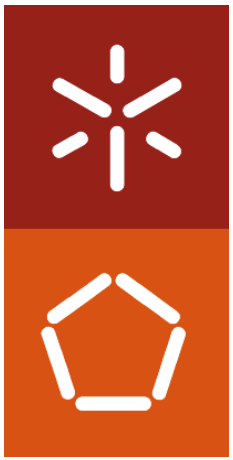
Nível de Transporte

Mestrado Integrado em Engenharia

de Telecomunicações e Informática

3º ano - 2º Semestre

2015/2016

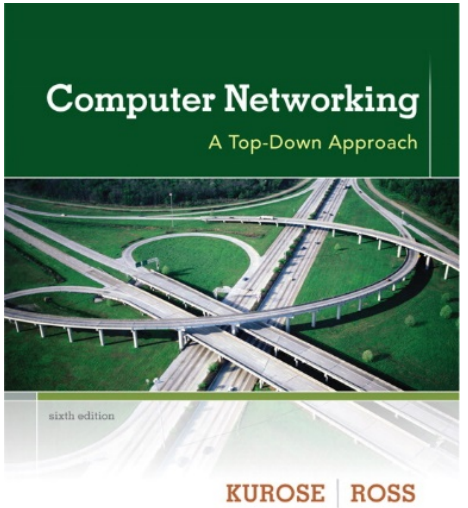


Sumário

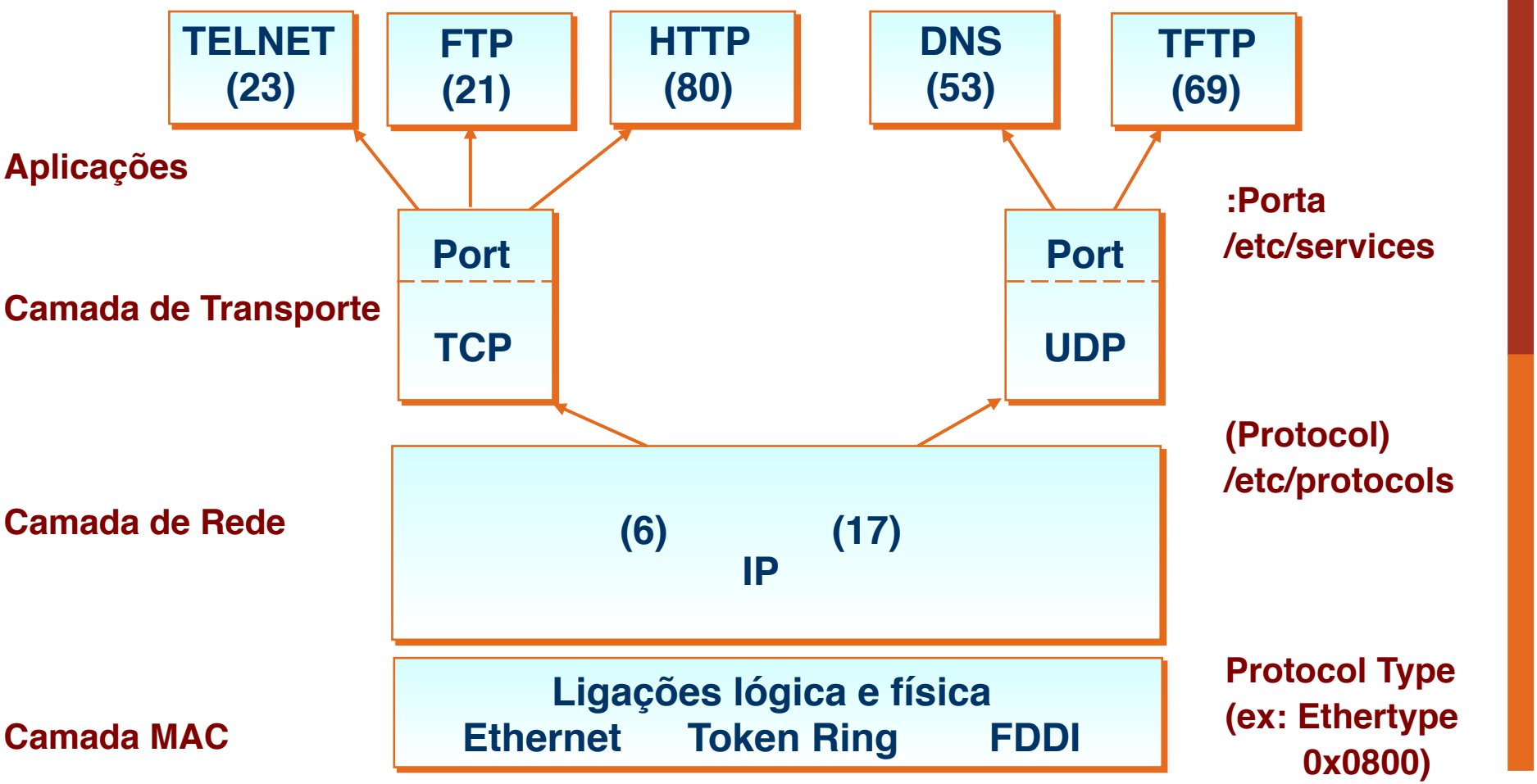
Sumário

- **A camada de transporte das redes TCP/IP**
- **Serviços e protocolos de transporte**
- **Transporte vs. rede**
- **Multiplexagem/desmultiplexagem**
  - conceito de porta de protocolo
- **O protocolo UDP**
  - caracterização do serviço UDP
- **O protocolo TCP**
  - caracterização do serviço TCP
  - controlo de fluxo
  - controlo de congestionamento

Estes slides são maioritariamente baseados no livro: *Computer Networking: A Top-Down Approach Featuring the Internet, Jim Kurose and Keith Ross,* Addison-Wesleyy

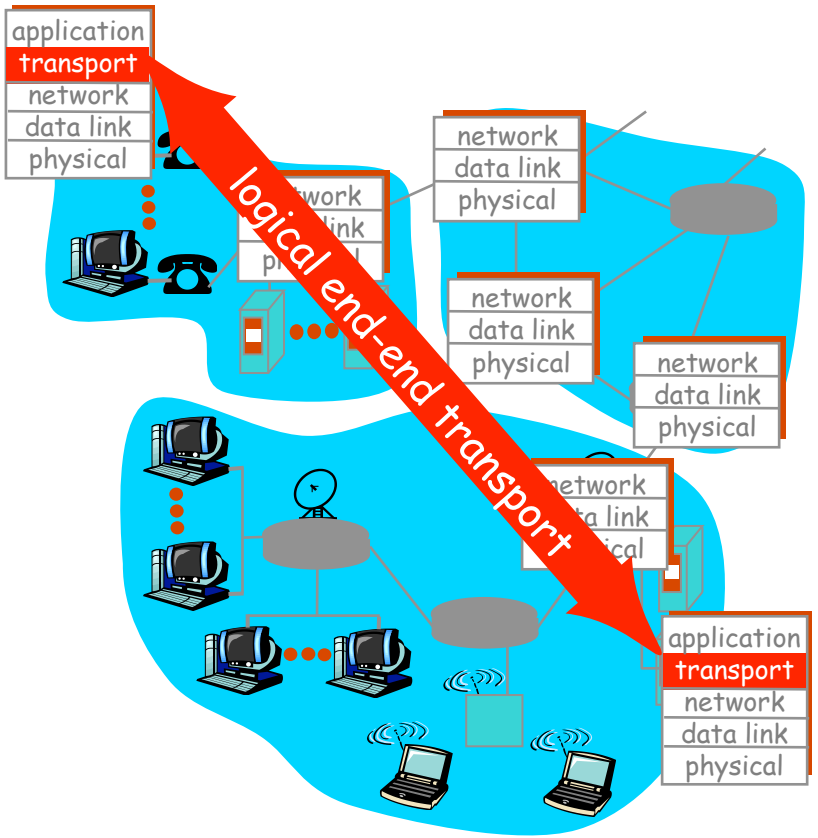


TCP/IP: protocolos de transporte UDP e TCP

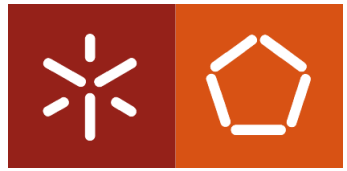


Serviços e protocolos de transporte

- **Disponibiliza uma ligação lógica entre aplicações (processos) que estão a ser executadas em Sistemas Terminais diferentes**
- **Os protocolos de transporte são executados nos Sistemas Terminais**
  - O emissor parte a mensagem gerada pela aplicação em segmentos que passa à camada de rede
  - O receptor junta os diferentes segmentos que constituem uma mensagem que passa à respectiva aplicação
  - Internet: TCP e UDP



Serviços de transporte



- **Existem dois tipos de serviço de transporte:**

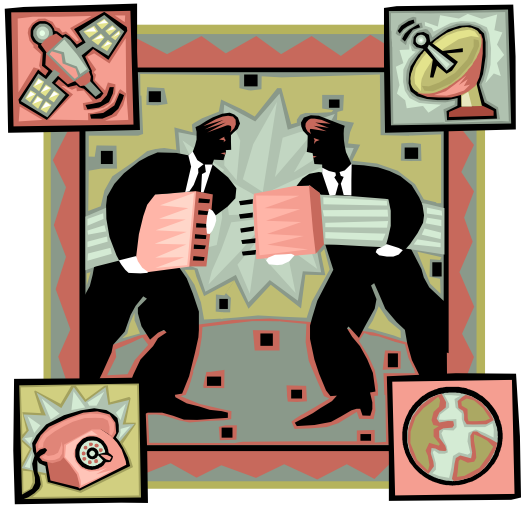
Orientado à conexão

- Estabelece, mantém e finaliza a conexão lógica entre processos
- Tem uma vasta variedade de aplicações
- Mais comum
- Fornece um serviço fiável

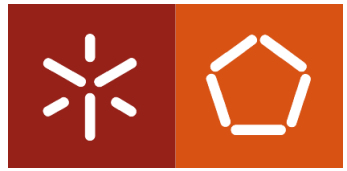
Não orientado à conexão

Estabelecimento e finalização da conexão

- **Serve três grandes propósitos:**
  - Permite assegurar que o sistema terminal com o qual se vai estabelecer a ligação existe
  - Permite a troca e a negociação de parâmetros adicionais
  - Permite a alocação de recursos nas entidades envolvidas no serviço de transporte
- **É feita por mútuo acordo**



