Redes de Computadores

LEIC-A

3 – Transport Layer (part 1)

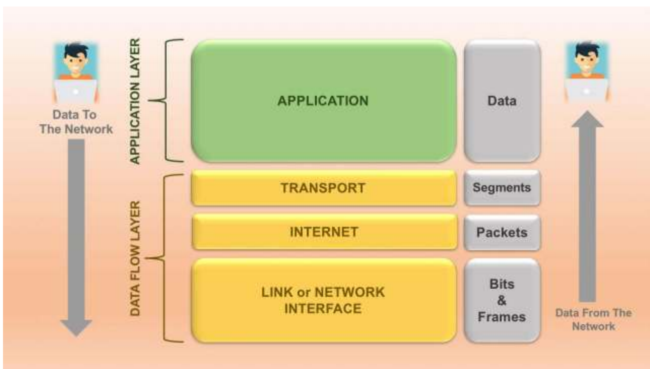
Prof. Paulo Lobato Correia

IST, DEEC – Área Científica de Telecomunicações

Modelo TCP/IP

Application
Transport
Network
Link
Physical

Five-layer Internet protocol stack

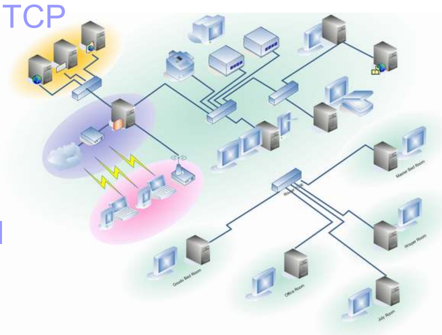


Objectives

- Understand the principles behind transport layer services:
  - Multiplexing/demultiplexing;
  - Reliable data transfer;
  - Flow control;
  - Congestion control.
- Transport layer protocols in the Internet:
  - UDP: connectionless transport;
  - TCP: connection-oriented transport;
  - TCP congestion control.

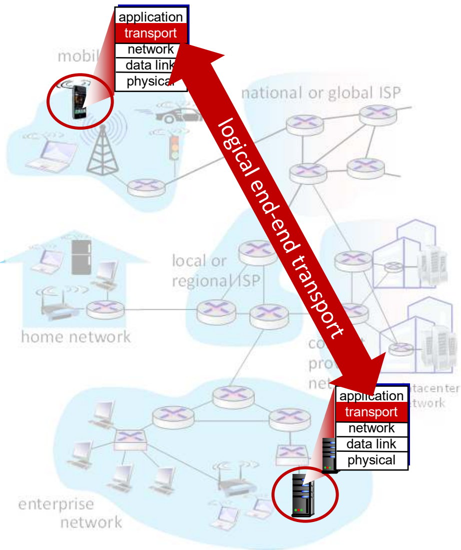
Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
  - Connection management;
- Principles of congestion control
- TCP congestion control



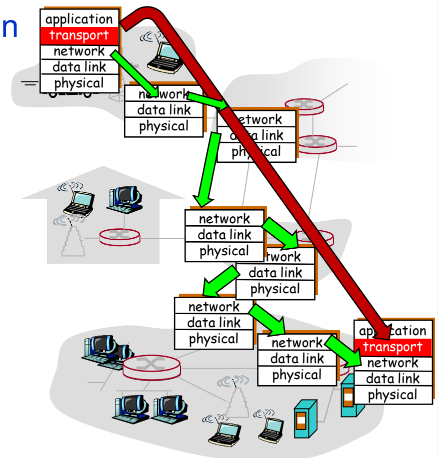
Transport Services and Protocols

- Provide *logical communication* between application processes running on different hosts;
- Transport protocols run in end systems:
  - Sender side: **breaks application messages into segments**, passes them to the network layer;
  - Receiver side: **reassembles segments into messages**, passes them to the application layer;
- Two transport protocols available to Internet applications:
  - TCP and UDP.

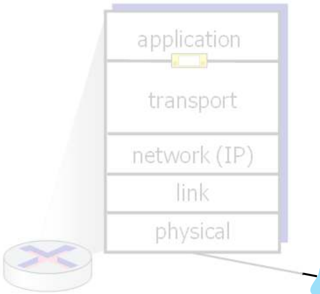


Transport vs. Network Layer

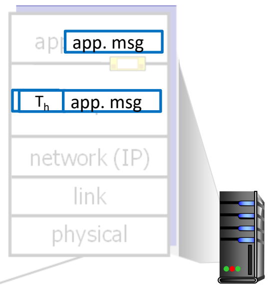
- *Transport layer:* logical communication between *processes*:
  - Relies on, and enhances, network layer services.
- *Network layer:* logical communication between *hosts*;



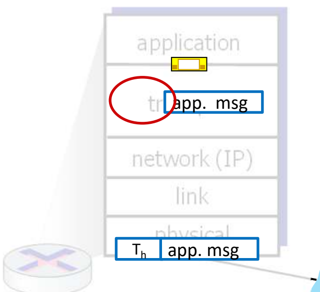
Transport Layer Actions



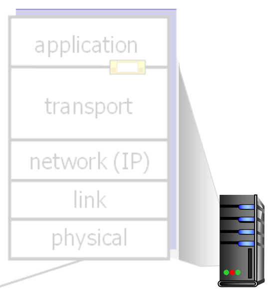
Sender:
- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP



Transport Layer Actions

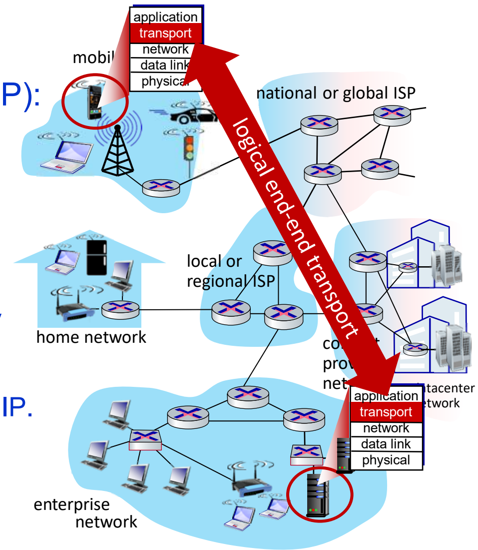


Receiver:
- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket



Internet Transport Layer Protocols

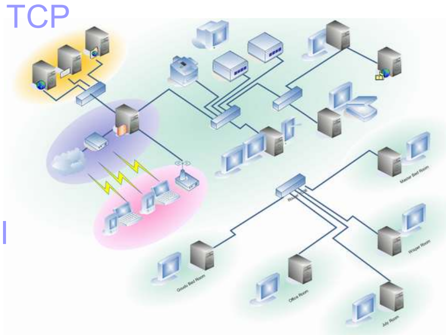
- Reliable, in-order delivery (TCP):
  - Congestion control;
  - Flow control;
  - Connection setup.
- Unreliable, unordered delivery (UDP):
  - Simple extension of "best-effort" IP.
- Services not available:
  - Delay guarantees;
  - Bandwidth guarantees.



