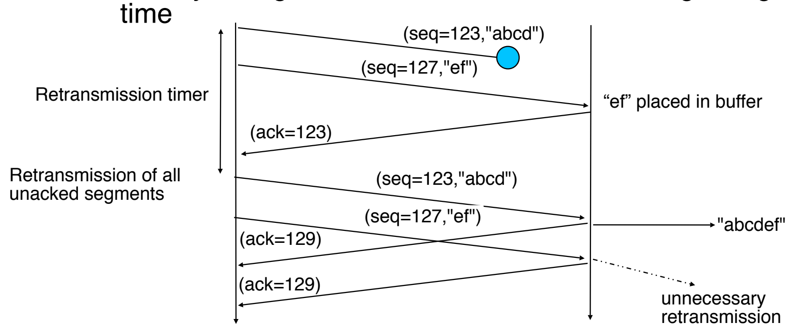
# Reliable data transfer

- Each TCP segment contains
  - 16 bits checksum
    - used to detect transmission errors affecting paylaod
  - 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

- 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




CNPP/2008.3.

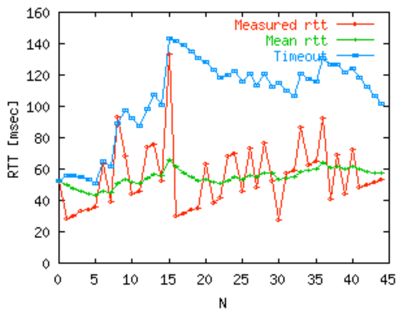




La retransmission après expiration du temporisateur est le seul mécanisme décrit dans la spécification originale de TCP. Toutes les implémentations TCP actuelles le supportent. Le bon fonctionnement du mécanisme de retransmission dépend de la bonne évaluation de la valeur à utiliser pour le temporisateur. Dans de nombreuses implémentations TCP, la valeur minimale du temporisateur est d'une centaine de millisecondes, même si le délai entre l'émetteur et le receveur n'est que d'une milliseconde.

# Retransmission timer

- TCP’s retransmission timer
  - One timer per connection
    - timer = mean(rtt) + 4*std_dev(rtt)
    - Estimation of the mean
      - est_mean(rtt) = $(1-\alpha)$*est_mean(rtt) +$\alpha$*rtt_measured
    - Estimation of the standard deviation of the rtt
      - est_std_dev=$(1-\beta)$*est_std_dev+$\beta$*|rtt_measured - est_mean(rtt)|









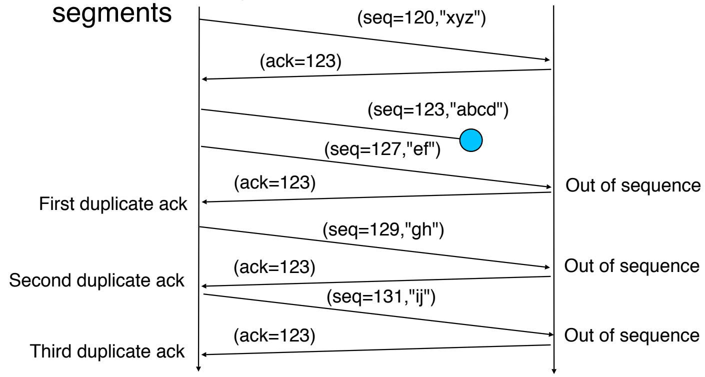

The computation of TCP’s retransmission timer is described in

RFC2988 Computing TCP's Retransmission Timer. V. Paxson, M. Allman. November 2000.

Usual values for alpha and beta are 1/8 and 1/4.