Transporte *versus* Rede



- **Camada de Rede: fornece uma ligação lógica entre dois sistemas terminais**
- **Camada de Transporte: fornece uma comunicação lógica entre dois processos**
  - Usa e melhora os serviços disponibilizados pela camada de Rede
  - Troca de Dados Fiável e ordenada (TCP)
    - Controlo de congestão
    - Controlo de fluxo
    - Estabelecimento da ligação
  - Troca de dados não fiável e desordenada (UDP)
  - Serviços não disponíveis
    - Garantia de atraso máximo e largura de banda mínima

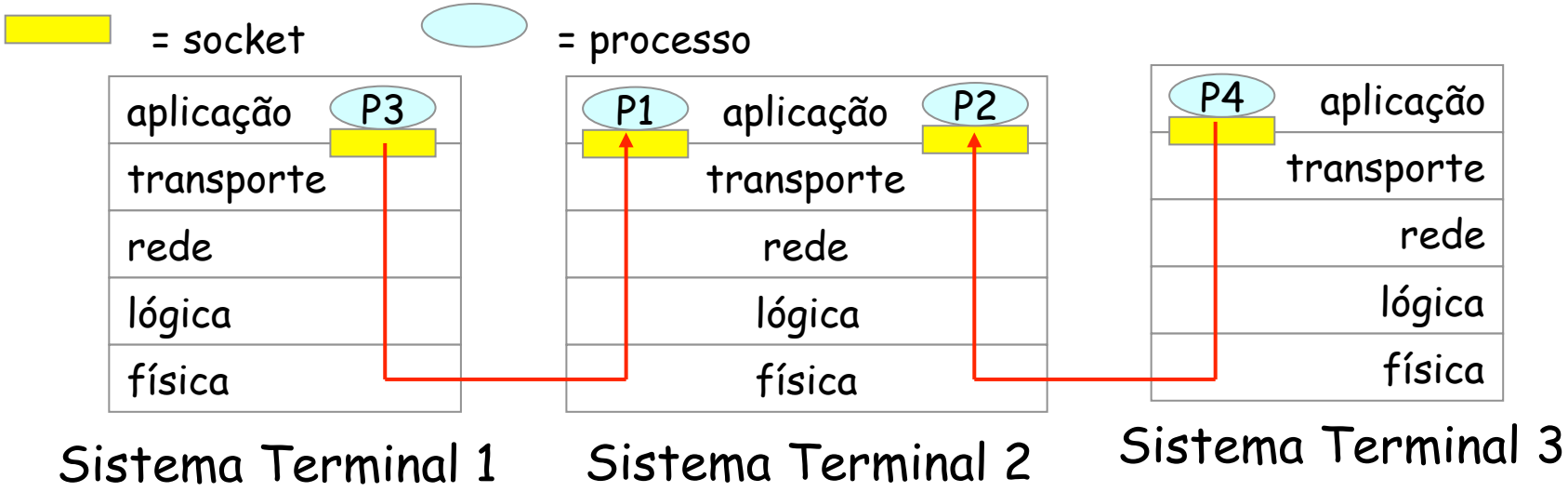
Multiplexagem/desmultiplexagem

Multiplexagem no emissor:

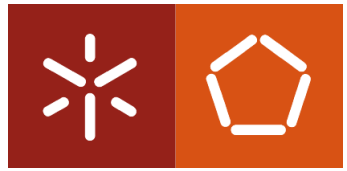
Recolher os dados de diferentes *sockets* e delimitá-los com os respectivos cabeçalhos construindo os respectivos segmentos

Desmultiplexagem no receptor

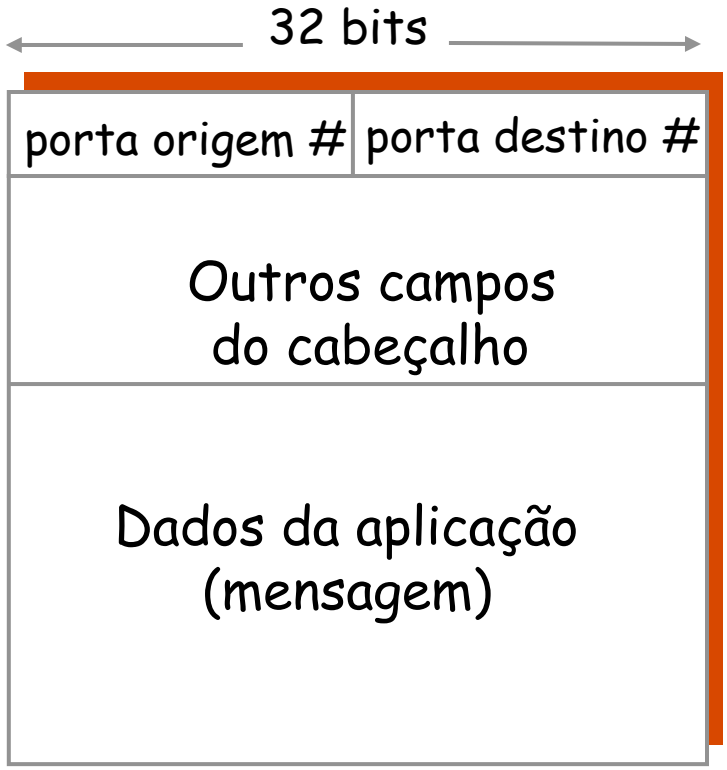
Entregar os diferentes segmentos ao *socket* correcto.



Desmultiplexagem

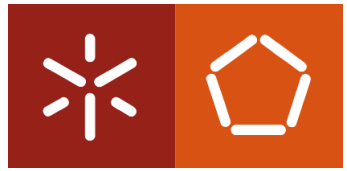


- **É efectuada pelo sistema terminal destino ao receber um datagrama IP**
  - Cada datagrama contém um segmento TCP ou UDP
  - Cada segmento possui a identificação da porta de origem e da porta destino.
  - O sistema terminal usa os endereços IP e os números de porta para encaminhar o segmento para o socket correcto



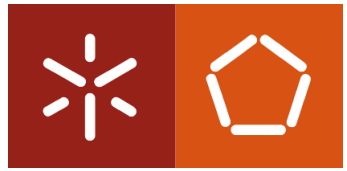
TCP/UDP segment format

UDP – User Datagram Protocol [RFC 768]



- **Funções do User Datagram Protocol:**
  - protocolo de transporte fim-a-fim, não fiável
  - orientado ao datagrama (sem conexão)
  - actua como uma interface da aplicação com o IP para multiplexar e desmultiplexar tráfego
  - usa o conceito de porta / número de porta
    - forma de direccionar datagramas IP para o nível superior
    - portas reservadas: 0 a 1023
    - portas dinâmicas: 1024 a 65535
  - é utilizado em situações que não justificam o TCP
    - exemplos: TFTP, RPC, DNS
  - ou em que o tempo é crítico:
    - exemplos: *streaming* de áudio e vídeo

Desmultiplexagem no UDP

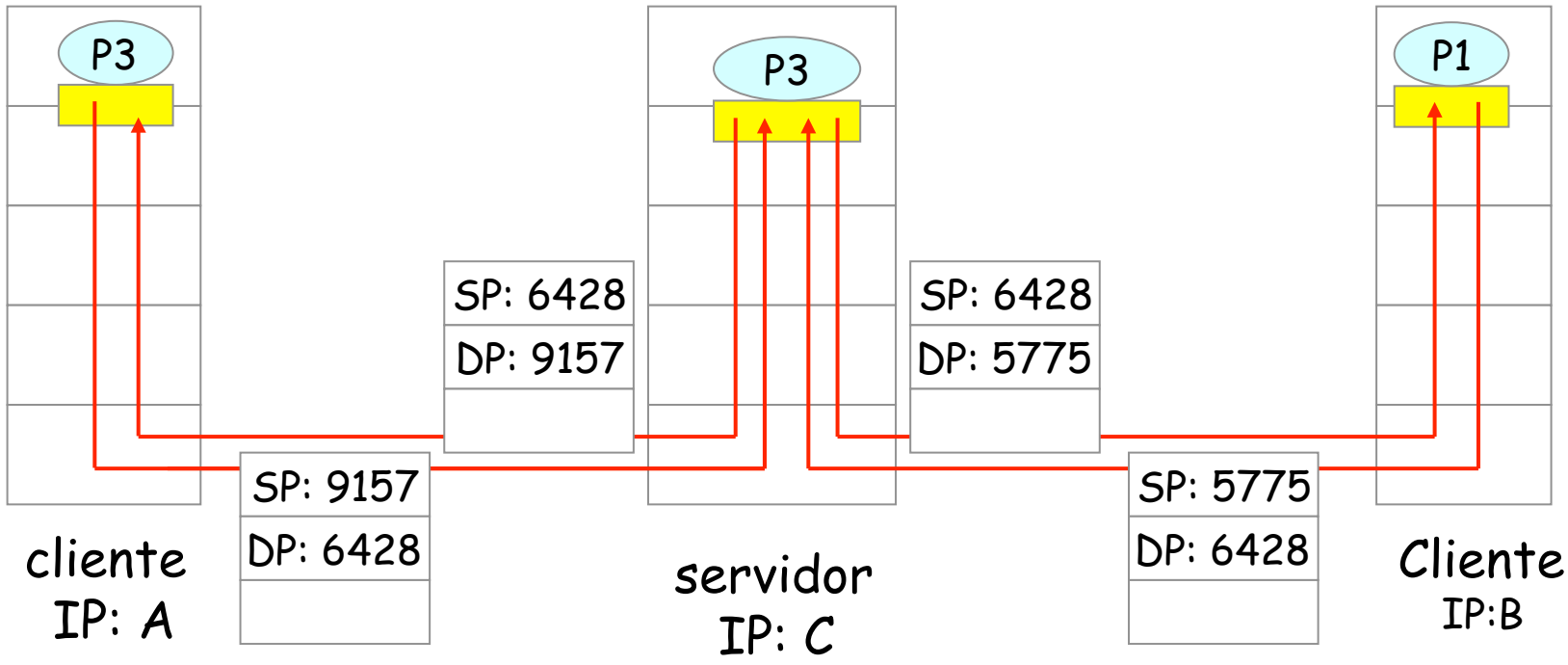


- O socket UDP é identificado através de dois números: endereço IP destino, e número de porta destino (dest IP address, dest port number)

- Quando um Sistema Terminal recebe um datagrama UDP verifica qual o número da porta destino que consta do datagrama UDP e redirecciona o datagrama para o socket com esse número de porta

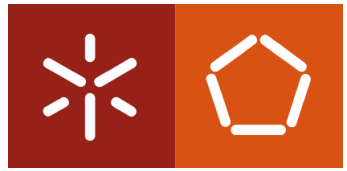
- Datagramas com diferentes endereços IP origem e/ou portas origem podem ser redireccionados para o mesmo socket