10

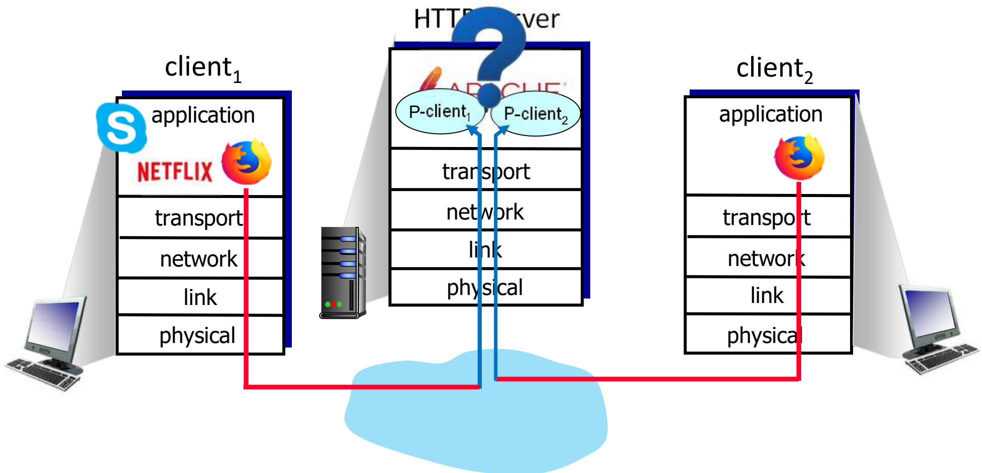
Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
  - Connection management;
- Principles of congestion control
- TCP congestion control



12

TÉCNICO LISBOA



TÉCNICO LISBOA

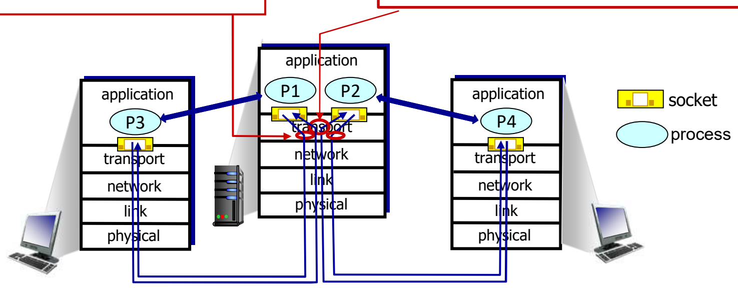
Multiplexing / Demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

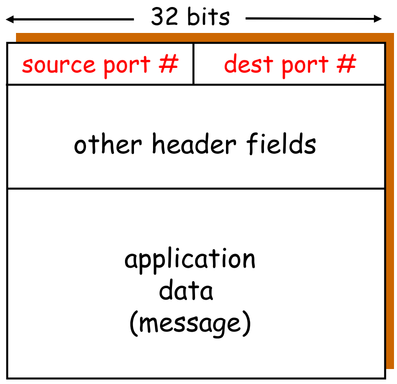
demultiplexing at receiver:

use header info to deliver received segments to correct socket



Multiplexing / Demultiplexing

- Host receives IP datagrams:
  - Each datagram has source and destination IP addresses;
  - Each datagram carries 1 transport layer segment;
  - Each segment has source and destination port numbers.
- Host uses **IP address** and **port number** to direct segment to the appropriate socket.
- Well-known ports (0-1023):
  - HTTP: 80 (TCP)
  - DNS: 53 (UDP)
  - (http://www.iana.org/assignments/port-numbers)



TCP/UDP segment format

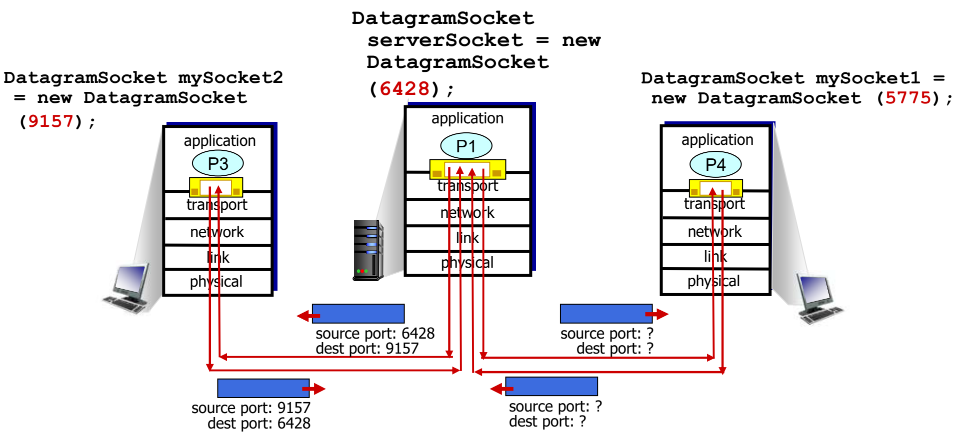
UDP: Connectionless Multiplexing

- Create socket with port number:

```
struct addrinfo hints,*res;
hints.ai_family = AF_INET;      // IPv4
hints.ai_socktype = SOCK_DGRAM; // UDP socket
hints.ai_flags = AI_PASSIVE|AI_NUMERICSERV;
getaddrinfo(NULL, PORT, &hints, &res);
MySocket = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(fd, res->ai_addr, res->ai_addrlen);
```

- **UDP sockets are identified by a 2-tuple:**
  (dest IP address, dest port number)
- When host receives UDP segment:
  - Check destination port number and direct UDP segment to corresponding socket.
- **IP datagrams with different source IP addresses and/or different source port numbers *can be directed to the same socket*.**

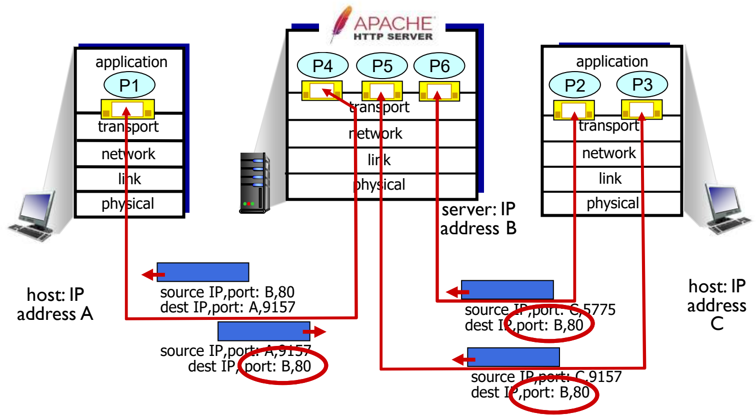
Connectionless Demultiplexing: Example



TCP: Connection-oriented Demultiplexing

- **TCP socket identified by 4-tuple:**
  - Source IP address;
  - Source port number;
  - Destination IP address;
  - Destination port number.
- Receiving host uses all four values to direct a segment to the appropriate socket;
- Server host may support many TCP sockets simultaneously:
  - Each socket identified by its own 4-tuple;
- Web servers have **different sockets for each connecting client**:
  - Non-persistent HTTP will have a different socket for each request.