## 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





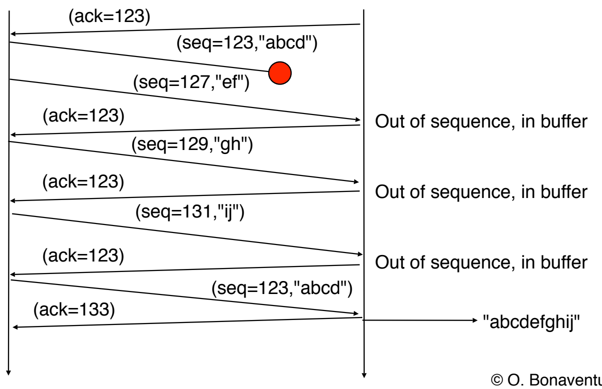




Don’t forget that TCP’s acknowledgements are cumulative.

## 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




CNPP/2008.3.



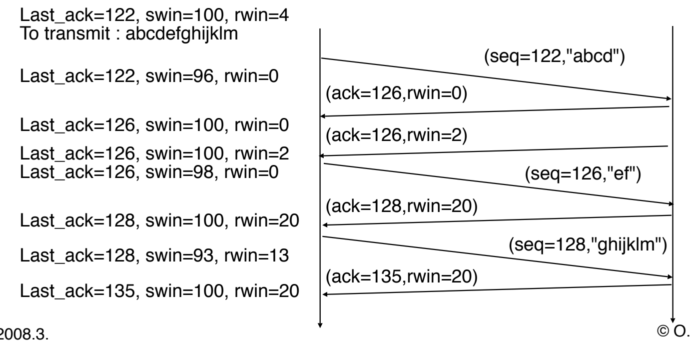


See e.g.

RFC2001 TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. W. Stevens. January 1997.

# 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



## Flow control (2)

- 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
    - ~ window / round-trip-time

| **Window** \ **rtt** | **1 msec** | **10 msec** | **100 msec** |
|---|---|---|---|
| **8 Kbytes** | 65.6 Mbps | 6.5 Mbps | 0.66 Mbps |
| **64 Kbytes** | 524.3 Mbps | 52.4 Mbps | 5.2 Mbps |

CNPP/2008.3.



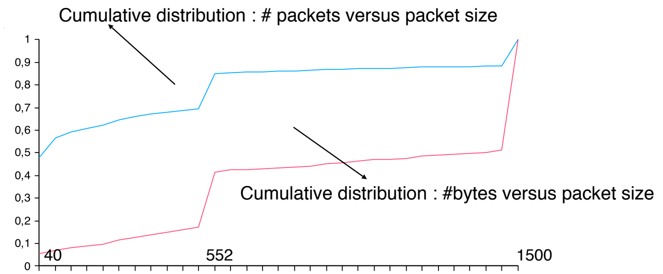


Une extension à la version initiale de TCP, décrite dans RFC1323 permet d'utiliser des fenêtres dont la taille est supérieure à 64 Kbytes. Ces extensions commencent à être supportées par les implémentations TCP courantes.

L'idée de base de cette extension est relativement simple. Tout d'abord, lors de l'établissement de la connexion TCP, on s'appuie sur le mécanisme d'extension d'entête pour transmettre de l'information complémentaire et déterminer si l'autre entité TCP de la connexion supporte les extensions RFC1323. Si oui, RFC1323, modifie un peu la sémantique du champ window contenu dans l'entête du TPDU TCP. Avant RFC1323, ce champ était codé sur 16 bits. Avec RFC1323, il est maintenu sur N bits (32>=N>=16) dans chaque entité, et le champ window du TPDU est utilisé pour transmettre les 16 bits de poids fort de cette fenêtre plutôt que l'entièreté de la fenêtre.

# 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









RFC896 Congestion control in IP/TCP internetworks. J. Nagle. Jan-06-1984.

Cet algorithme est implémenté dans toutes les implémentations de TCP ou presque, il ne nécessite que quelques lignes de code.