Desmultiplexagem no UDP



SP fornece o "return address"

Serviço UDP

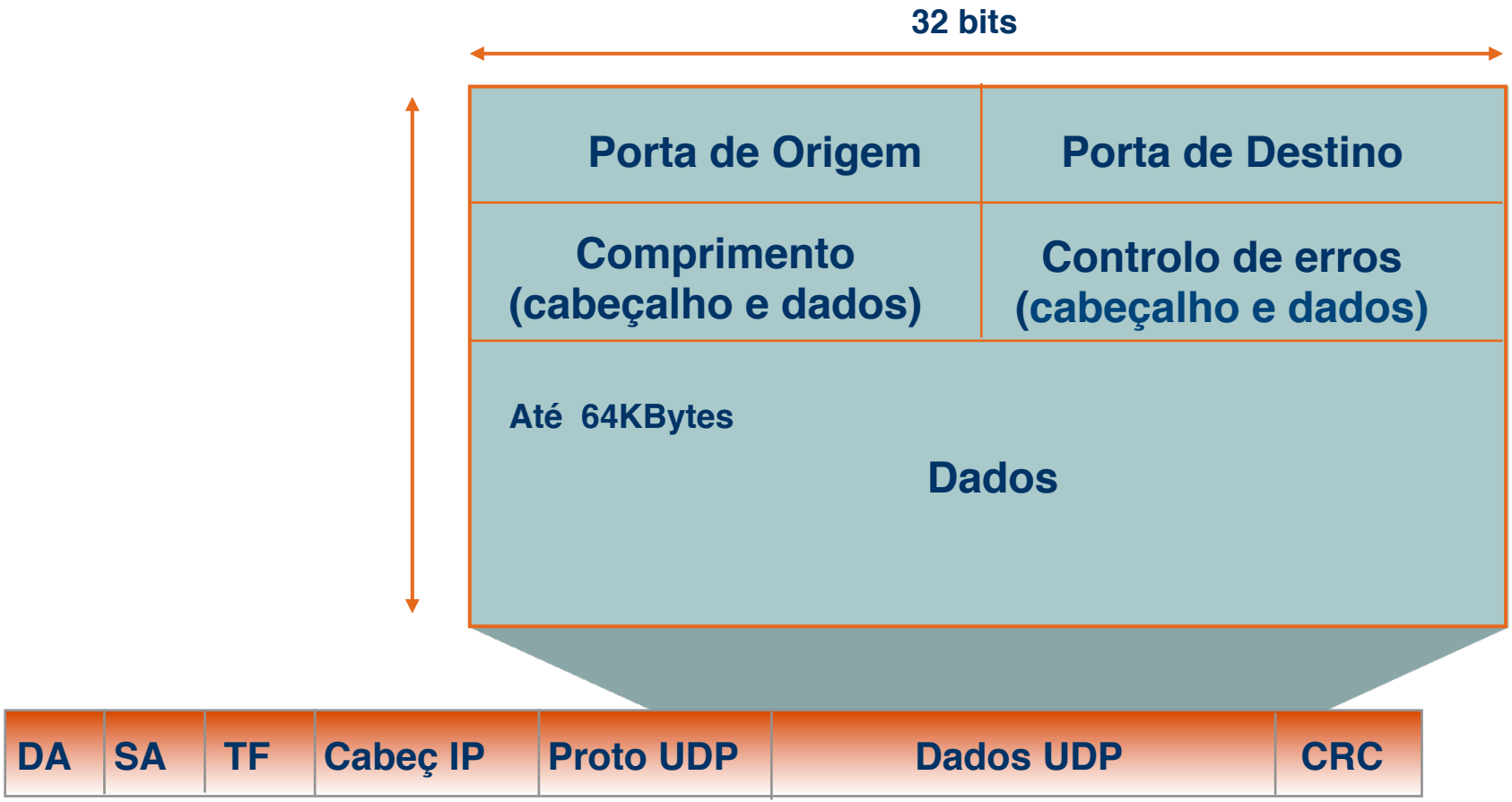


- **Serviço "best effort"; os datagramas UDP podem ser:**
  - perdidos
  - entregues fora de ordem
- **Não orientado à conexão:**
  - não há *handshaking* entre emissor e o receptor
  - cada segmento UDP é processado de forma independente

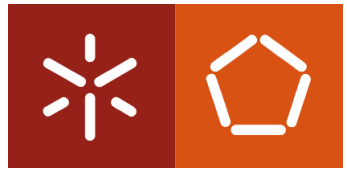
Porque existe UDP?

- **não há estabelecimento de conexão (menos atraso)**
- **simples: não há variáveis de estado**
- **tamanho do cabeçalho é pequeno**
- **não há controlo do congestionamento: UDP envia sempre que precisa**

Datagramas UDP



UDP - fiabilidade



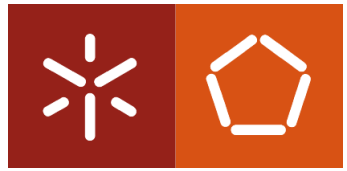
- **Serviço não fiável:**
  - sem garantia de entrega dos datagramas
  - sem garantia de entrega ordenada dos datagramas

- **Com detecção de erros (*checksum*):**
  - complemento para 1 da soma de grupos de 16 bits
  - cobre o datagrama completo (cabeçalho e dados)

- **Sem correcção de erros**

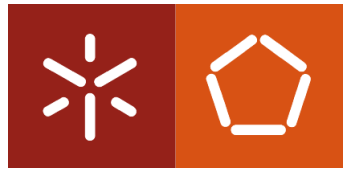
TCP – Transmission Control Protocol

[RFCs 793, 1122, 1323, 2018, 2581]



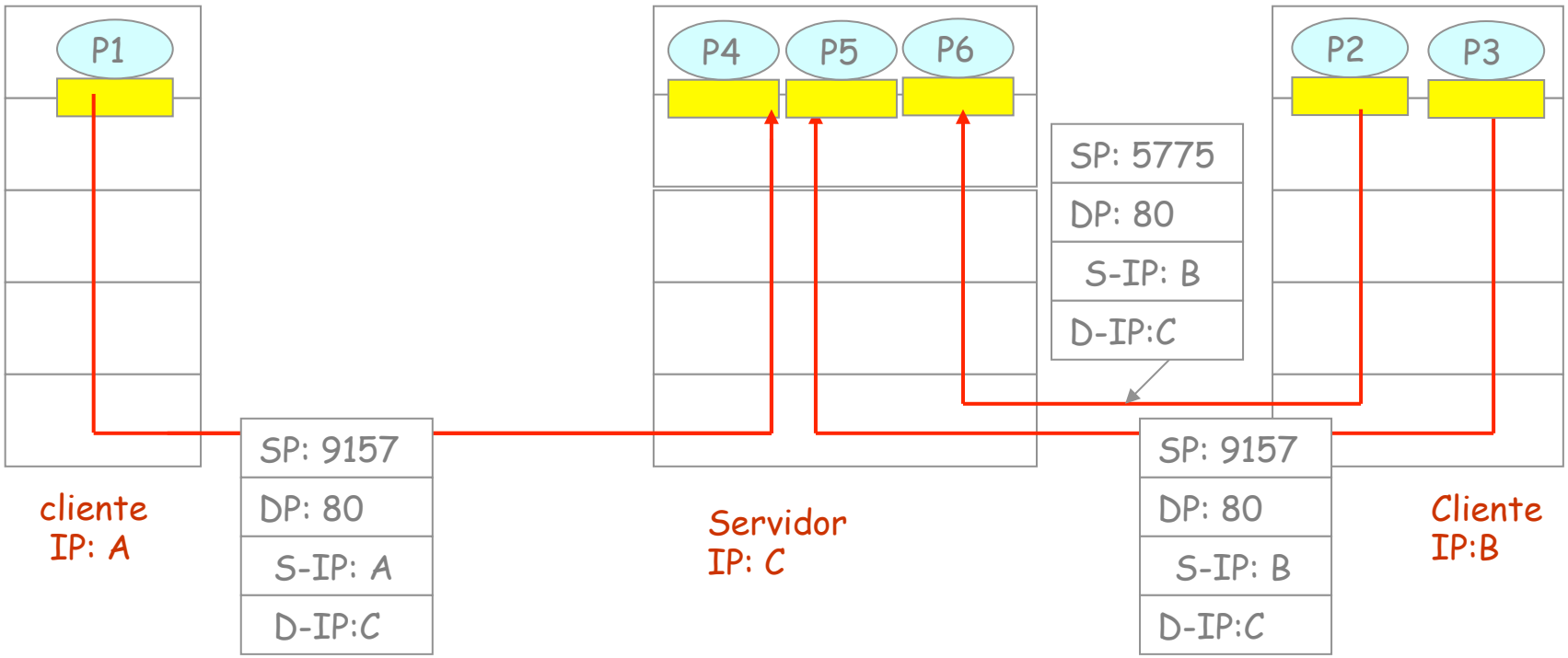
- **Funções do Transmission Control Protocol:**
  - transporte fiável de dados fim-a-fim (aplicações)
  - efectua associações lógicas fim-a-fim: conexões
  - cada conexão é identificada por um par de sockets: (IP_origem:porta_origem, IP_destino:porta_destino)
  - uma conexão é um circuito virtual entre portas de aplicações (também designadas portas de serviço)
  - multiplexa os dados de várias aplicações através de número de porta
  - efectua controlo de fluxo, de congestão e de erros

Desmultiplexagem no TCP

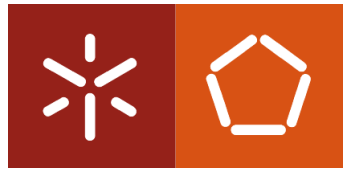


- **O socket TCP é identificado pela 4-tupla:**
  - Endereço IP de origem
  - Número de porta de origem
  - Endereço IP de destino
  - Número de porta de destino
- **O sistema terminal que recebe os segmentos usa estes 4 parametros para encaminhar os segmentos para o socket apropriado**
- **Um Servidor suporta múltiplos sockets em simultâneo:**
  - Cada socket é identificado pela sua tupla de 4 parâmetros
- **Servidores Web têm diferentes sockets para cada conexão do cliente**