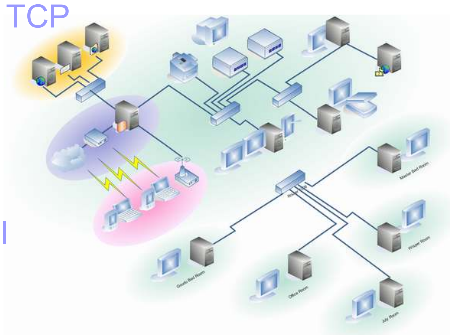
Connection-oriented Demultiplexing: Example



3 segments, all destined to IP address: B, dest port: 80 – demultiplexed to *different* sockets

Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
  - Connection management;
- Principles of congestion control
- TCP congestion control



UDP: User Datagram Protocol [RFC 768]

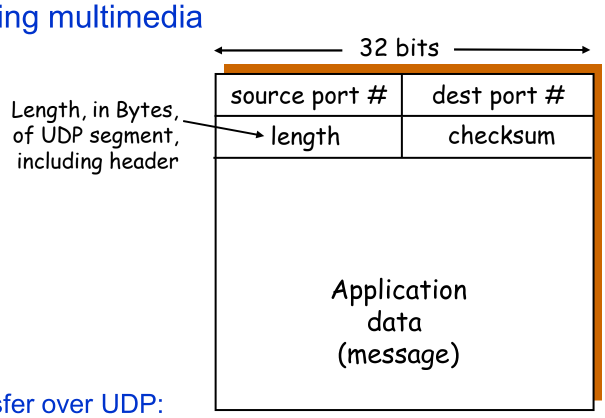
- Simple, “bare bones”, Internet transport protocol;
- **“Best effort” service** – UDP segments may be:
  - Lost;
  - Delivered out of order to application.
- *Connectionless:*
  - No handshaking between UDP sender and receiver;
  - Each UDP segment is handled independently of others.

Why is there a UDP?

- No connection establishment (which can add delay);
- Simple: no connection state at sender or receiver;
- Small segment header;
- No congestion control: UDP can “blast away” as fast as desired.

UDP: User Datagram Protocol

- Often used for streaming multimedia applications:
  - Loss tolerant;
  - Rate sensitive.
- Other UDP uses
  - DNS;
  - SNMP;
  - HTTP/3.

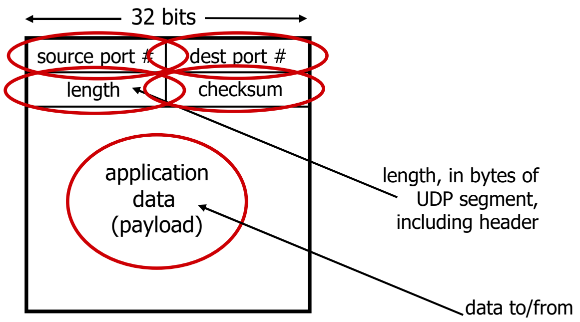


UDP segment format

To have reliable transfer over UDP:

- Need to add reliability at application layer;
- Application-specific error recovery.
- Congestion control at application layer.

UDP Segment Header



UDP segment format

UDP Checksum

Goal: detect errors (i.e., flipped bits) in transmitted segment

	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11

receiver-computed checksum ≠ sender-computed checksum (as received)

!

TÉCNICO LISBOA

UDP Checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment.

Sender:

- Treat segment contents as sequence of 16-bit integers;
- Checksum: addition (1's complement sum) of segment contents;
- Sender puts checksum value into the **UDP checksum field**.

Receiver:

- Compute checksum of received segment;
- Check if computed checksum equals checksum field value:
  - NO - error detected;
  - YES - no error detected.
    But there may be errors nonetheless?

TÉCNICO LISBOA

UDP Checksum Example

- Note:
  - When adding numbers, a carryout from the most significant bit needs to be added to the result !
- Example: add two 16-bit integers:

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum (one's complement)		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Internet Checksum: Weak Protection!