Pour une description détaillée d'une implémentation de TCP dans un kernel Unix, voir :

G. Wright, R. Stevens, TCP/IP Illustrated, volume 2, Addison-Wesley

Source : http://www.nlanr.net/NA/Learn/packetsizes.html

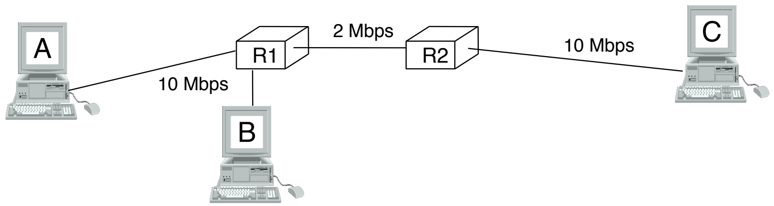
Cette information date de quelques années. Pour des statistiques plus récentes sur les tailles des paquets, voir par exemple :
http://ipmon.sprint.com

et par exemple
http://ipmon.sprint.com/packstat/packet.php?030407

# 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

# Congestion in TCP/IP networks




- 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

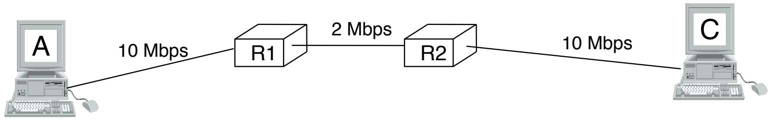
- Possible solutions
  - The network indicates explicitly the bandwidth allocated to each host
    - network sends regularly control information to hosts
    - Available Bit Rate in ATM networks
  - Endhosts measure the state of the network and adapt their bandwidth to the network state
    - Endhosts must be able to measure the amount of congestion inside the network
    - Solution used by TCP in the Internet



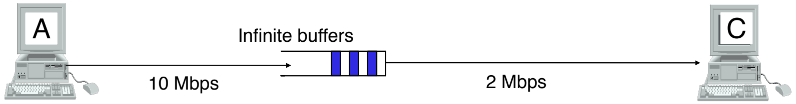



See March/April 1995 issue of IEEE Network Magazine.

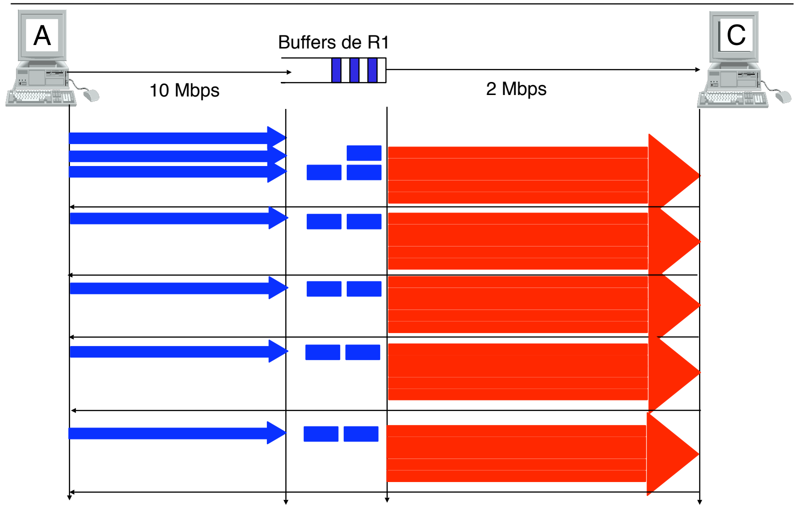
# Simple congestion




- Simplified model



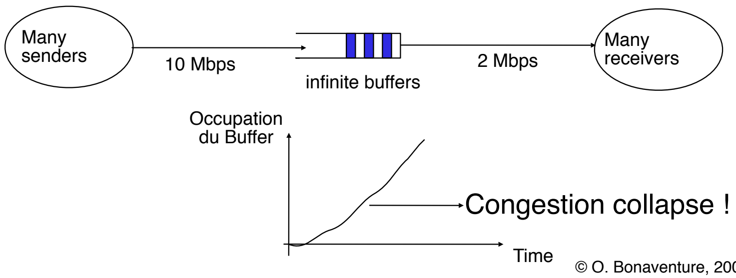
# Simple congestion




- 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









L'Internet a souffert de plusieurs épisodes d'effondrement à cause de la congestion lorsque les implémentations TCP ne supportaient que le contrôle de flux TCP de base défini dans [RFC793]. Par exemple, en 1986, le débit utilisable sur un chemin composé de trois routeurs entre l'université de Berkeley et le LBL en Californie était tombé de 32 Kbps à 40 bits par seconde suite à la congestion.

Une première solution au problème de la congestion avait été proposée par Karn et Partridge sous la forme d'un exponential backoff. L'idée de cette solution est que lorsqu'un segment TCP retransmis est perdu, c'est un signe de congestion importante et que si le même segment doit être transmis 4 fois, la congestion est plus importante que si un segment ne doit être retransmis qu'une seule fois. La solution proposée est la suivante :

- lorsqu'un segment de données qui a été retransmis précédemment doit à nouveau être retransmis suite à l'expiration du temporisateur de retransmission, la valeur de ce temporisateur est doublée.

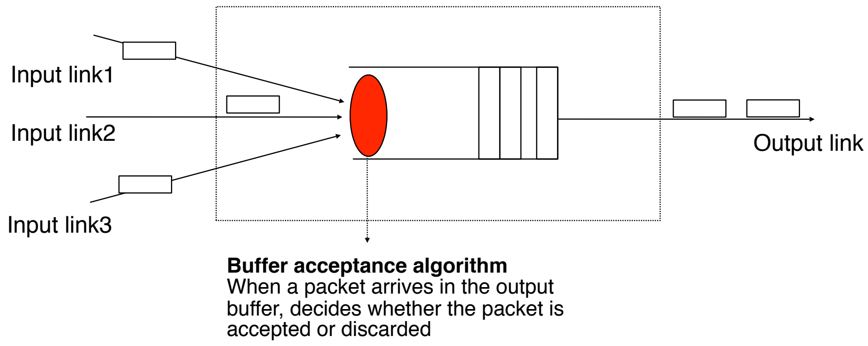
Ainsi, il le temporisateur a initialement une valeur de 1 seconde, après une première expiration pour un segment donné, il passera à 2 secondes, puis à 4, puis à 8,... En pratique, une implémentation de TCP fermera la connexion après une dizaine d'essais infructeux d'envoi du même segment TCP

# 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

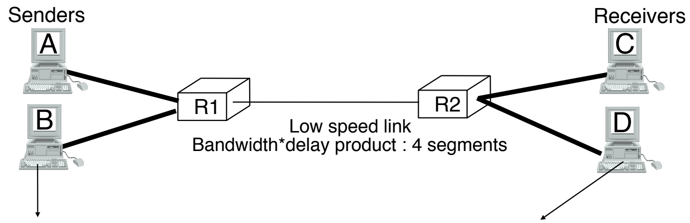
# A simple router

- Output buffer




- 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 :
[ last_ack , last_ack + min(cwnd,min(swin,rwin)) ]

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

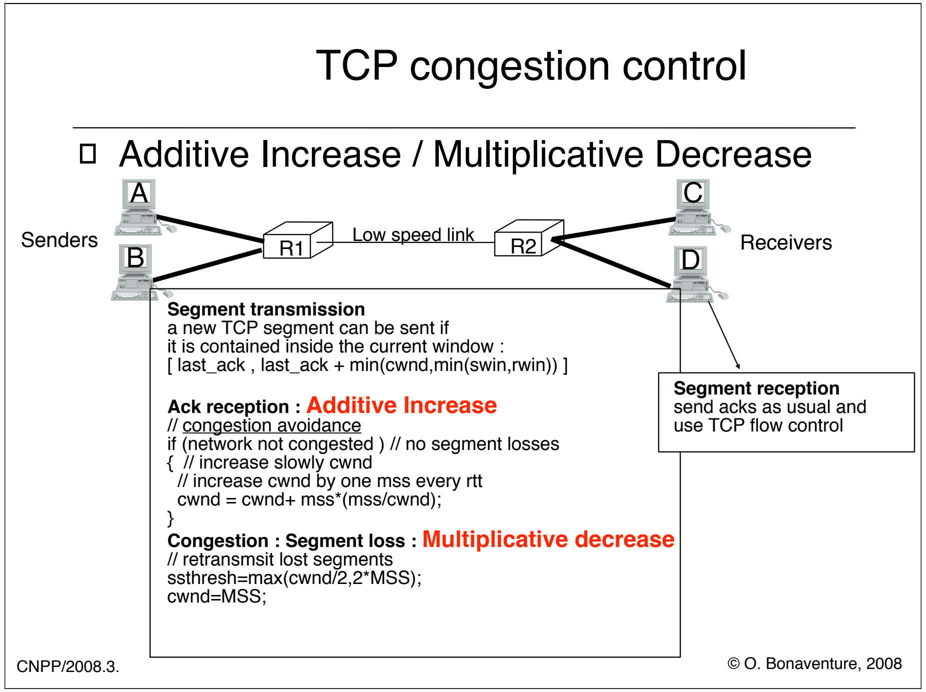
| cwnd[A] \ Cwnd[B] | 1 | 2 | 3 |
|---|---|---|---|
| 1 | underused | underused | ok, but unfair |
| 2 | underused | ok, fair | congestion |
| 3 | ok, but unfair | congestion | congestion |

- How to dynamically update cwnd ?






La fenêtre de congestion est donc une variable maintenue par l'émetteur et qui sert à limiter la quantité d'information que l'émetteur peut envoyer sans attendre d'acquits du receveur. Le receveur n'est pas au courant de la valeur actuelle de la fenêtre de congestion.






TCP congestion control was proposed in

V. Jacobson, Congestion avoidance and control, Proc. ACM SIGCOMM88, August 1988

A more detailed description may be found in :

RFC2581 TCP Congestion Control. M. Allman, V. Paxson, W. Stevens. April 1999.

# TCP congestion control

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

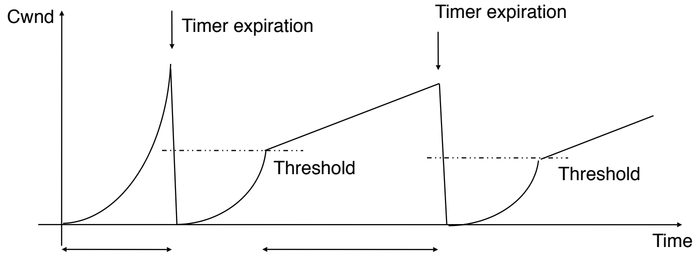
Slowstart

Congestion avoidance

```
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);
   }
}
```

# TCP congestion control

- Example




**Slow-start**
exponential increase of cwnd

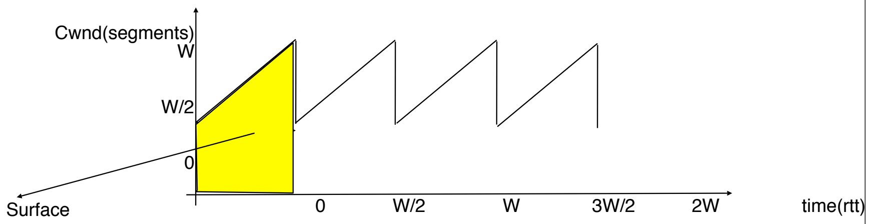
**Congestion avoidance**
linear increase of cwnd

# 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




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

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

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

Maximum throughput without losses

Throughput with losses/ congestion







More detailed models can be found in the scientific literature :

M. Mathis,J. Semke, J. Mahdavi and T. Ott, The macroscopic behaviour of the TCP congestion avoidance algorithm, ACM Computer Communication Review, 1997