Desmultiplexagem no TCP

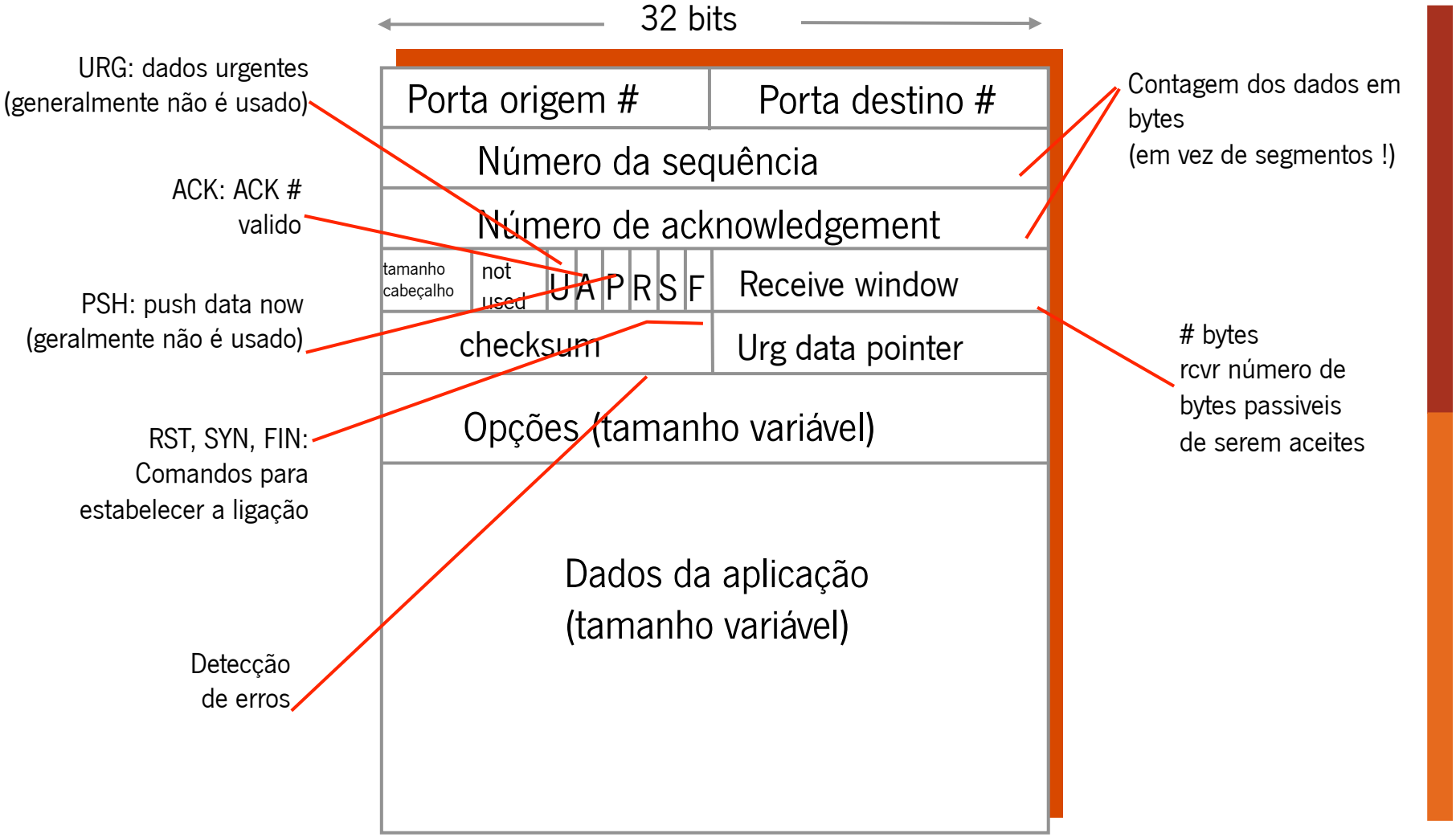


Serviço TCP



- **Serviço *full-duplex***
- **Serviço fiável:**
  - garantia de entrega dos dados
  - garantia de entrega ordenada
  - detecção de erros
  - correcção de erros por retransmissão
- **Orientado à conexão:**
  - estabelecimento de conexão entre dois processos
  - transmissão de dados (nos dois sentidos)
  - finalização da conexão
- **Com controlo de fluxo fim-a-fim**
  - *buffers* nos dois extremos da ligação

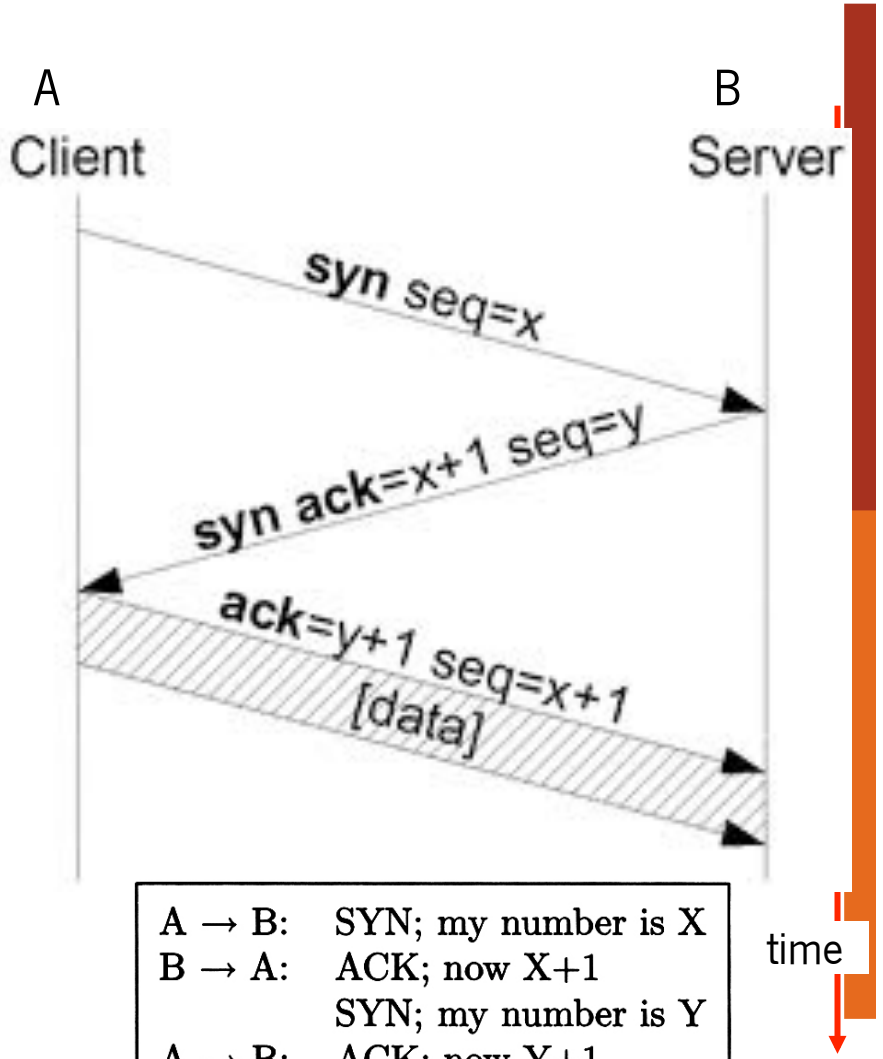
Segmento TCP



Handshaking no TCP

A troca de mensagens neste processo de handshake triplo é a seguinte:

- O primeiro segmento pode ser identificado porque tem o bit SYN igual a 1.
- A segunda mensagem tem o bit SYN e ACK activos indicando que se trata de uma confirmação do segmento SYN.
- A mensagem final é só uma mensagem de confirmação que indica ao destinatário que ambos os lados concordam que foi estabelecida uma ligação.



A → B:	SYN; my number is X
B → A:	ACK; now X+1
	SYN; my number is Y
A → B:	ACK; now Y+1
	(start talking)

TCP: seq. #'s e ACKs

Seq. #'s:

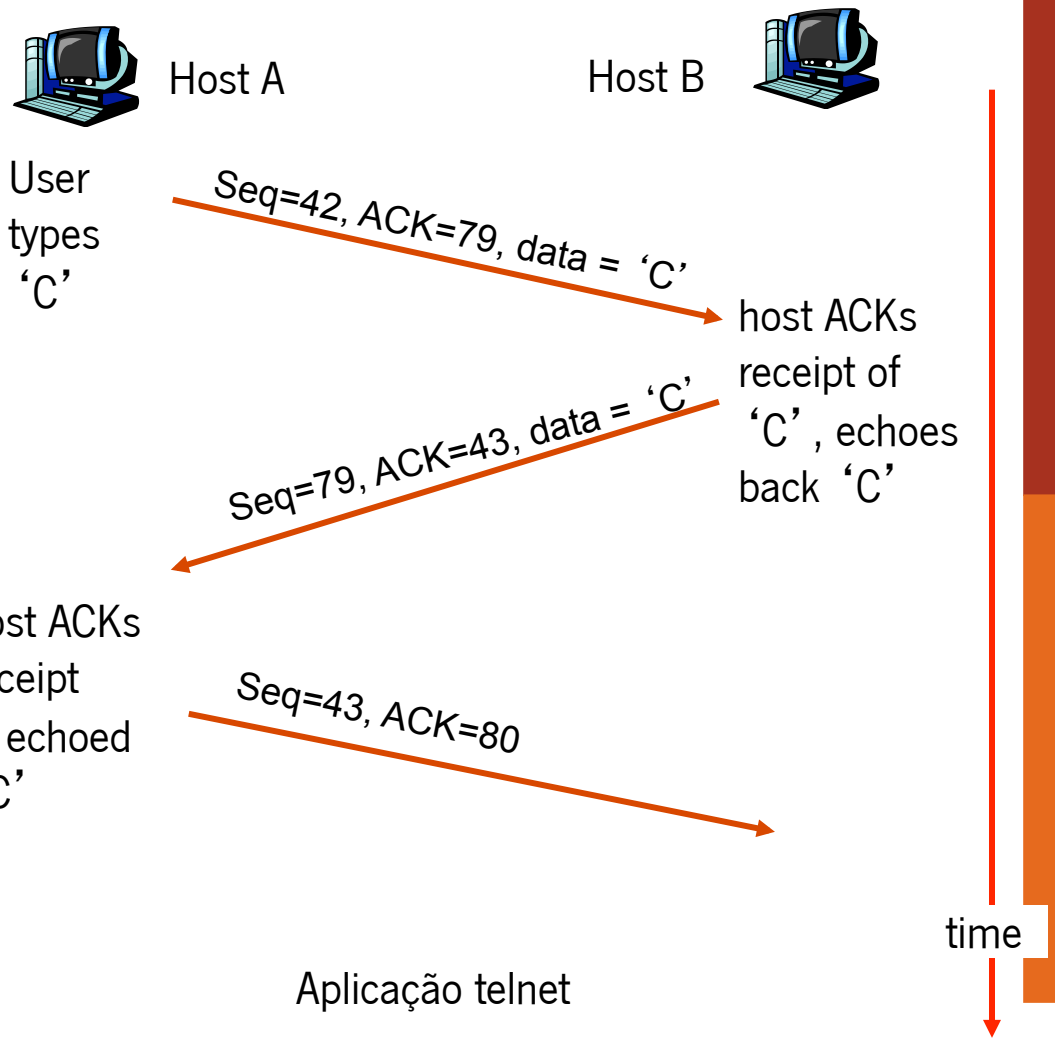
- Número do primeiro byte do segmento

ACKs:

- seq # do próximo byte esperado do host A
- ACK cumulativo

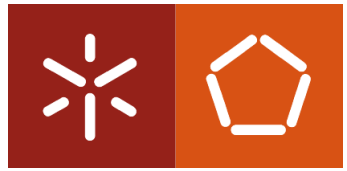
Q: como é que o receptor lida com os segmentos desordenados

- A especificação do TCP não diz como se implementa...