example: add two 16-bit integers

```
              1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0      0 1
              1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1      1 0

wraparound  1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

       sum    1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
  checksum    0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

Even though numbers have changed (bit flips), *no* change in checksum!

UDP Checksum

UDP checksum is computed over a pseudo header including:

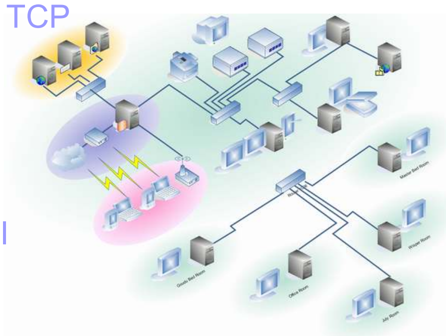
- the entire payload;
- the other fields in the header;
- some fields from the IP header.

IPv4 Pseudo Header Format

Offsets / Octet	Octet / Bit	0–7	8–15	16–23	24–31
0	0	Source IPv4 Address			
4	32	Destination IPv4 Address			
8	64	Zeroes	Protocol	UDP Length	
12	96	Source Port		Destination Port	
16	128	Length		Checksum	
20	160+	Data			

Outline

- Transport-layer services;
- Multiplexing and demultiplexing;
- Connectionless transport: UDP;
- Principles of reliable data transfer;
- Connection-oriented transport: TCP
  - Segment structure;
  - Reliable data transfer;
  - Flow control;
  - Connection management;
- Principles of congestion control
- TCP congestion control

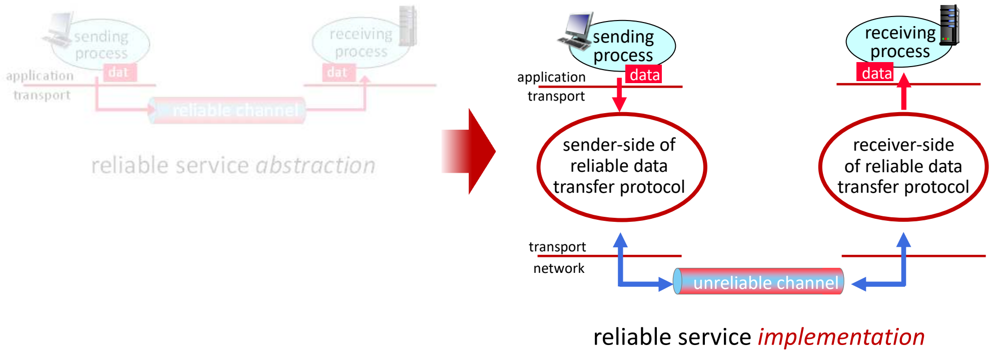


Principles of Reliable Data Transfer

reliable service *abstraction*

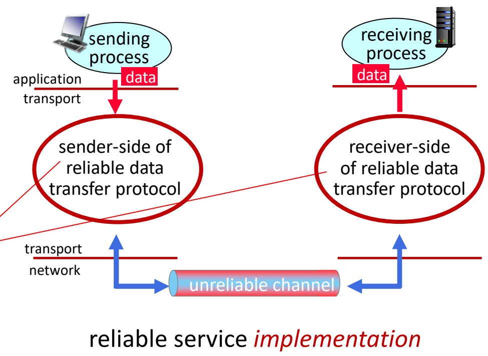
- Reliable transfer of information is important for many applications;
- Among the top-10 list of important networking topics!

Principles of Reliable Data Transfer



Principles of Reliable Data Transfer

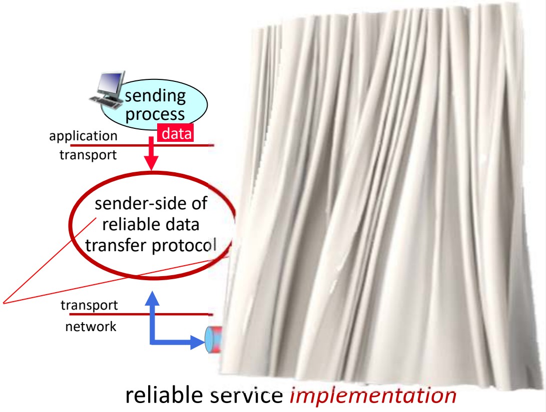
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



Principles of Reliable Data Transfer

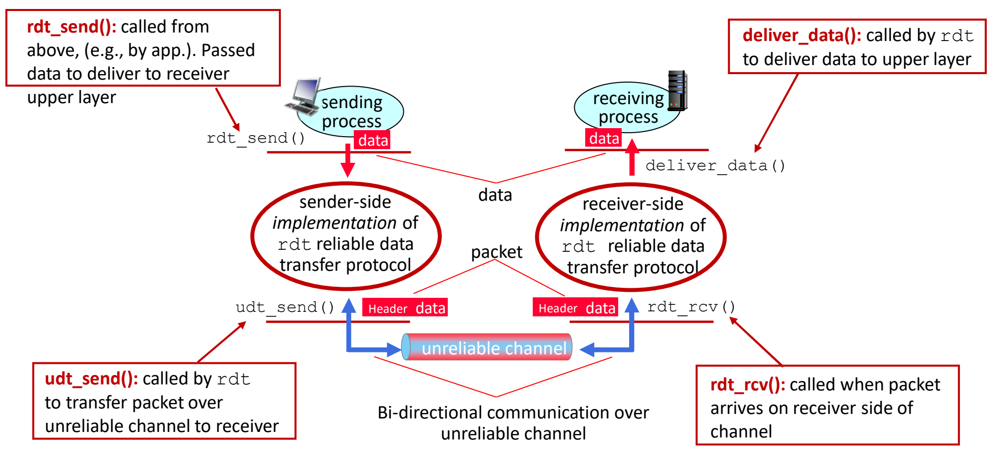
Sender, receiver do *not* know the “state” of each other, e.g., was a message received?

- unless communicated via a message



reliable service *implementation*

Reliable Data Transfer Protocol (rdt): Interfaces

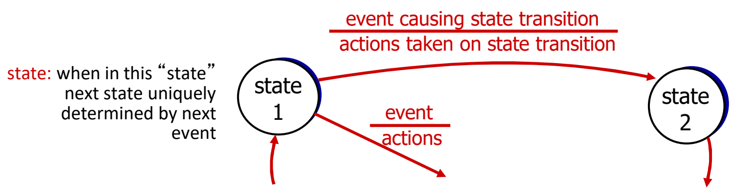


TÉCNICO LISBOA

Reliable Data Transfer: Getting Started

We will:

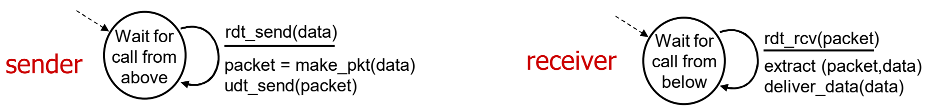
- incrementally develop sender, receiver sides of <u>r</u>eliable <u>d</u>ata <u>t</u>ransfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



TÉCNICO LISBOA

rdt1.0: Reliable Transfer over a Reliable Channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- *separate* FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel



rdt2.0: Channel with Bit Errors

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

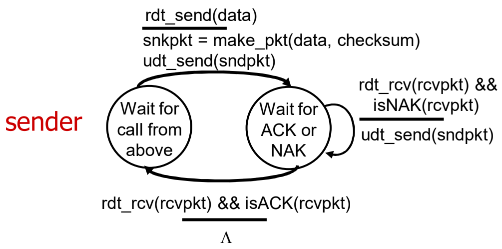
How do humans recover from "errors" during conversation?

rdt2.0: Channel with Bit Errors

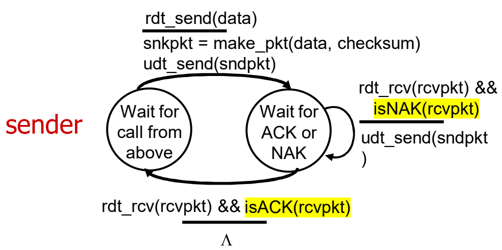
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender *retransmits* pkt on receipt of NAK

stop and wait
sender sends one packet, then waits for receiver response

rdt2.0: FSM Specification



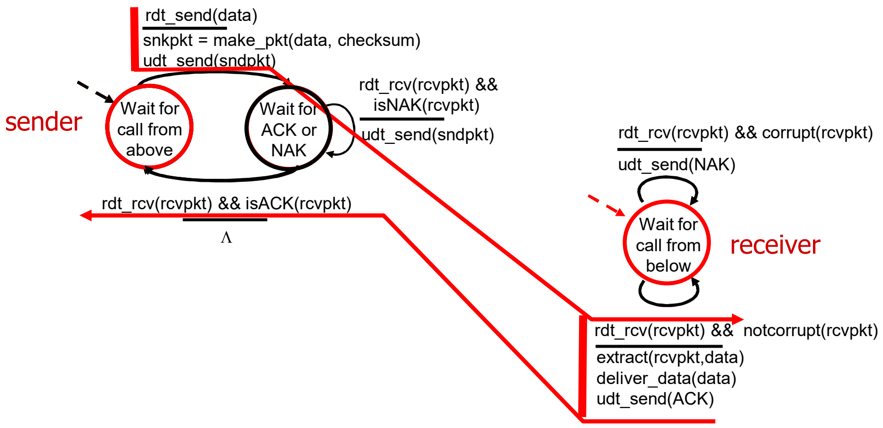
rdt2.0: FSM Specification



Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

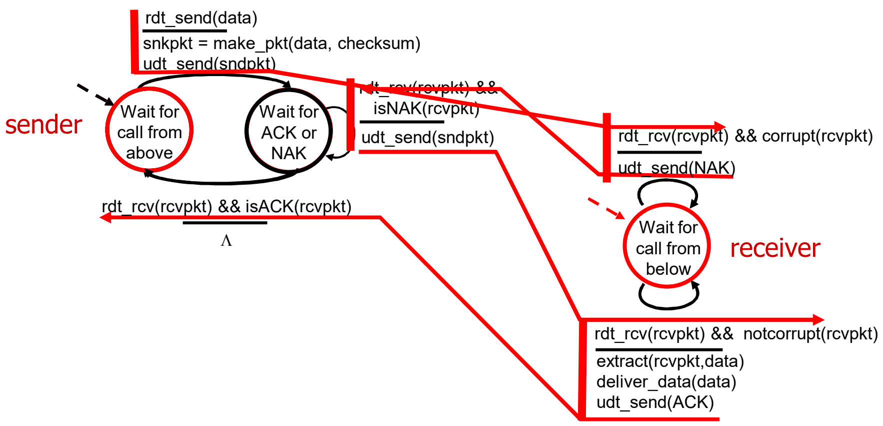
- that's why we need a protocol!

rdt2.0: Operation with No Errors



44

rdt2.0: Corrupted Packet Scenario



45

TÉCNICO LISBOA

rdt2.0 Has a Fatal Flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

handling duplicates:

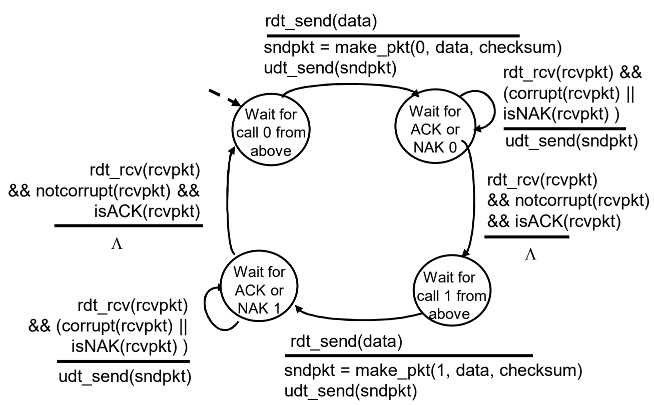
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds ***sequence number*** to each pkt
- receiver discards duplicate pkt

stop and wait

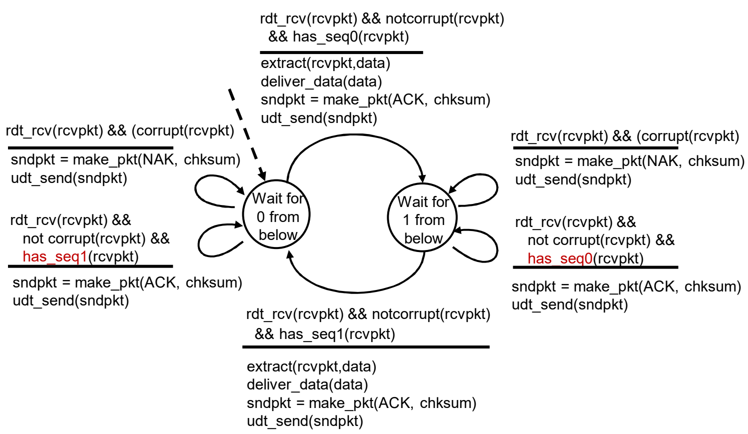
sender sends one packet, then waits for receiver response

TÉCNICO LISBOA

rdt2.1: Sender, Handling Garbled ACK/NAKs



rdt2.1: Receiver, Handling Garbled ACK/NAKs



rdt2.1: Discussion

Sender:

- Sequence number (seq #) added to packet;
- **Two seq #'s (0 and 1) are sufficient.** *Why?*
- Must check if received ACK/NAK is corrupted;
- Twice as many states:
  - State must "remember" whether "current" packet has 0 or 1 seq #.

Receiver:

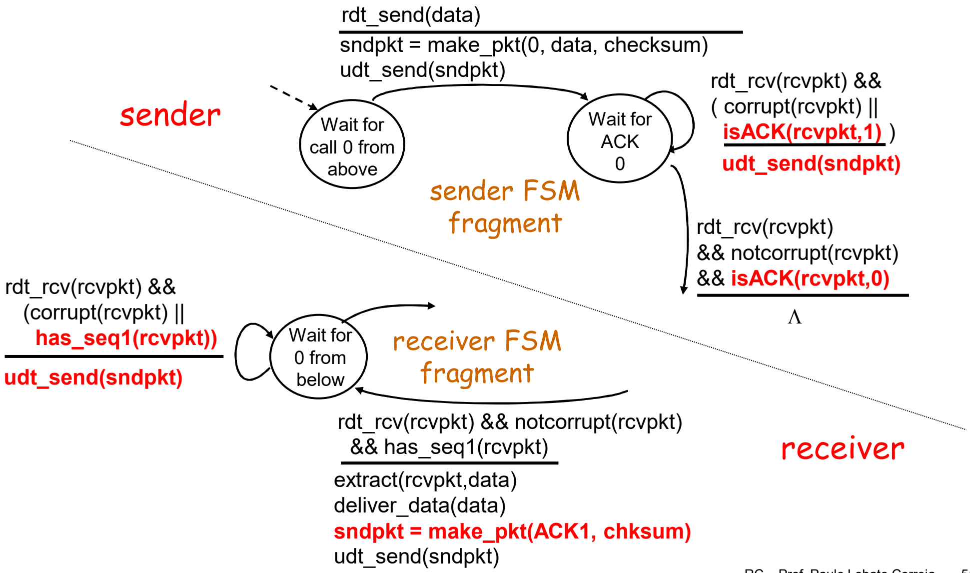
- Must check if received packet is duplicate:
  - State indicates whether 0 or 1 is expected packet seq #;

Note: receiver can *not* know if its last ACK/NAK was received OK at the sender.

rdt2.2: a NAK-Free Protocol

- Same functionality as rdt2.1, but **using ACKs only**;
- **Instead of NAK, receiver sends ACK for last packet received OK**:
  - Receiver must *explicitly* include seq # of packet being ACKed.
- Duplicate ACK at sender results in same action as a NAK would have: *retransmit the current packet.*

rdt2.2: Sender, Receiver (Fragments)



rdt3.0: Channels with Errors and Loss

New assumption:

The underlying **channel can also lose packets** (data or ACKs):

- Checksum, seq #, ACKs, retransmissions - help but are **not enough!**

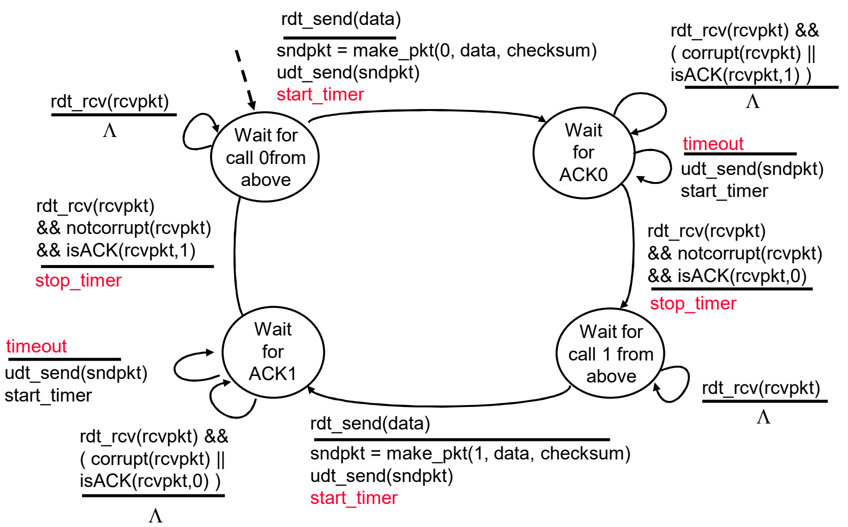
Approach:

Sender waits “reasonable” amount of time for ACK.

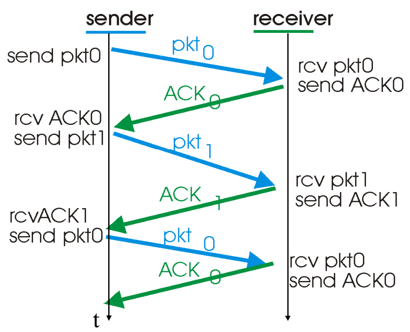