Aplicação telnet

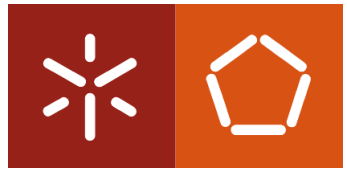
TCP: seq. #'s e ACKs



- sequenciação necessária para ordenação na chegada
- o *número de sequência* é incrementado pelo número de bytes do campo de dados
- cada segmento TCP tem de ser confirmado (ACK), contudo é válido o ACK de múltiplos segmentos
- o campo ACK indica o próximo byte (*sequence*) que o receptor espera receber (*piggyback*)
- o emissor retransmite por *timeout:* o protocolo define o tempo máximo de vida dos segmentos ou MSL (maximum segment lifetime)

TCP/IP

TCP Round Trip Time e Timeout

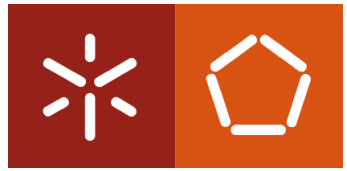


- **No TCP não há confirmações negativas**
  - Por esse motivo o emissor pode apenas **desconfiar** que um determinado segmento enviado não chegou ao destino

    Por *timeout* ou através da recepção de *Acks* duplicados

- **Como definir o valor do Timeout no TCP?**
  - Com base no RTT (intervalo de tempo deste que se envia um segmento até que se recebe a respectiva confirmação
  - As amostras RTT variam, a estimativa do RTT deve ser "smoother"
    - Média das várias medidas recentes, e não só da amostra mais recente

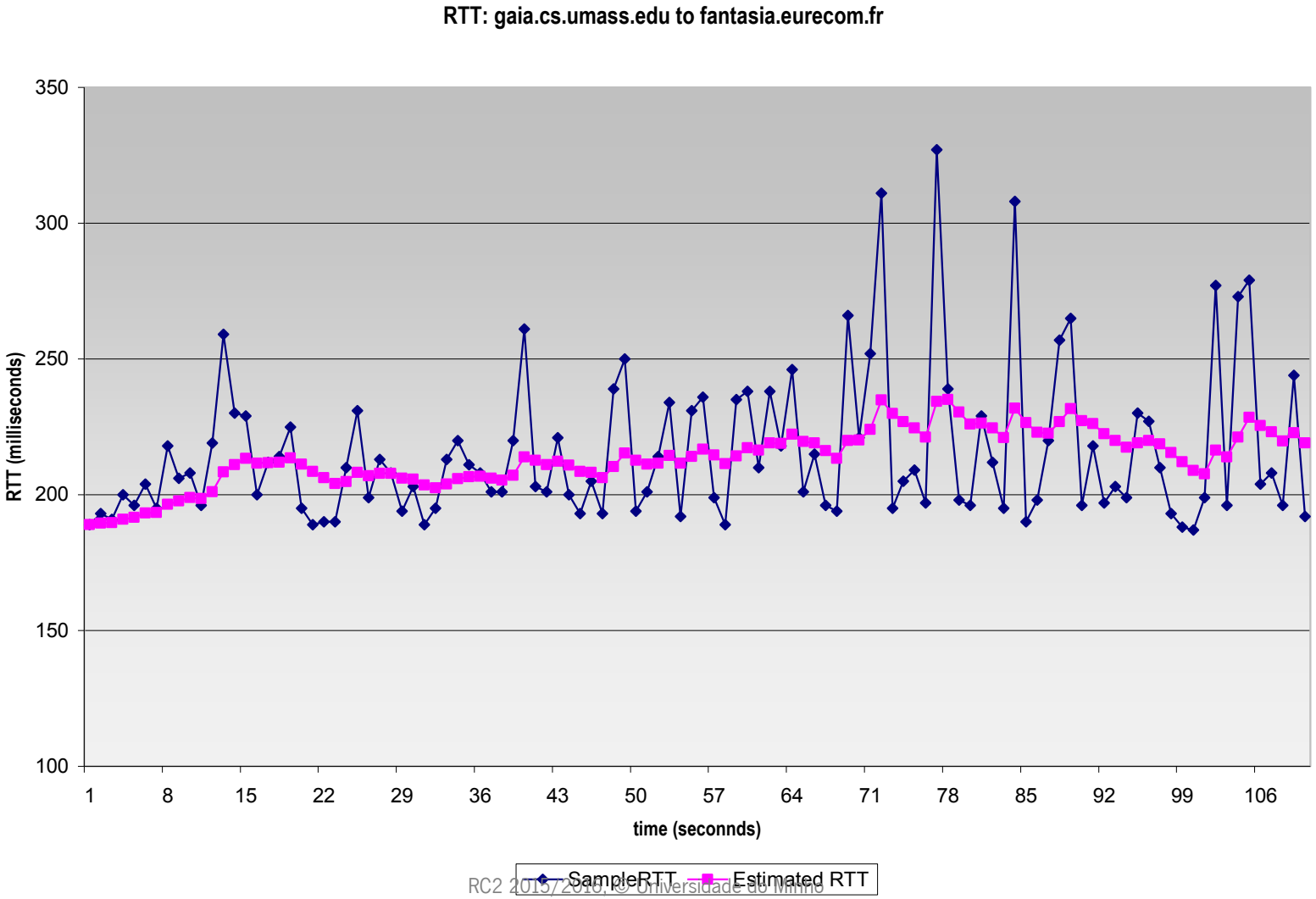
TCP: Round Trip Time e Timeout



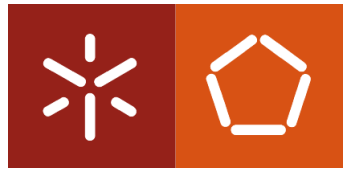
$$RTT\_estimado = (1- \alpha) \times RTT\_estimado + \alpha \times AmostraRTT$$

- **A influência das mostragens passadas decresce exponencialmente**
- **RTT_estimado - Média móvel de peso exponencial onde a importância de uma amostra passada decresce exponencialmente**
- **O AmostraRTT é a medida de tempo desde a transmissão de um segmento até à recepção do Ack respectivo**
- **Valor típico de α 0.125**

TCP: Round Trip Time e Timeout



TCP: Round Trip Time e Timeout



Configurar o timeout

- **RTT_estimado mais uma "margem de segurança "**
  - **Uma variação grande no RTT_estimado -> uma margem de segurança maior**
- **Estimar quanto é que a AmostraRTT se afasta (desvia) do valor RTT_estimado:**

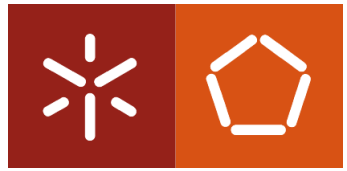
$$DesvioRTT = (1-\beta)*DesvioRTT + \beta*|AmostraRTT-RTT\_estimado|$$

$$(tipicamente,\ \beta = 0.25)$$

Depois configura-se o intervalo do timeout:

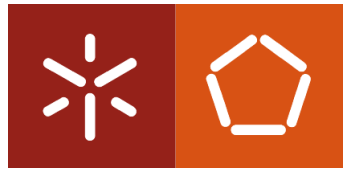
$$IntervaloTimeout = RTT\_estimado + 4*DesvioRTT$$

TCP: transferência fiável de dados



- **O TCP cria um serviço fiável de transferência de dados no topo do serviço IP não fiável**
- **O TCP utiliza ACKs cumulativos**
- **O TCP usa um único relógio para as retransmissões**
- **As retransmissões são despoletadas por:**
  - Eventos de timeout
  - ACKs duplicados

TCP: eventos no emissor



- Cria um segmento com seq #
- seq # é número do primeiro byte no segmento
- Inicia o relógio se ainda não estiver inicializado
- O intervalo de expiração: `TimeOutInterval`

Timeout:

- retransmite o segmento que causou o timeout
- reinicializa o relógio

ACK rcvd:

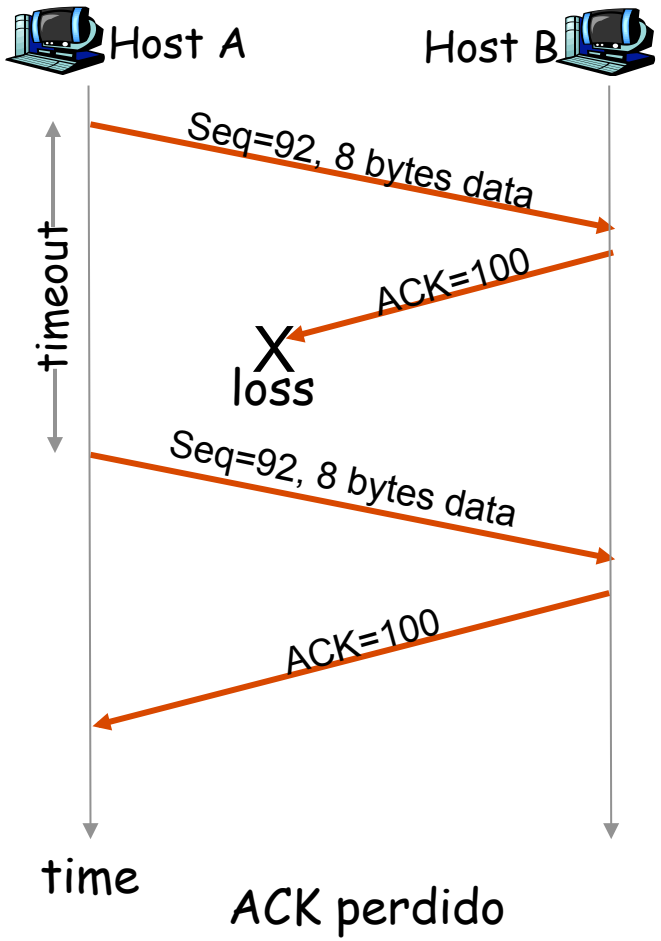
- se receber um ACK, os segmentos previamente enviados são também confirmados
- actualiza os segmentos confirmados com o ACK
- inicia o relógio se houver mais segmentos em trânsito

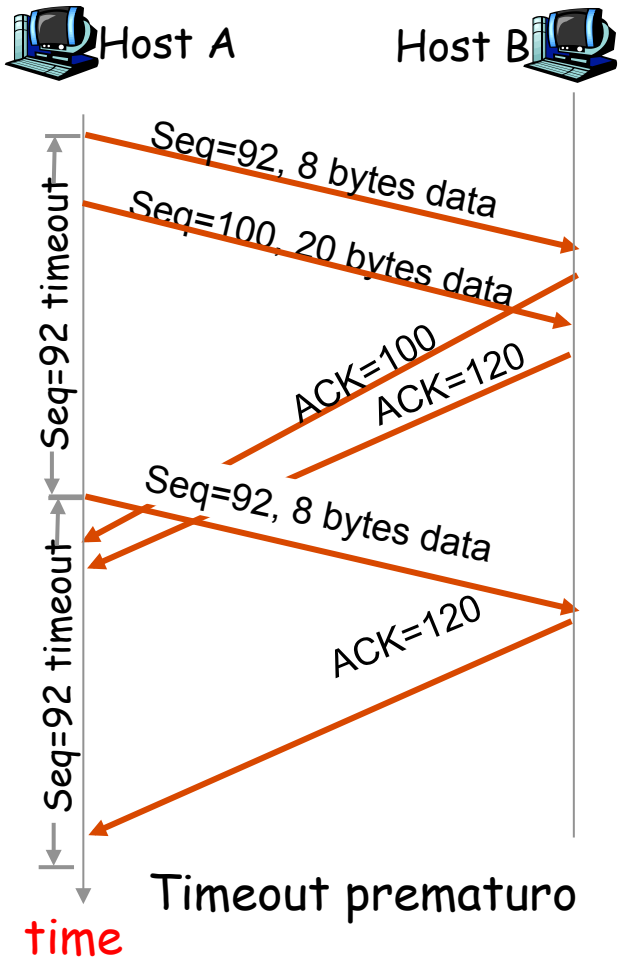
Entidade TCP (sender)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
          smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
              start timer
          }

} /* end of loop forever */
```

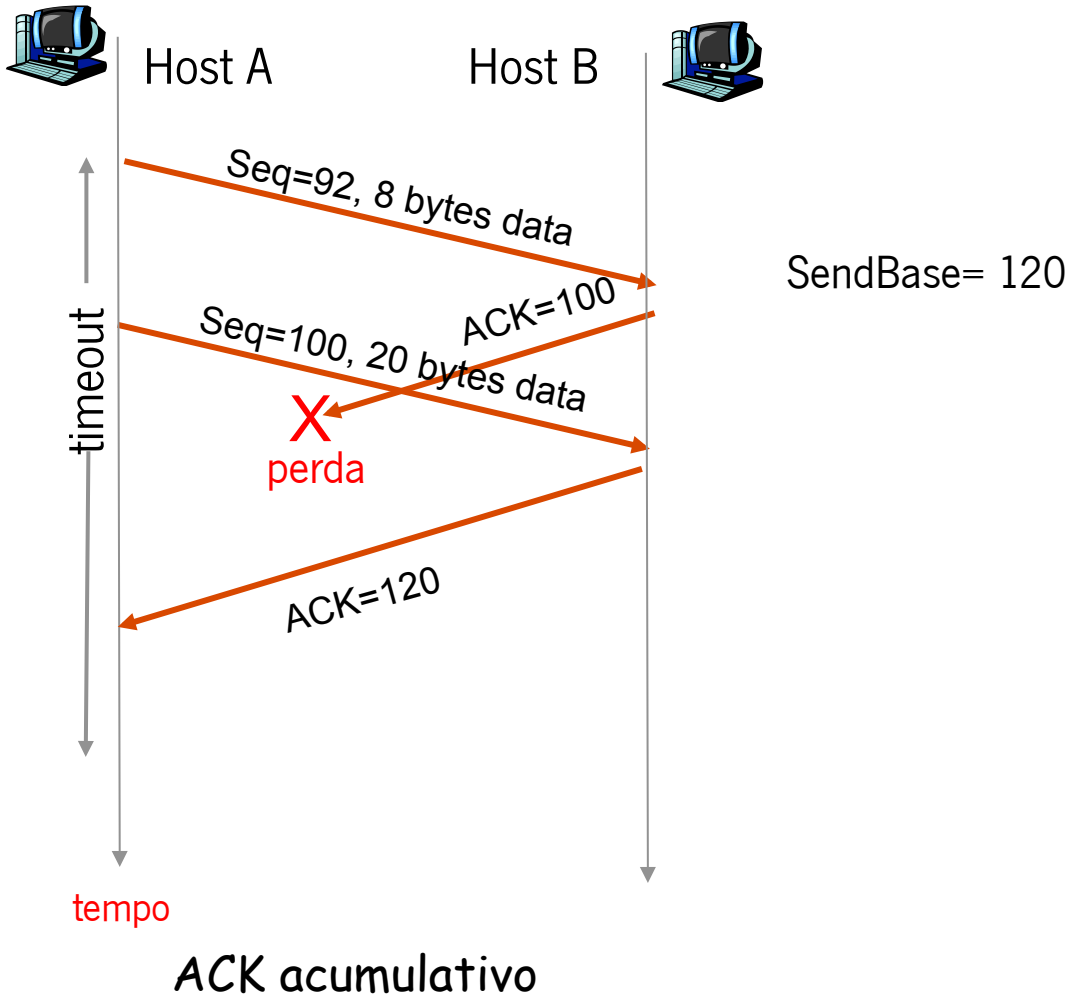
NOTA: As retransmissões são controladas por um único timer

TCP: exemplos de retransmissão





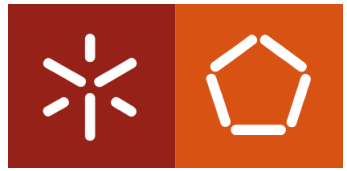
TCP: exemplos de retransmissão



SendBase= 120

ACK acumulativo

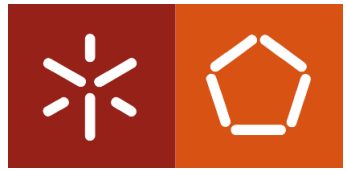
TCP: eventos no recetor



Evento no Recetor	Acção da entidade TCP
Chegada de um segmento com o número de sequência esperado e tudo para trás confirmado.	Atrasa envio de ACK 500ms para Verificar se chega novo segmento. Senão chegar, envia ACK
Chegada de um segmento com o número de sequência esperado e um segmento por confirmar	Envia imediatamente um ACK cumulativo que confirma os dois Segmentos.
Chegada de um segmento com o número de sequência superior ao esperado. Buraco detectado	Envia imediatamente um ACK duplicado indicando o número de sequência esperado
Chegada de um segmento que preenche ou completa um buraco	Se o número do segmento coincidir com o limite inferior do buraco envia ACK imediatamente.

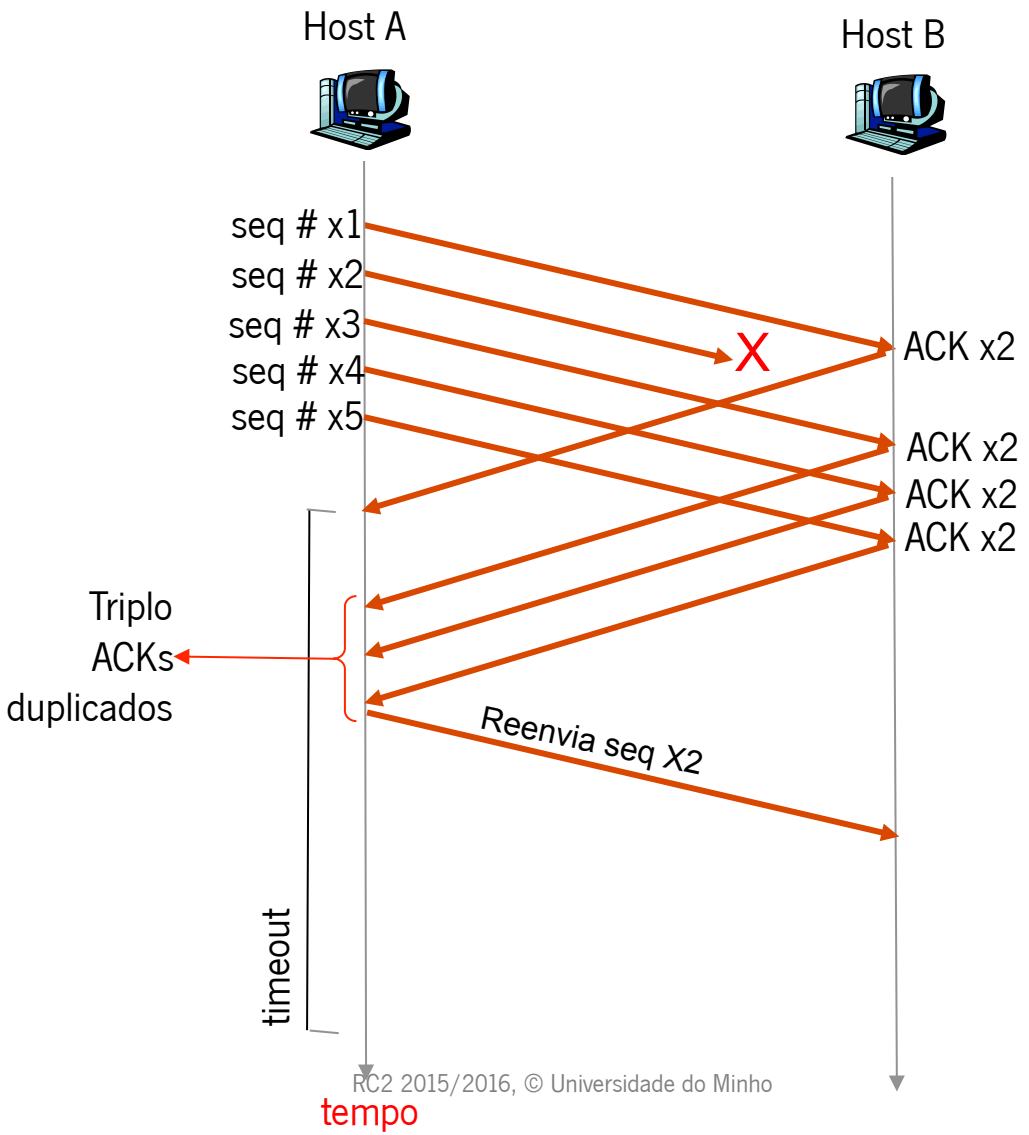
TCP/IP

Fast Retransmit



- **A duração do *timeout* é por vezes demasiado longa, o que provoca atrasos na retransmissão de um pacote perdido**
- **Para minimizar esse problema, o emissor procura detetar perdas através da receção de ACKs duplicados**
  - O emissor envia normalmente vários segmentos seguidos. No caso de algum deles se perder vai haver vários ACKs duplicados.
  - Se o emissor recebe três ACKs duplicados supõe que o segmento respetivo foi perdido de retransmiti-o (***Fast Restransmit e Fast Recovery***)

TCP: *fast retransmit*

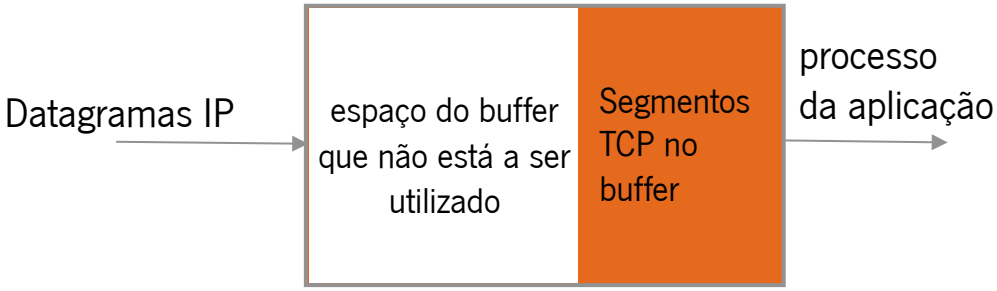


TCP: controlo de fluxo

- O lado do recetor de uma conexão TCP tem um buffer:

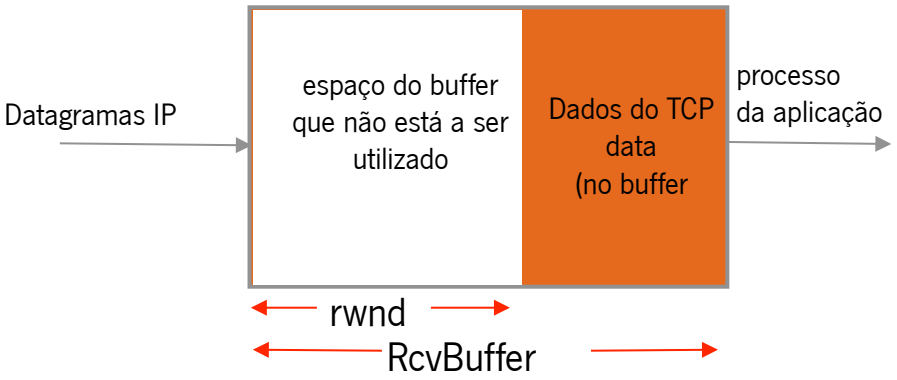
Controlo do fluxo

O transmissor não quer sobrecarregar o buffer do receptor com o envio de demasiados dados num curto espaço de tempo



- O processo da aplicação pode ser lento a ler os dados do buffer

TCP: controlo de fluxo



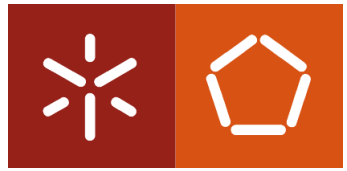
(suponha que o receptor TCP descarta os segmentos que chegam desordenados)

- Espaço do buffer não utlizado:

```
rwnd = RcvBuffer-[LastByteRcvd - LastByteRead]
```

- **O receptor**: informa o transmissor do espaço do buffer que ainda não está a ser utilizado, incluindo o valor do rwnd no cabeçalho do segmento
- **O transmissor**: limita o número de bytes não confirmados (unACKed) para rwnd
  - Isto garante que o buffer do receptor não fica sobrecarregado

TCP: gestão da conexão



- O emissor e o recetor TCP estabelecem uma ligação antes de iniciarem a troca de segmentos de dados.
- Inicializa as variáveis TCP:
  - Números de sequência - seq. #s
  - buffers, informação de controlo de fluxo (ex. **`RcvWindow`**)
- Cliente: inicia o pedido da conexão

```
Socket clientSocket = new
Socket("hostname","port
number");
```

- Servidor: é contactado pelo cliente e aceita o pedido da conexão ou ligação

```
Socket connectionSocket =
welcomeSocket.accept();
```

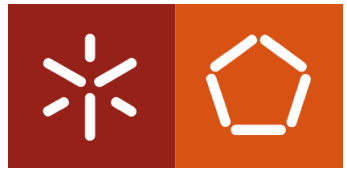
Three way handshake:

Passo 1: O cliente envia segmento SYN para o servidor
- Especifica o número inicial da sequência - seq #
- Não contém dados

Passo 2: O servidor recebe o SYN e responde com um segmento SYNACK
- Aloca espaço nos buffers
- especifica o número de sequência inicial - seq. #

Passo 3: O cliente recebe o segmento SYNACK, e responde com um segmento ACK que pode conter dados

TCP: gestão da conexão



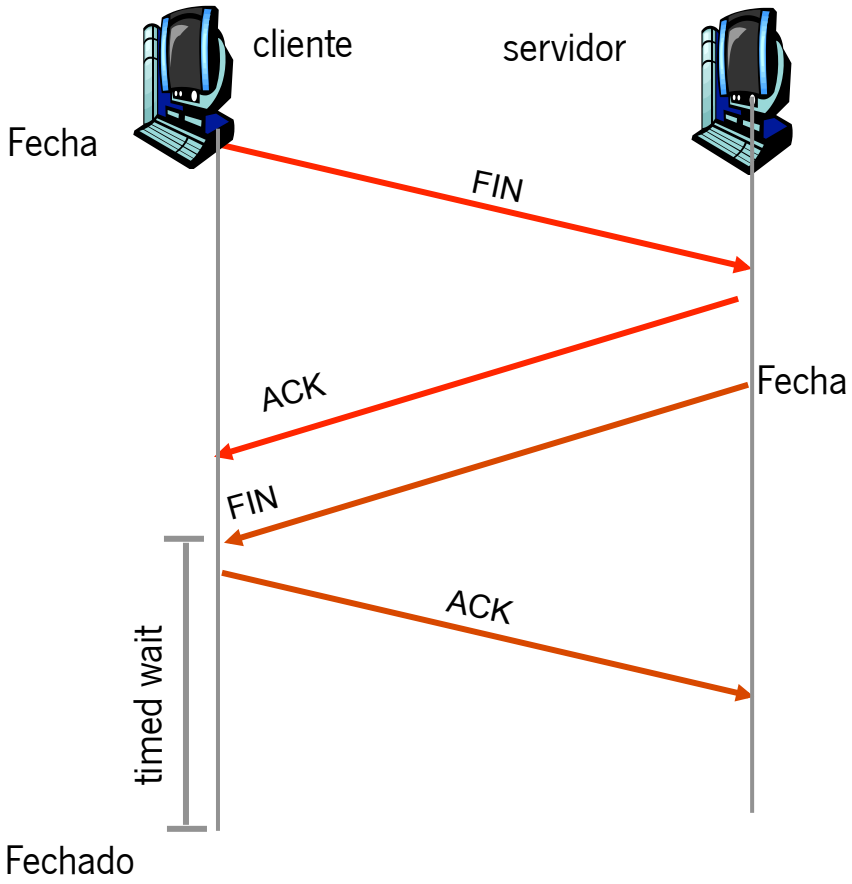
Terminar a conexão:

O cliente fecha o socket:

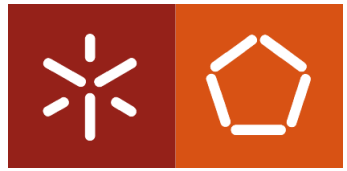
```
clientSocket.close();
```

Passo 1: O cliente envia um segmento de controlo TCP FIN para o servidor

Passo 2: O servidor recebe o FIN, e responde com um ACK. Fecha a conexão e envia um FIN.

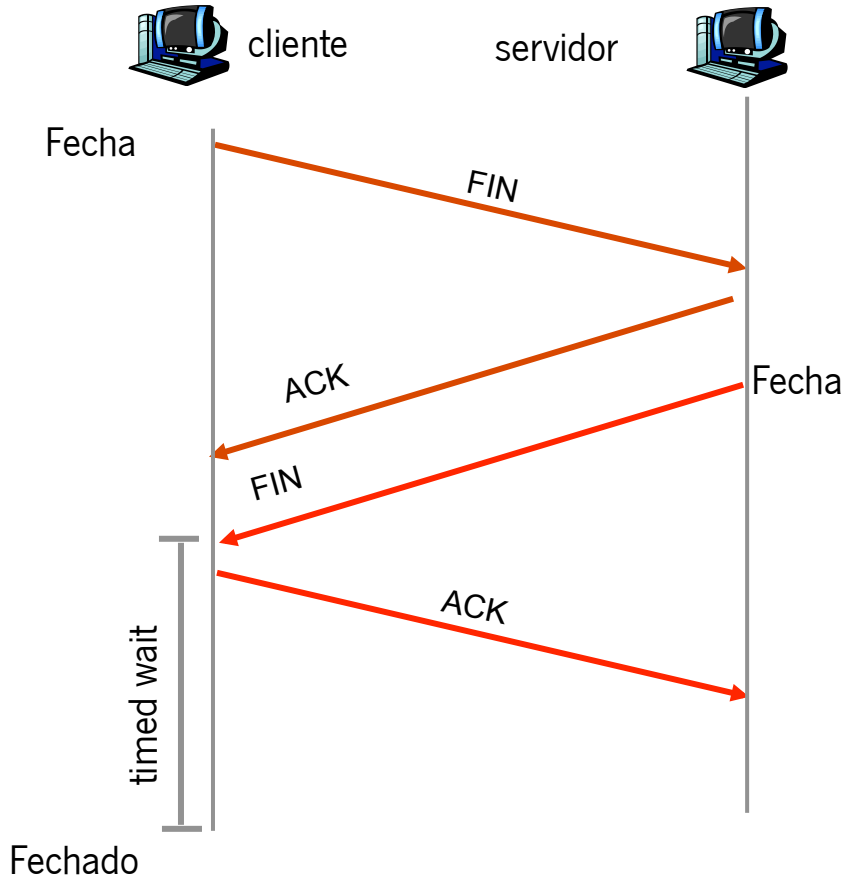


TCP: gestão da conexão

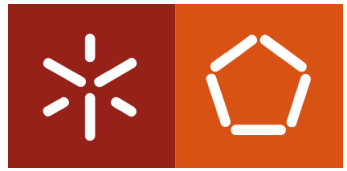


Passo 3: O cliente recebe o FIN, e responde com um ACK.

Passo 4: O servidor, recebe o ACK. A conexão fica fechada.

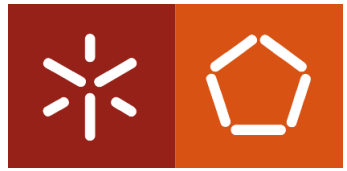


TCP: controlo de congestionamento

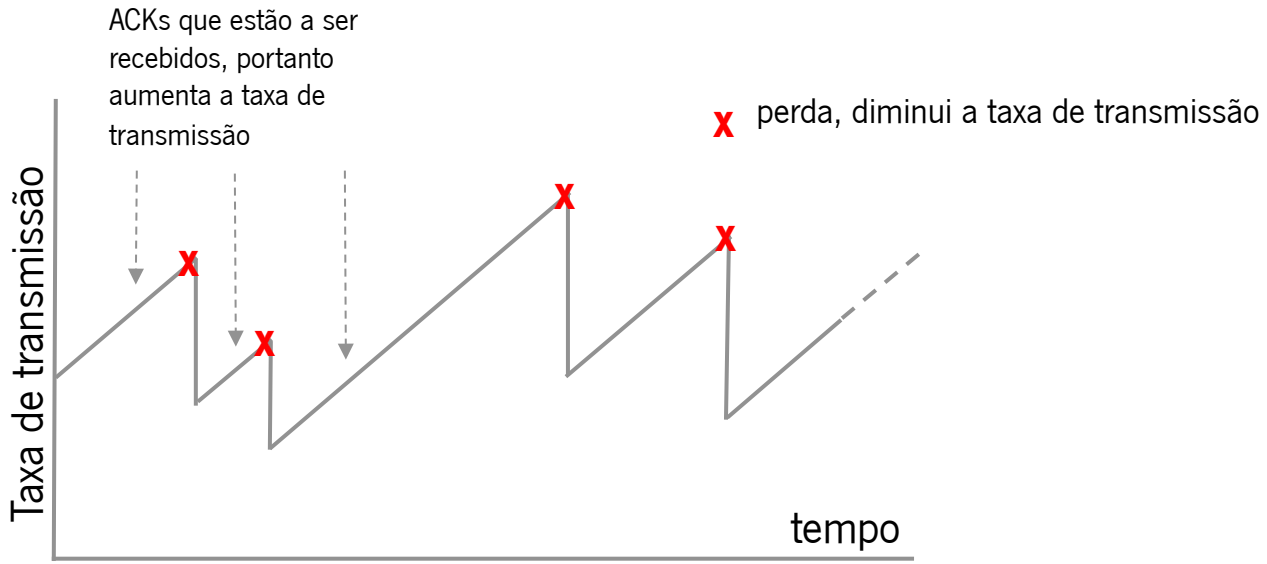


- **Objectivo: O transmissor TCP deve enviar tão rápido quanto possível, mas sem causar o congestionamento da rede**
  - Pergunta: Como encontrar a taxa de transmissão máxima, mas que fique abaixo do nível de congestionamento da rede?
- **Modelo descentralizado: cada transmissor TCP configura a sua própria taxa de transmissão com base no *feedback* implícito recebido:**
  - ACK: segmento recebido, a rede não está congestionada, portanto pode aumentar a taxa de transmissão
  - Segmento perdido: assume que a perda se deve ao congestionamento da rede, portanto diminui a taxa de transmissão

TCP: controlo de congestionamento



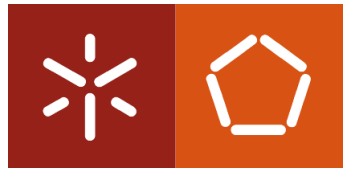
- **"Sonda a largura de banda": aumenta a taxa de transmissão se um ACK for recebido, até, eventualmente, ocorrer uma perda, depois diminui a taxa de transmissão**
  - Continua a aumentar quando recebe um ACK, diminui quando ocorre uma perda



Q: Com que velocidade aumenta e diminui a taxa de transmissão?

- A resposta vem nos próximos diapositivos

TCP: controlo de congestionamento



- O transmissor limita a taxa de transmissão limitando o número de bytes não confirmados em trânsito (in pipeline):