- Retransmits if no ACK received in this time;
- If packet (or ACK) was just delayed (not lost):
  - Retransmission will be duplicate, but use of seq. #’s already handles this;
  - Receiver must specify seq # of packet being ACKed.
- Requires countdown **timer**.

timeout

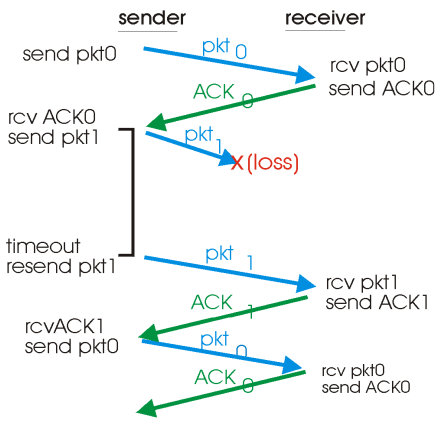
rdt3.0 Sender



rdt3.0 in Action – Stop and Wait

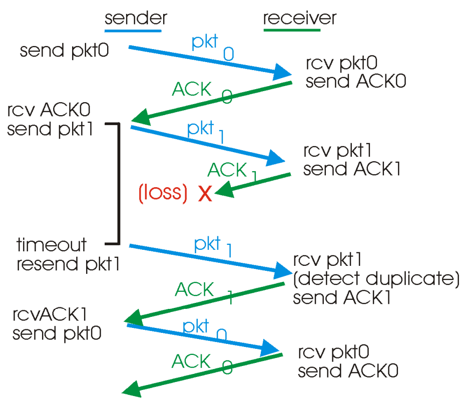


(a) operation with no loss

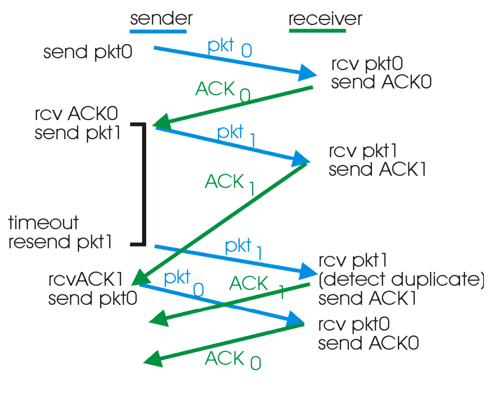


(b) lost packet

rdt3.0 in Action – Stop and Wait

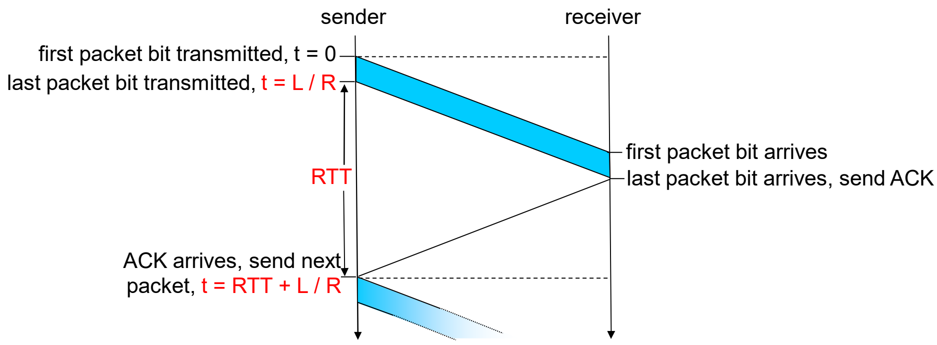


(c) lost ACK



(d) premature timeout

rdt3.0: Stop-and-Wait Operation



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Performance of rdt3.0 Stop and Wait

- rdt3.0 works, but performance is quite poor…
- Ex: 1 Gbit/s link, 15 ms prop. delay, 8000 bit packet:

$$t_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bit/s}} = 8\,\mu\text{s}$$

  - U_{sender}: utilization – fraction of time that sender is busy sending:

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

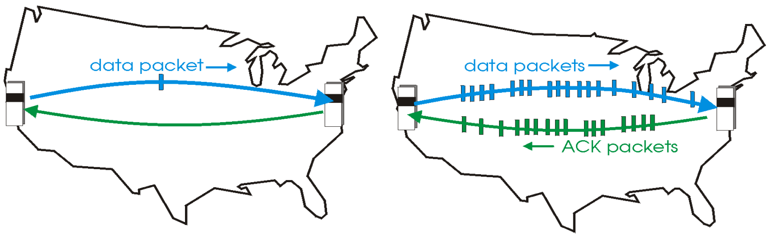
  - 1 kB packet every 30 ms -> 33kB/s (267 kbit/s) throughput over 1 Gbit/s link;
  - The protocol limits usage of physical resources!

Sliding Window Protocols (or Pipelined Protocols)

Sliding Window / Pipelining

Sender allows multiple “in-flight”, yet-to-be-acknowledged, packets:

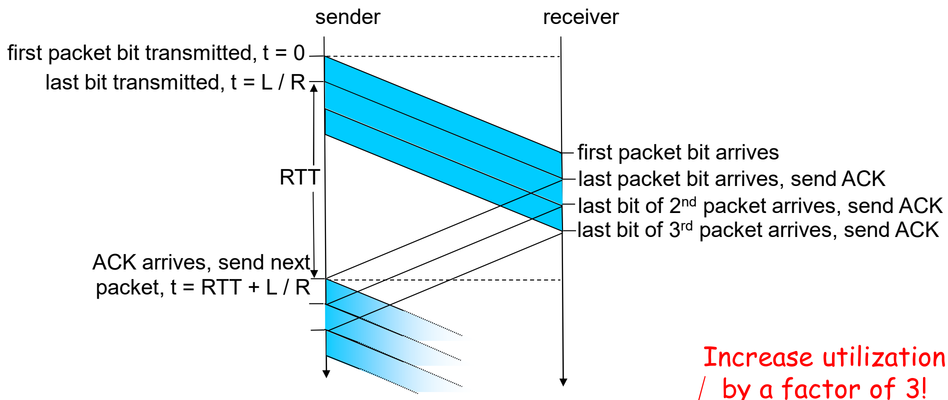
- Range of sequence numbers must be increased;
- Buffering at sender and/or receiver needed.



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of sliding window protocols: ***Go-Back-N***, ***Selective Repeat***.

Sliding Window Protocols: Increased Utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Sliding Window Protocols

Go-back-N

- Sender can have up to N unacked packets in pipeline;
- Receiver sends cumulative ACKs:
  - Doesn't ACK packet if there is a gap.
- Sender has timer for oldest unacked packet:
  - If timer expires, retransmit all unacked packets.

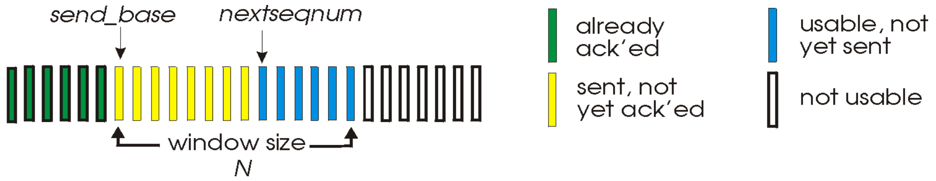
Selective Repeat

- Sender can have up to N unacked packets in pipeline;
- Receiver ACKs individual packets;
- Sender maintains timer for each unacked packet:
  - When timer expires, retransmit only the unack packet.

Go-Back-N

Sender:

- k-bit sequence number (0 - 2^{k-1}) in packet header;
- "Window" of up to N, consecutive unacked packets allowed:



- ACK(n): ACKs all packets up to, including seq. # n – "**cumulative ACK**":
  - May receive duplicate ACKs.
- Timer for each in-flight packet;
- *timeout(n):* retransmit packet n and all higher seq. # packets in window.

Test applet at:
https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html

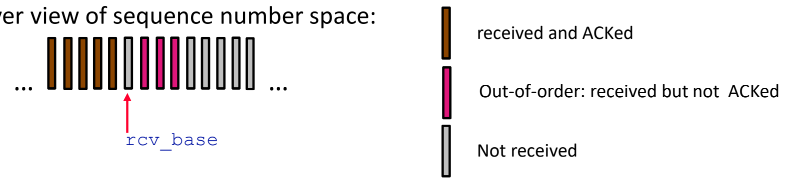
Go-Back-N: Receiver

Always send ACK for correctly received packet with highest *in-order* seq #:

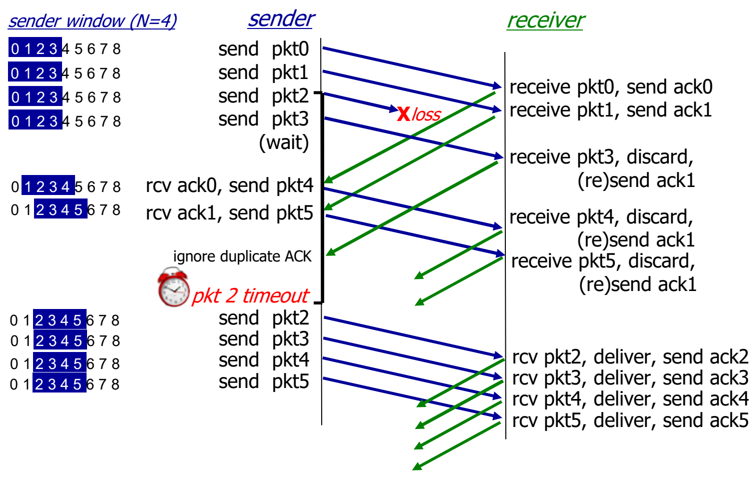
- May generate duplicate ACKs;
- Need only remember `rcv_base`.

- Out-of-order packet:
  - Discard (don't buffer) -> no receiver buffering! (implementation decision)
  - Re-ack packet with highest in-order seq #.

Receiver view of sequence number space:



Go-Back-N in Action



TÉCNICO LISBOA

Go-Back-N

Example:

- Sender window size N_w:

  $N_w = 6$

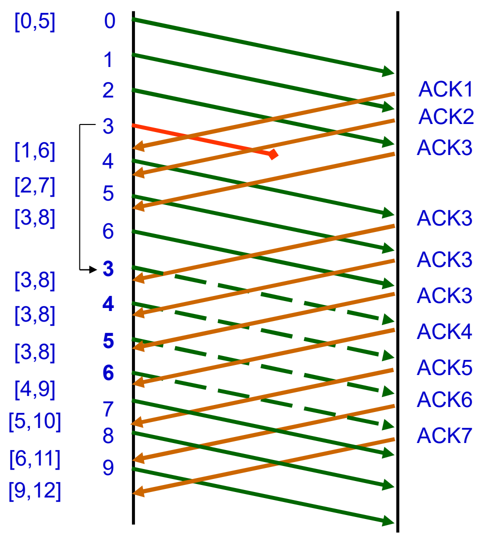
- Assume channel keeps the order of sent frames;
- Frames are numbered modulo N:

  $N = N_w + 1$

- Flow control can be done by controlling the cadence of ACKs;
- Max sender window size:

  $N_w \leq N - 1$, with $N = 2^k$

k – number of bits used for frame numbering

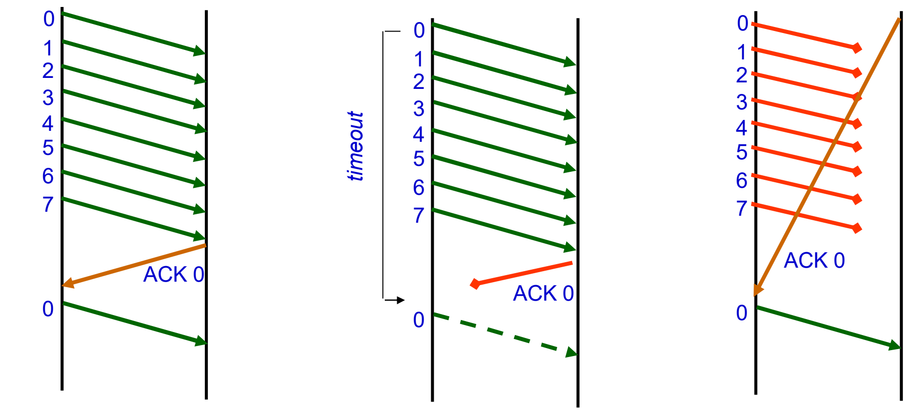


TÉCNICO LISBOA

Go-Back-N: Max Sender Window Size

Max sender window size: $N_w \leq N - 1$, with $N = 2^k$

Example: N=8 identifiers available (k = 3), sender window size: $N_w = 8$

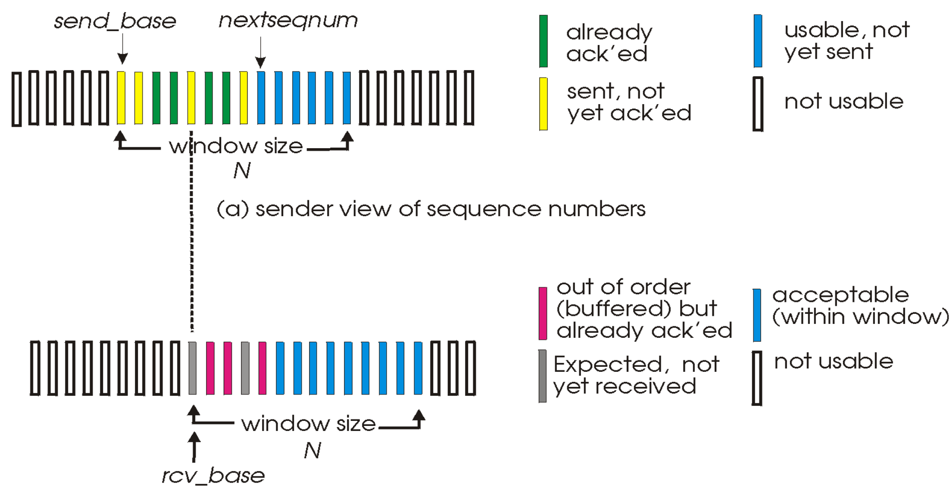


Normal operation (no errors) *8 frames duplicated* *8 frames lost*

Selective Repeat

- Receiver *individually* acknowledges all correctly received packets:
  - Buffer packets, as needed, for eventual in-order delivery to upper layer;
- Sender only resends packets for which ACK was not received:
  - Sender timer for each unACKed packet;
- Sender window:
  - N consecutive seq #'s;
  - Again limits seq #s of sent, unACKed packets.

Selective Repeat: Sender, Receiver Windows



Test applet at: (b) receiver view of sequence numbers

https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html

Selective Repeat: Sender and Receiver

Sender

data from above:
- if next available seq # in window, send packet

timeout(*n*):
- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N]:
- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

Receiver

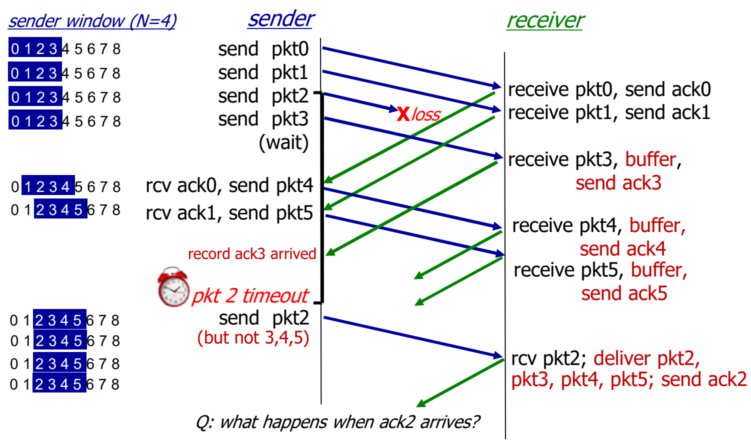
packet *n* in [rcvbase, rcvbase+N-1]
- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet *n* in [rcvbase-N,rcvbase-1]
- ACK(*n*)

otherwise:
- ignore

71

Selective Repeat in action

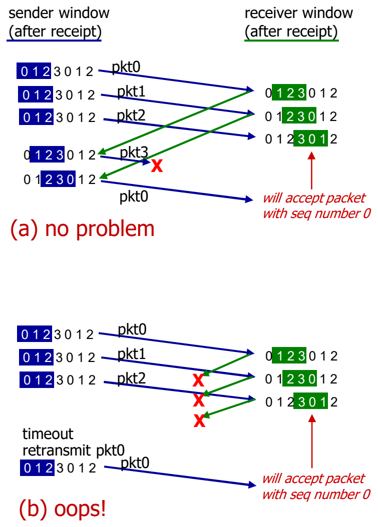


72

Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



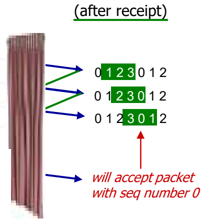
Selective Repeat: a Dilemma!

Example:

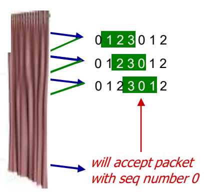
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

sender window
(after receipt)

receiver window
(after receipt)



- *receiver can't see sender side*
- *receiver behavior identical in both cases!*
- *something's (very) wrong!*



Q: What relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

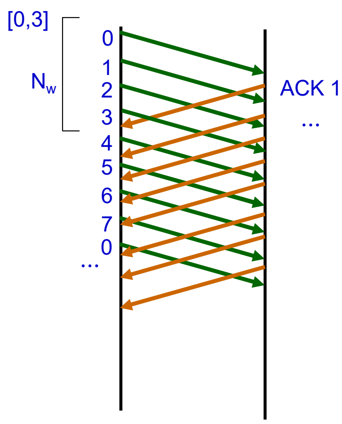
Sliding Window: Continuous Transmission

Example: ***Selective Repeat***

- Numbering modulo N = 8
  (Identifiers: 0, 1, ... 7)
- **Max window size: N_w = N/2 = 4**
- Max usage (**efficiency**) is: $U = 1$

(assuming all bits in the frames are useful bits)

In this case, it is never needed to stop waiting for an ACK (sender window is never fully used).



76

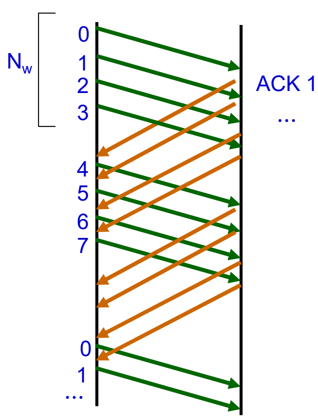
Sliding Window: Discontinuous Transmission

Example: ***Selective Repeat***

- Numbering modulo N = 8
  (Identifiers: 0, 1, ... 7)
- Max window size: N_w = N/2 = 4
- Assuming the processing and ACK transmission times can be ignored, the usage (**efficiency**) is:

$$U = \frac{N_w \cdot t_{frame}}{2 \cdot t_{prop} + t_{frame}} = \frac{N_w}{2 \cdot a + 1}$$

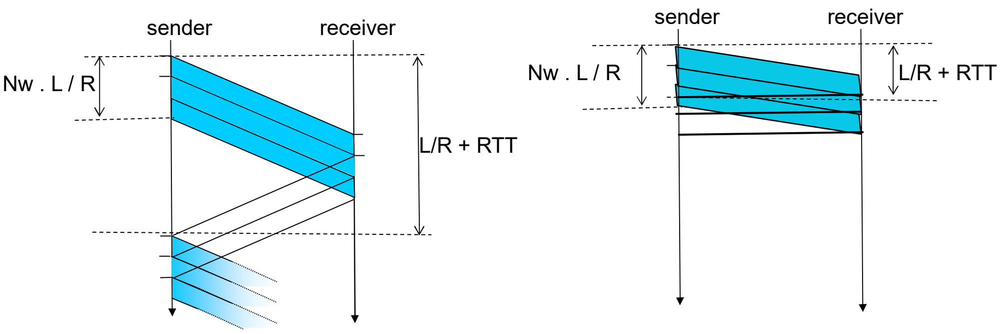
(assuming all bits in the frames are useful bits)



77

TÉCNICO LISBOA

Sliding Window Protocols: Utilization



If (L/R + RTT) > Nw . L/R

$$U = \frac{Nw \,.\, L/R}{RTT + L/R}$$

else:

U = 1

TÉCNICO LISBOA

Sliding Window: Efficiency

TPC: Prob. 6 e 7

Continuous transmission: U = 1

Discontinuous transmission: $U = N_w / (2.a+1)$

Piggybacking – improves line usage in bidirectional connections:

- If a station has data and ACKs to send, the ACKs are temporarily delayed and included in the data frames;
- The ACK only uses a few bits in the information frame header, while a separate ACK frame needs the full header and FCS;
- Possible disadvantage: sender may *timeout* if the wait for the *piggybacking* ACK is too long.

Flow Control

ARQ error control techniques also allow to perform flow control, by controlling the cadence of ACK transmission.

Chapter 3 (part 1): Summary

- Principles behind transport layer services:
  - Multiplexing and demultiplexing;
  - Reliable data transfer;
  - Flow control;

Next:
- Leaving the network "edge" (application, transport layers);
- Into the network "core".