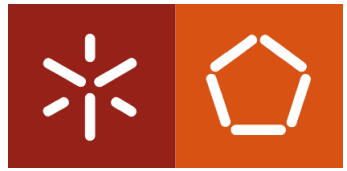
  `LastByteSent-LastByteAcked ≤ cwnd`

  - **cwnd:** difere da **rwnd** (como, porquê?)
  - O transmissor é limitado por **min(cwnd,rwnd)**
- De forma aproximada:

$$\text{Taxa de transmissão} = \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- A cwnd é dinâmica, varia em função da percepção que o TCP tem do estado da rede

TCP: controlo de congestionamento



Perda de um segmento por:

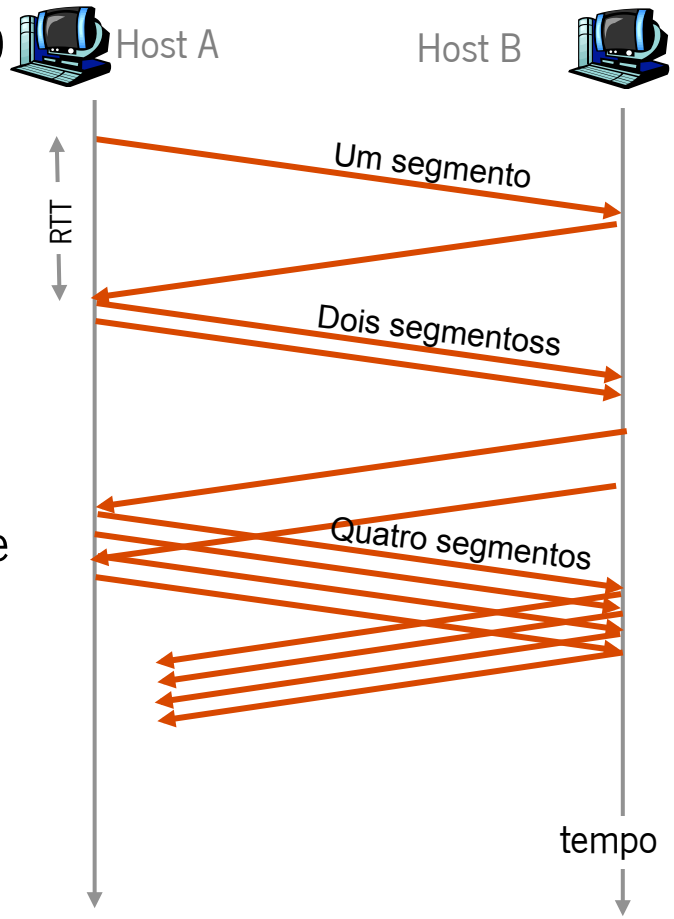
- timeout: o receptor não responde
  - Reduz-se a **cwnd** para 1
- 3 ACKs duplicados: pelo menos alguns dos segmentos enviados foram recebidos (algoritmo *fast retransmit / fast recovery*)
  - Quando tudo volta ao normal, a **cwnd** reduz para metade, é menos agressivo do que o timeout

ACK recebido: aumenta a **cwnd**

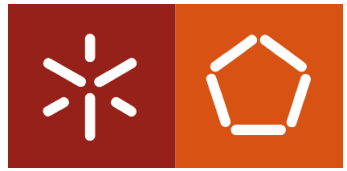
- Fase *slowstart*:
  - Aumenta exponencialmente, acontece quando iniciamos a conexão, ou quando ocorre um timeout
- Fase *congestion avoidance*:
  - Aumenta linearmente

TCP: *slow start*

- Quando uma conexão começa, cwnd = 1 MSS
  - exemplo: MSS = 500 bytes, e o RTT = 200 msec
  - Taxa de transmissão inicial = 20 kbps
- Largura de banda disponível pode ser >> MSS/RTT
  - É desejável atingir rapidamente a máxima taxa de transmissão possível
- A taxa de transmissão cresce exponencialmente até, ocorrer a primeira perda, ou quando o threshold é atingido
  - Duplica a **cwnd** em cada RTT
  - Fá-lo aumentando em 1 a **cwnd** por cada ACK recebido



TCP: controlo de congestionamento

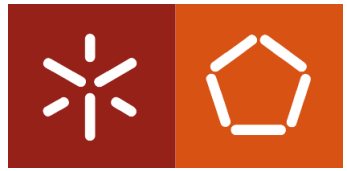


Transição do slow start para o congestion avoidance:

ssthresh: o threshold é mantido pelo TCP

- Quando ocorre uma perda por Timeout:
  - configura ssthresh para cwnd/2
- Quando cwnd >= ssthresh:
  - transita da fase slow start para a fase congestion avoidance

TCP: *congestion avoidance*



- Quando cwnd > ssthresh a cwnd cresce linearmente
  - Aumenta a **cwnd** em 1 MSS por cada RTT
  - Aproxima-se mais lentamente para um cenário de possível congestionamento do que o slowstart
  - implementação: **cwnd = cwnd +MSS*MSS/cwnd** para cada ACK recebido

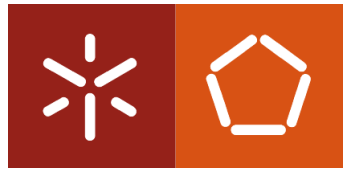
AIMD: Additive Increase Multiplicative Decrease

AIMD

- ACKs: aumenta a **cwnd** em 1 MSS por RTT: additive increase
- perda: reduz a **cwnd** para metade (quando não é detectada uma perda por timeout): multiplicative decrease

TCP/IP

Fast Recovery



- **Ao receber três ACKs duplicados, a entidade TCP retransmite o respetivo segmento (*Fast Retransmit*), diminui para metade o valor da janela de congestão e entra numa fase que se designa por *Fast Recovery*.**
  - A janela de congestão é incrementada por cada ACK duplicado recebido (começando logo pelos três ACKs duplicados).
  - Quando chega o ACK que confirma todos os dados, nomeadamente o segmento retransmitido, a janela volta ao valor inicial (que tinha quando entrou na fase de Fast Recovery), igual a metade do valor da janela de congestão antes do *Fast Retransmit.*

TCP: implementações populares

Reno e Tahoe: