Nível de Transporte

Baseado no Capítulo 3 do livro

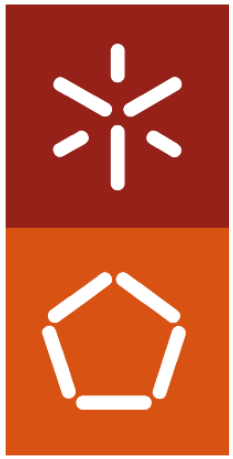
Computer Networking: A Top Down Approach - Chapter 3,

Jim Kurose, Keith Ross, Addison-Wesley ©2016

Comunicações por Computador

Licenciatura em Engenharia Informática

Universidade do Minho



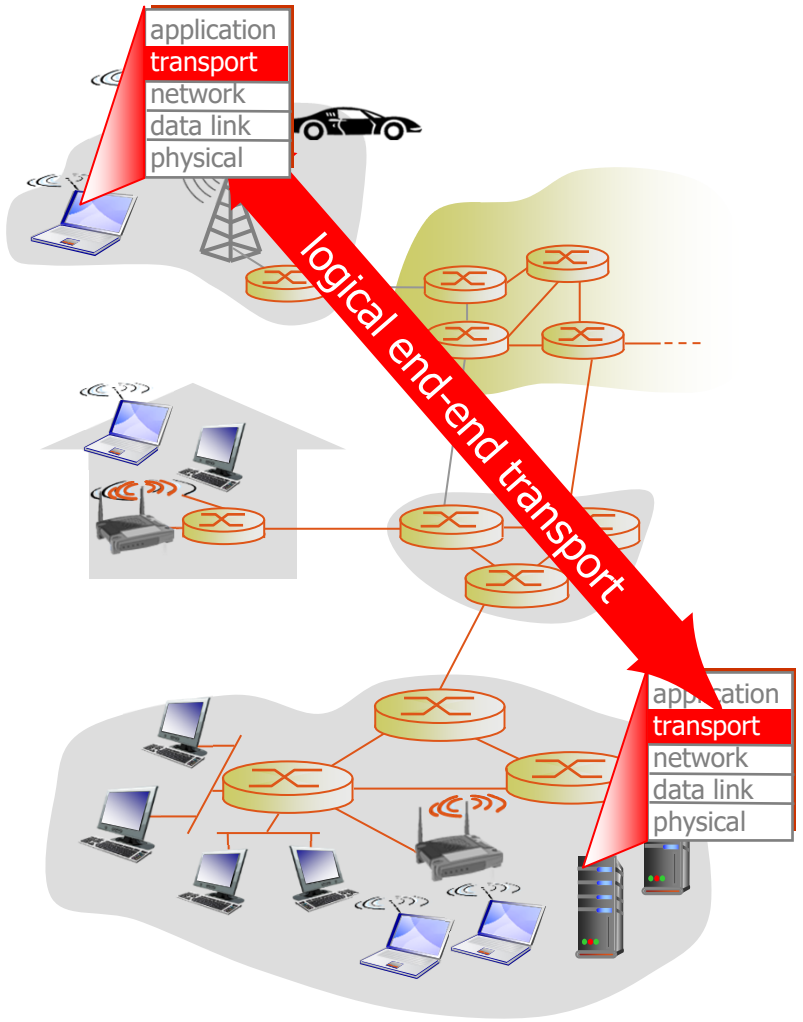
Nível de Transporte

Objetivos:

- **Compreender os princípios subjacentes aos serviços de camada de transporte:**
  - Transferência confiável de dados
  - Controlo de fluxo
  - Controlo de congestão
- **Conhecer os protocolos da camada de transporte da Internet**
  - UDP: transporte não orientado à conexão
  - TCP: transporte confiável e orientado à conexão
  - Controlo de congestão TCP

Serviço 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 recetor: junta os diferentes **segmentos** que constituem uma mensagem que passa à respetiva aplicação
  - Internet: TCP e UDP



Transporte *versus* Rede

Camada de Rede:

- fornece uma ligação lógica entre dois *sistemas terminais*

Camada de Transporte:

- fornece uma comunicação lógica entre *processos*

- Usa e melhora os serviços disponibilizados pela camada de Rede
- Troca de dados **fiável e ordenada** (TCP)
  - Controlo de **Fluxo**, Estabelecimento da Ligação
  - Controlo de **erros**
  - Controlo de **congestão**
- Troca de dados **não fiável e desordenada** (UDP)
- Serviços não disponíveis: garantia de atraso máximo e largura de banda mínima

Transporte

- É mesmo necessário termos uma camada de transporte?
- Tudo o que a camada de transporte faz não pode ser feito pelas aplicações?
- Não é possível desenvolver aplicações diretamente sobre o protocolo de rede IP?
- Não é possível desenvolver aplicações diretamente sobre a camada lógica (MAC)?

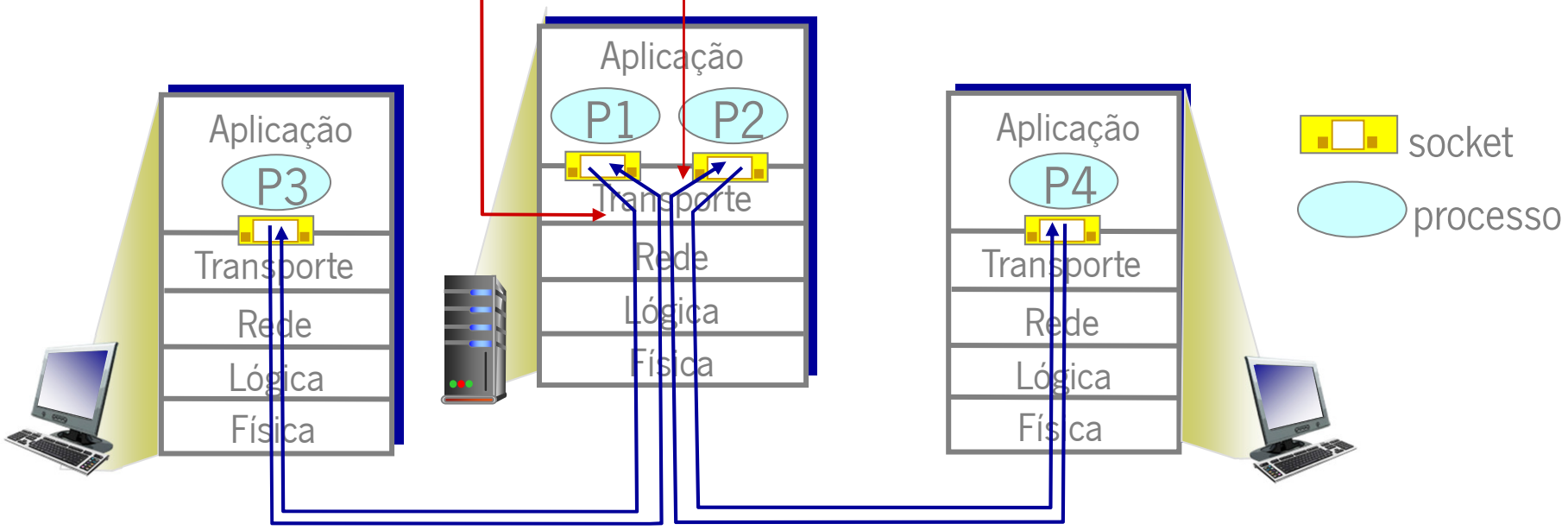
Multiplexagem / Desmultiplexagem

Multiplexagem no emissor:

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

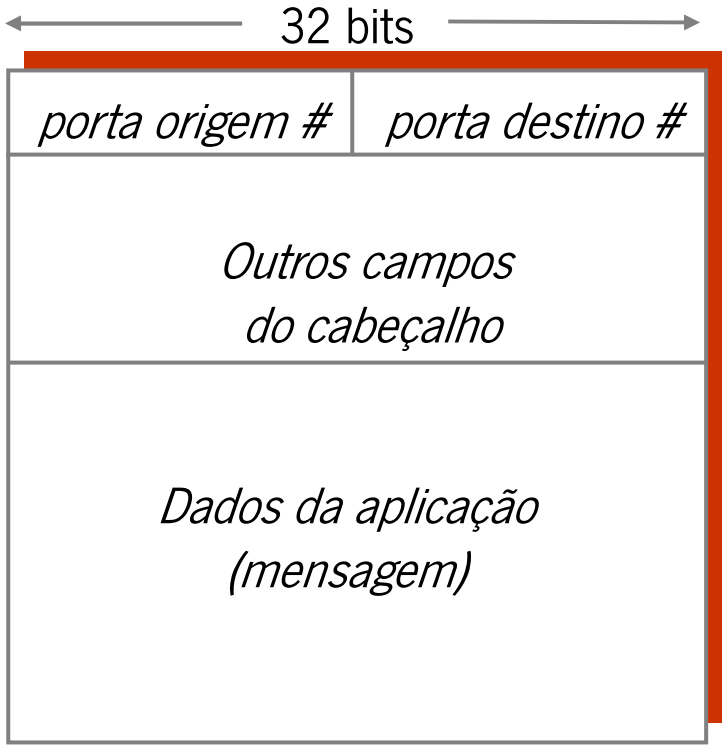
Desmultiplexagem no recetor:

Entregar os diferentes segmentos ao *socket* correcto.



Desmultiplexagem

- **É efetuada 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* correto



Formato dos segmentos TCP/UDP

Desmultiplexagem
– *não orientado à conexão*

- **As aplicações criam um *socket* … e limitam-se a enviar datagramas para IP Destino, Porta destino**

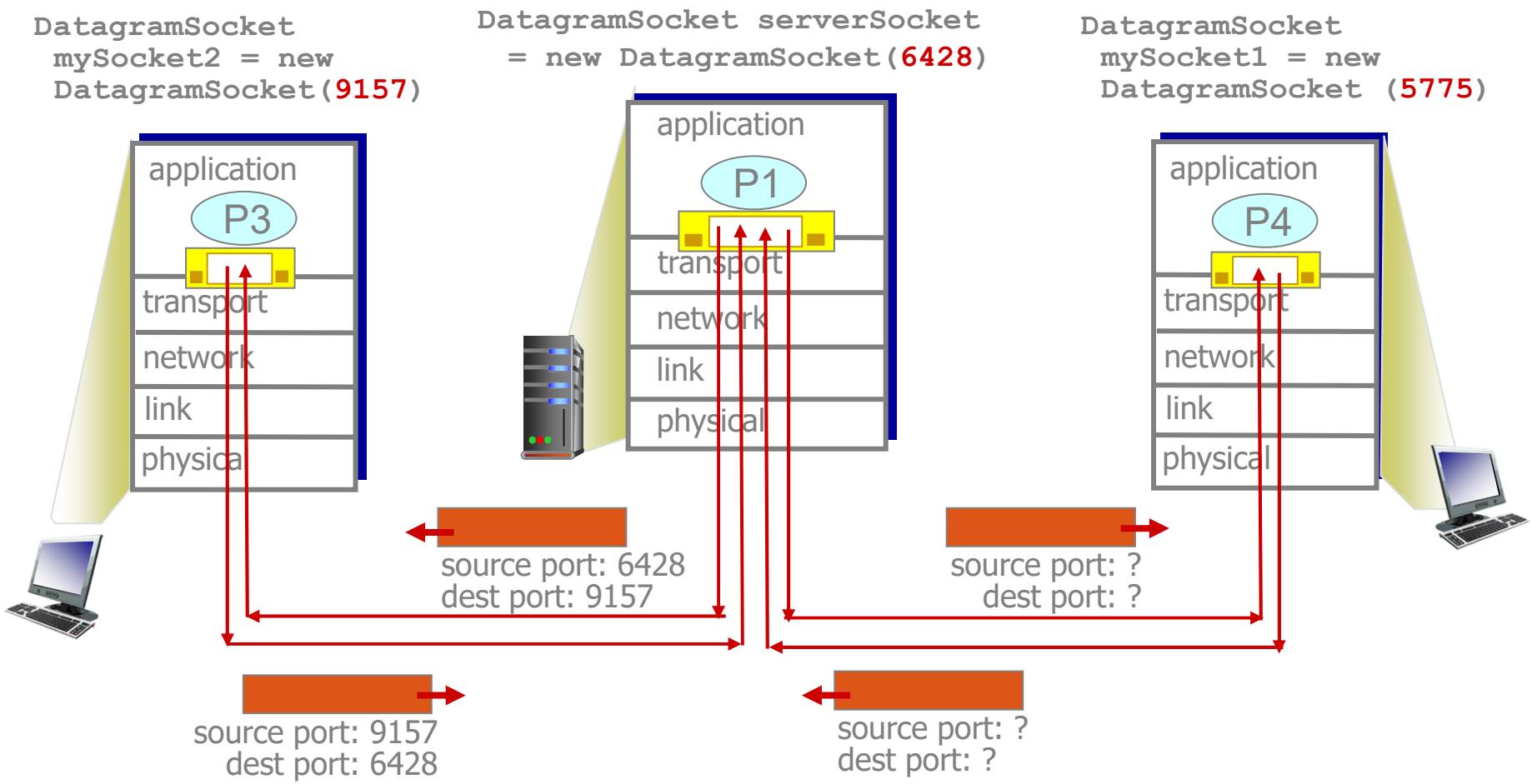
```
DatagramSocket s= new DatagramSocket();
DatagramPacket p = new DatagramPacket(aEnviar, aEnviar.length, IPAddress, 9999);
 s.send(p);
```

- **Quando o *host* recebe um segmento UDP:**
  - verifica a ***porta#*** destino do segmento
  - direciona o segmento UDP para o *socket* com essa ***porta#***

Datagramas IP com o mesmo *IP Destino, Porta destino,* mas com diferentes IP de origem e/ou portas de origem são dirigidos ao ***mesmo socket*** no destino!

Desmultiplexagem

– *não orientado à conexão*



Desmultiplexagem
– *orientado à conexão*

- **As aplicações criam um *socket* e uma conexão com servidor destino para enviar dados**

Opcionais!

```
Socket socketCliente = new Socket(IPDestino, portaDestino, IPLocal, portaLocal);
```

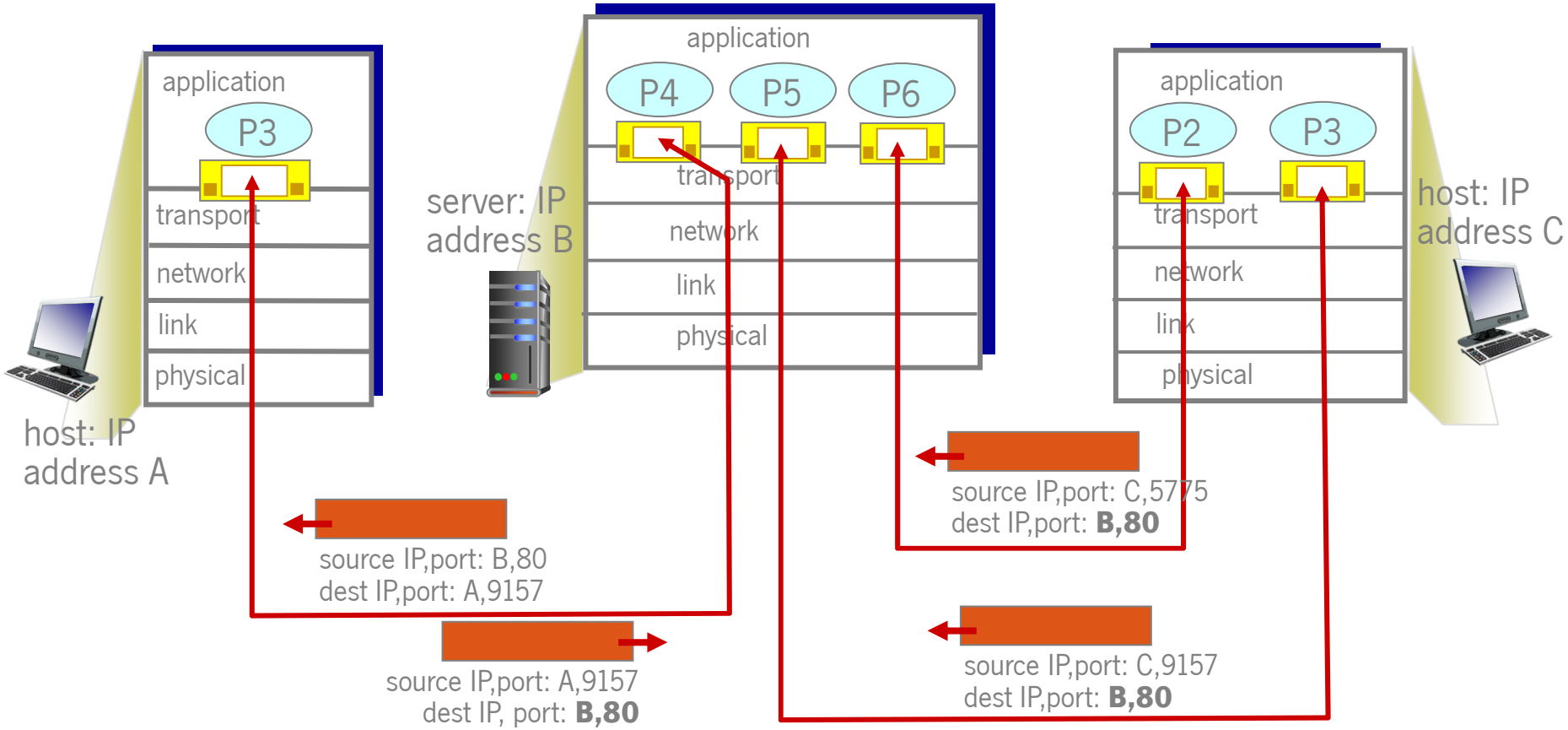
- ***Socket* TCP identifica-se com 4 valores:**
  - endereço IP origem
  - nº porta origem
  - endereço IP destino
  - nº porta destino

Recetor usa sempre **os 4 valores** para redirecionar para o *socket* correto!

→ Servidor pode ter várias conexões TCP distintas em simultâneo, com um ***socket* distinto** para cada uma delas!

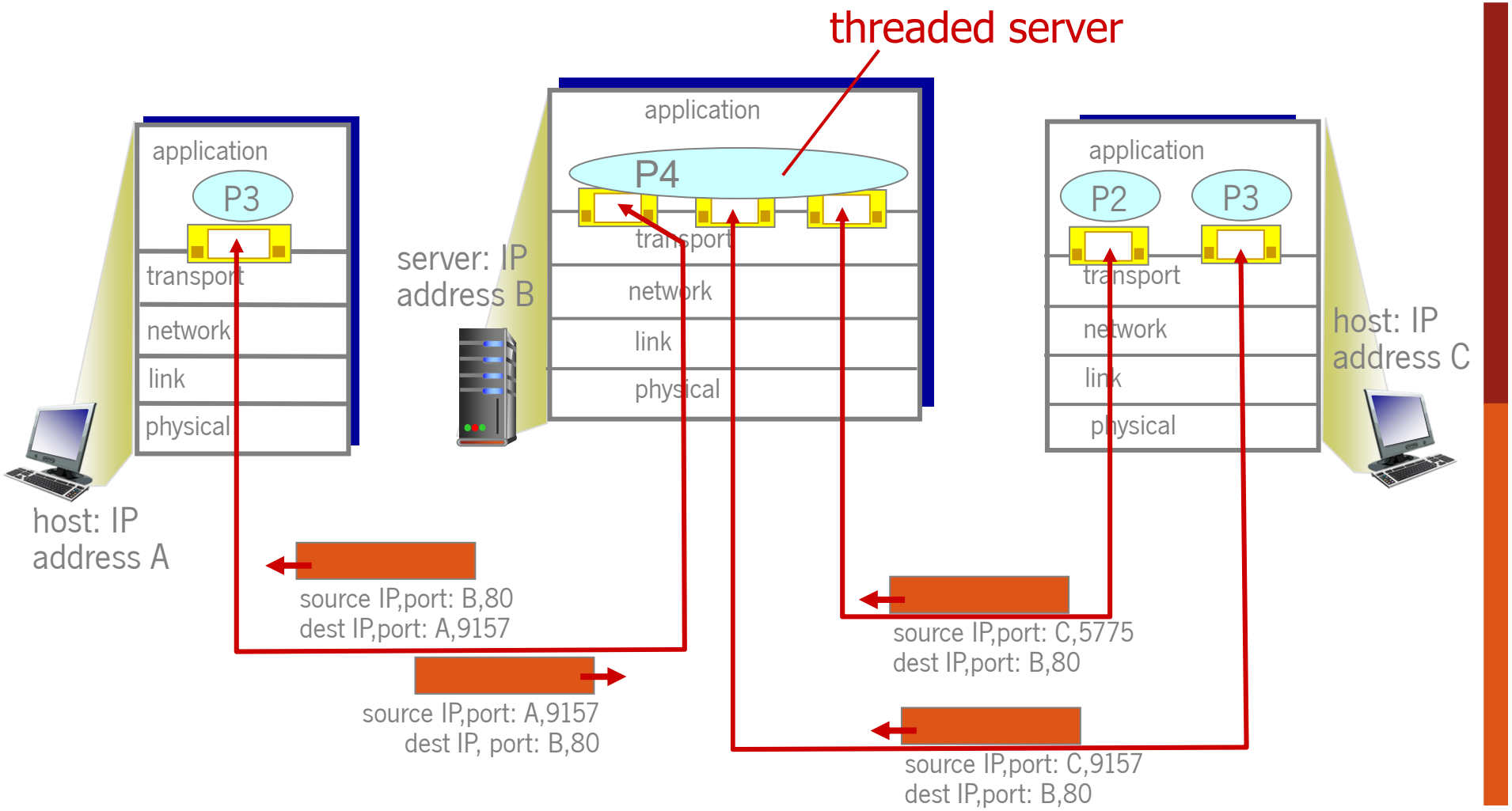
Desmultiplexagem
– *orientado à conexão*

Três segmentos, todos destinados ao endereço destino (IP B, porta 80) são desmultiplexados em *sockets* diferentes!



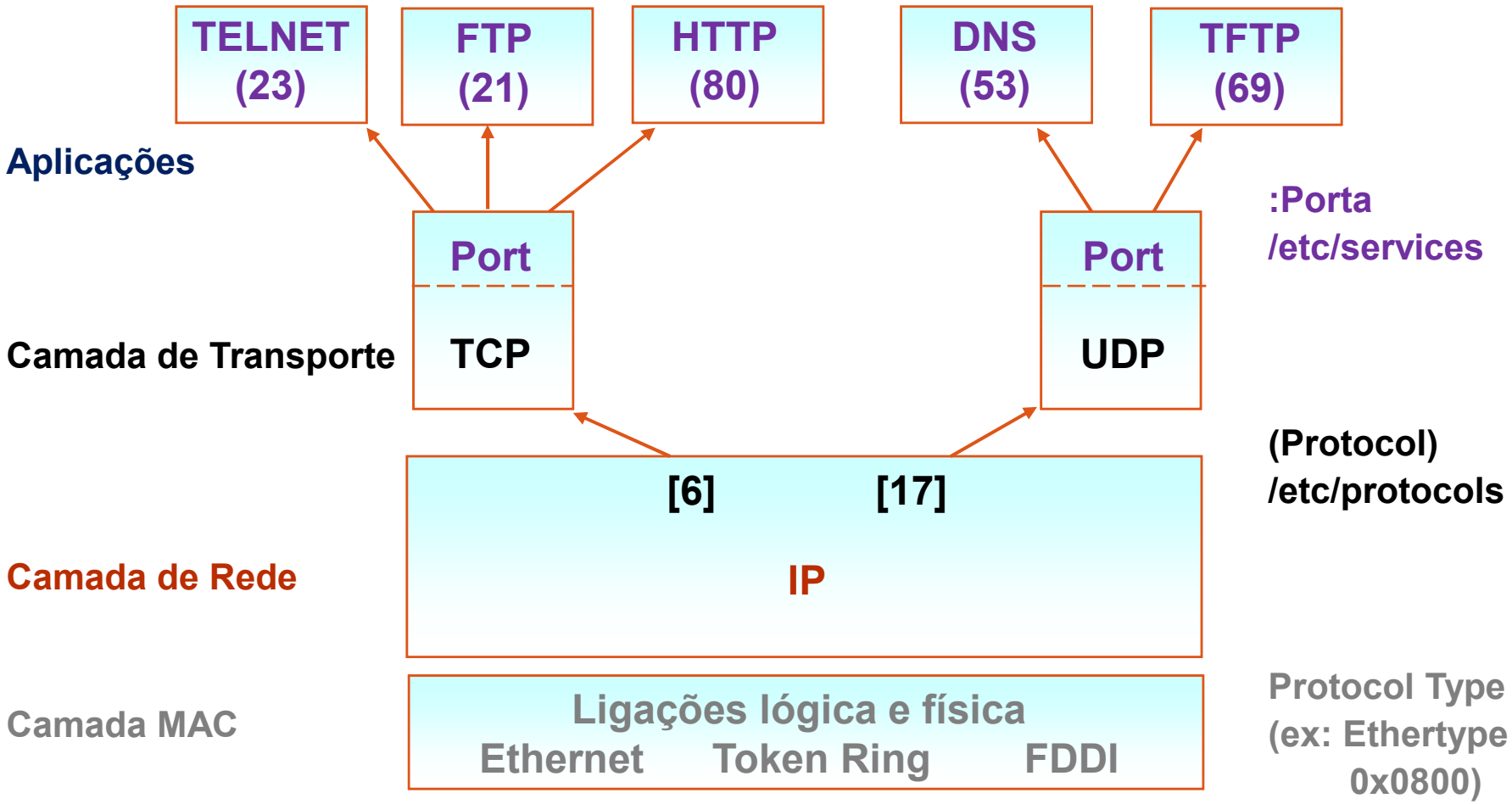
Desmultiplexagem

– *orientado à conexão*



TCP/IP

Protocolos de Transporte: UDP e TCP



TCP/IP

User Datagram Protocol (UDP)

Funções do UDP:

- protocolo de transporte fim-a-fim, não fiável
- orientado ao datagrama (sem conexão)
- atua como uma interface da aplicação com o IP para multiplexar e desmultiplexar tráfego
- usa o conceito de porta / número de porta
  - forma de direcionar datagramas IP para o nível superior
  - portas reservadas: 0 a 1023, dinâmicas: 1024 a 65535
- é utilizado em situações que não justificam o TCP
  - exemplos: TFTP, RPC, DNS
- ... ou quando as aplicações querem controlar o fluxo de dados e gerir erros de transmissão diretamente

UDP

Desmultiplexagem

- O *socket* UDP é identificado através de dois valores: endereço IP destino e número de porta destino
- Quando um sistema terminal recebe um segmento UDP verifica qual o número da porta destino que consta do segmento UDP e redireciona o segmento para o *socket* com esse número de porta
- Datagramas com diferentes endereços IP origem e/ou portas origem podem ser redirecionados para o mesmo *socket*

UDP

Sockets

Criar o socket:

```
DatagramSocket s= new DatagramSocket(9876);
```

Fica em estado de escuta e pronto a receber datagramas

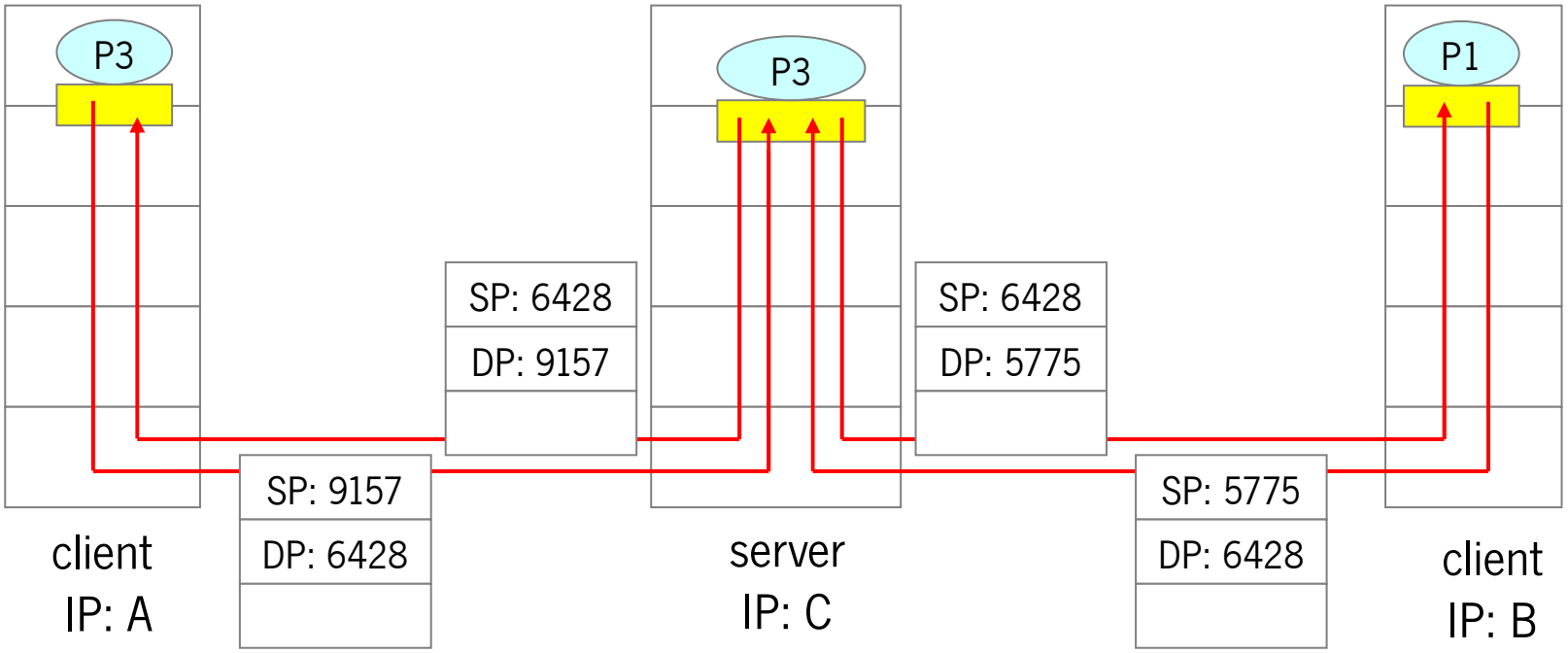
```
$ netstat –n –a
Proto  Local Address          Foreign Address        State
UDP    0.0.0.0:9876           *:*
```

E está pronto a receber dados:

```
byte[] aReceber = new byte[1024];
DatagramPacket pedido =  new DatagramPacket(aReceber, aReceber.length);
s.receive(pedido);
```

UDP

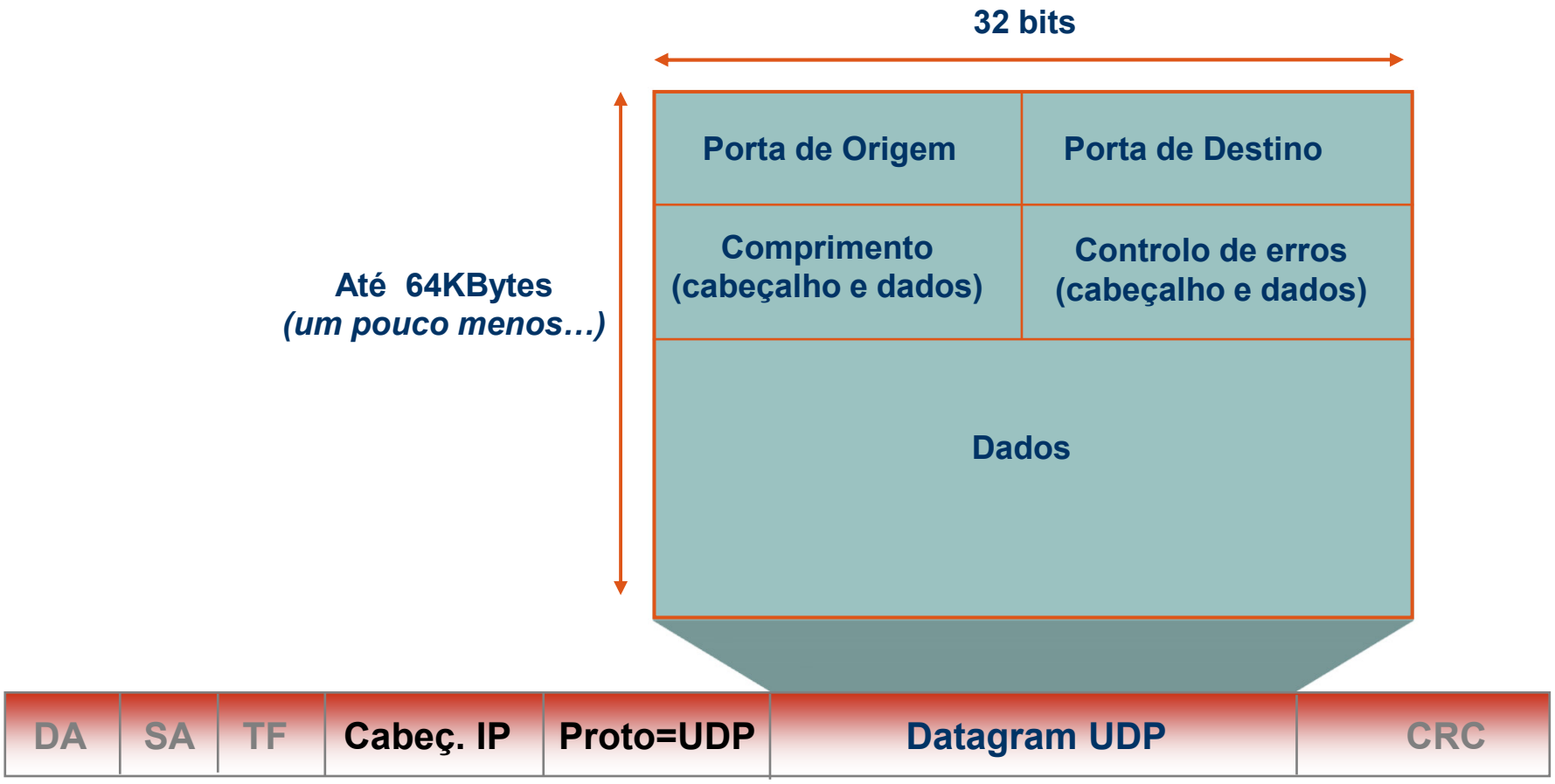
Desmultiplexagem



O lado SP fornece o "return address"...

UDP

Protocol Data Unit (PDU) – Datagram



UDP

Controlo de Erros

Checksum:

- complemento para 1 da soma de grupos de 16 bits
- cobre o datagrama completo (cabeçalho e dados)
- o cálculo é facultativo mas a verificação é obrigatória
- **Checksum = 0** significa que o cálculo não foi efetuado
- **Checksum ≠ 0** significa que o recetor deteta erro na soma e:
  - o datagrama é ignorado (descartado)
  - não é gerada mensagem de erro para enviar ao transmissor
  - a aplicação de receção é notificada

UDP

Discussão...

O que pode levar um programador a escolher o UDP como suporte à comunicação na sua aplicação sabendo que, à partida, o UDP não dá garantias nenhumas de entrega dos dados, fornece apenas um serviço mínimo de multiplexagem/desmultiplexagem de dados aplicacionais e possui um mecanismo de verificação de erros que é opcional? Isto tendo em consideração que existem alternativas que fornecem muito mais garantias!

UDP

Discussão...

O que leva uma aplicação a escolher o UDP?

- Maior controlo sobre o envio dos dados por parte da aplicação (muitas vezes serve como fuga ao controlo de congestão do TCP)
  - aplicação controla quando deve enviar ou reenviar os dados sem deixar essa decisão ao serviço de transporte
  - aplicação decide quantos bytes envia realmente de cada vez
- Não há estabelecimento e terminação da conexão, diminuindo o atraso inicial na comunicação
- Não é necessário manter informação de estado da conexão
- Menor *overhead* de dados (cabeçalho UDP são apenas 8 bytes)

TCP

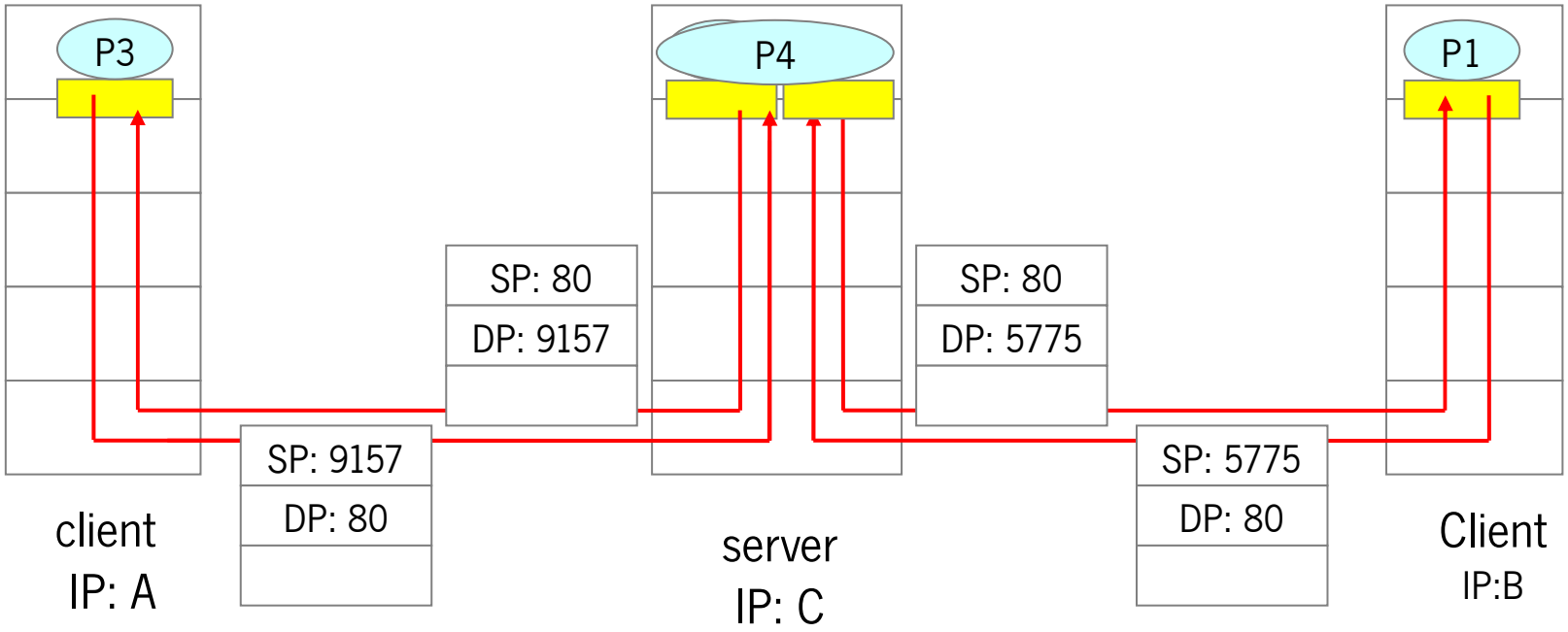
Transmission Control Protocol

Funções do TCP

- permite transporte fiável de dados aplicacionais fim-a-fim
- efetua associações lógicas fim-a-fim (conexões)
- cada conexão é identificada por um *socket* TCP e cada conexão é um circuito virtual entre portas de aplicações (também designadas por portas de serviço)
- multiplexa os dados de várias aplicações através de número de porta
- efetua controlo de erros, de fluxo e de congestão

TCP

Desmultiplexagem



TCP

Desmultiplexagem

- Um *socket* TCP é identificado por quatro valores:
  endereço IP e número de porta de origem +
  endereço IP e número de porta de destino
- O sistema terminal (destino) ao receber um pacote IP com um segmento TCP usa esses 4 valores para redirecionar o segmento para o *socket* respetivo
- Um servidor pode suportar vários *sockets* TCP simultaneamente
- Os servidores Web utilizam um *socket* diferente para cada cliente ou até um *socket* diferente por cada pedido (http não-persistente)

TCP

Sockets

Criar o socket de atendimento principal:

```
ServerSocket welcomeSocket = new ServerSocket(9876);
```

```
$ netstat –n –a
Proto  Local Address        Foreign Address      State
TCP    0.0.0.0:9876         0.0.0.0:0            LISTENING
```

E lidar com cada conexão num *socket* específico:

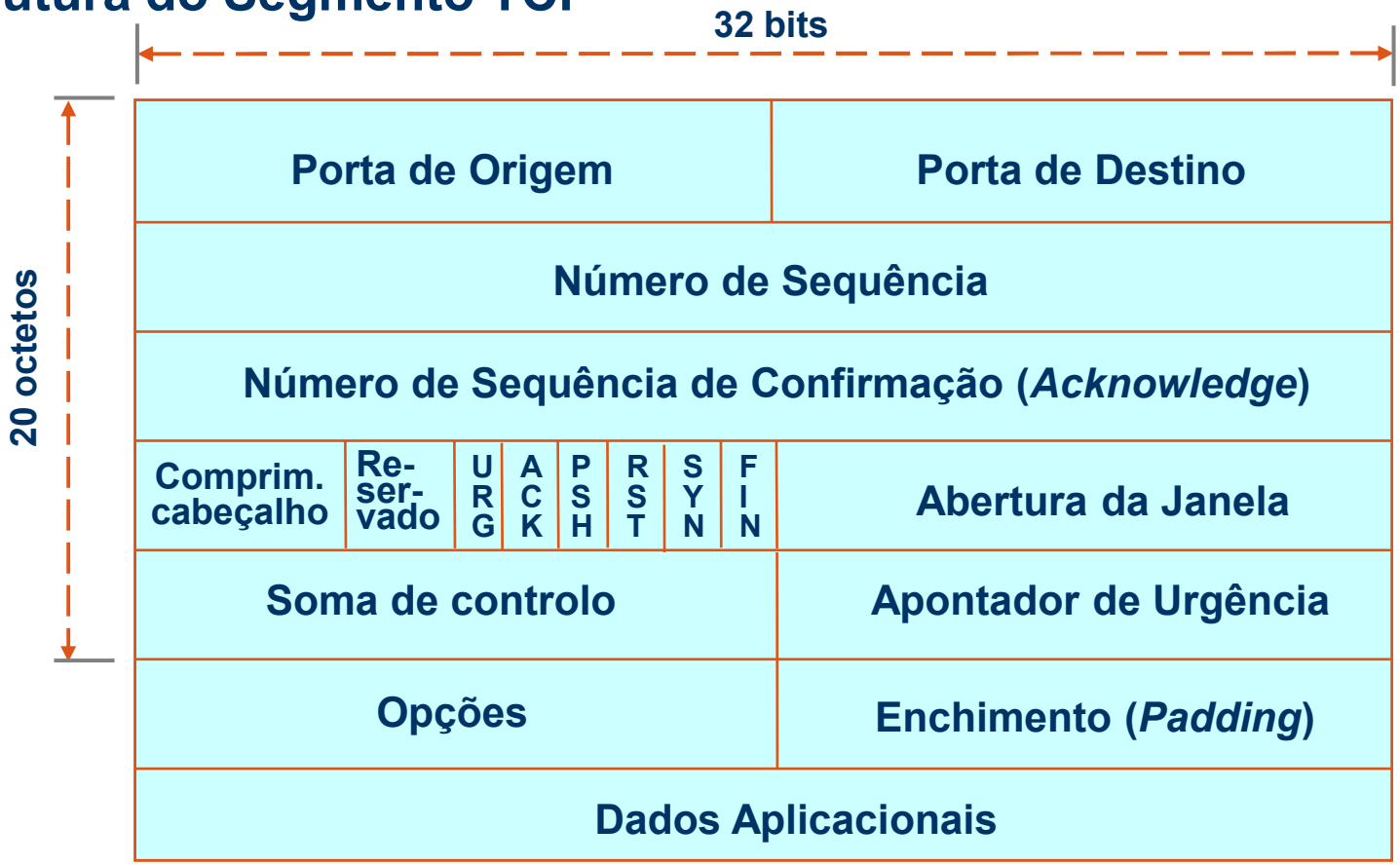
```
Socket connectionSocket = welcomeSocket.accept();
```

```
$ netstat –n –a
Proto  Local Address        Foreign Address      State
TCP    127.0.0.1:9876       127.0.0.1:5459       ESTABLISHED
```

TCP

Protocol Data Unit (PDU) – Segmento

Estrutura do Segmento TCP



TCP

Segmento

- Porta Origem/Destino – Nº da porta TCP da aplicação/processo de Origem/Destino
- Número de Sequência – ordem do primeiro octeto de dados no segmento
  (se SYN = 1, este número é o *initial sequence number*, ISN)
- Número de Ack (32 bits) – o número de ordem do primeiro octeto que a entidade TCP espera receber a seguir
- Comprimento do Cabeçalho (4 bits) – número de palavras de 32 bits no cabeçalho
- Flags (6 bits) – indicações específicas por cada bit...
- Janela – número de octetos que o recetor é capaz de receber sem processar
  (controlo de fluxo)
- Soma de controlo (16 bits) – valor para deteção de erros
- Apontador de Urgência (16 bits) – adicionado ao número de sequência atual dá o número de sequência do último octeto de dados urgentes
- Opções (variável) – especifica características opcionais (ex. MSS, *timestamp*, fator de escala para a janela, etc.)

TCP

Segmento – Flags

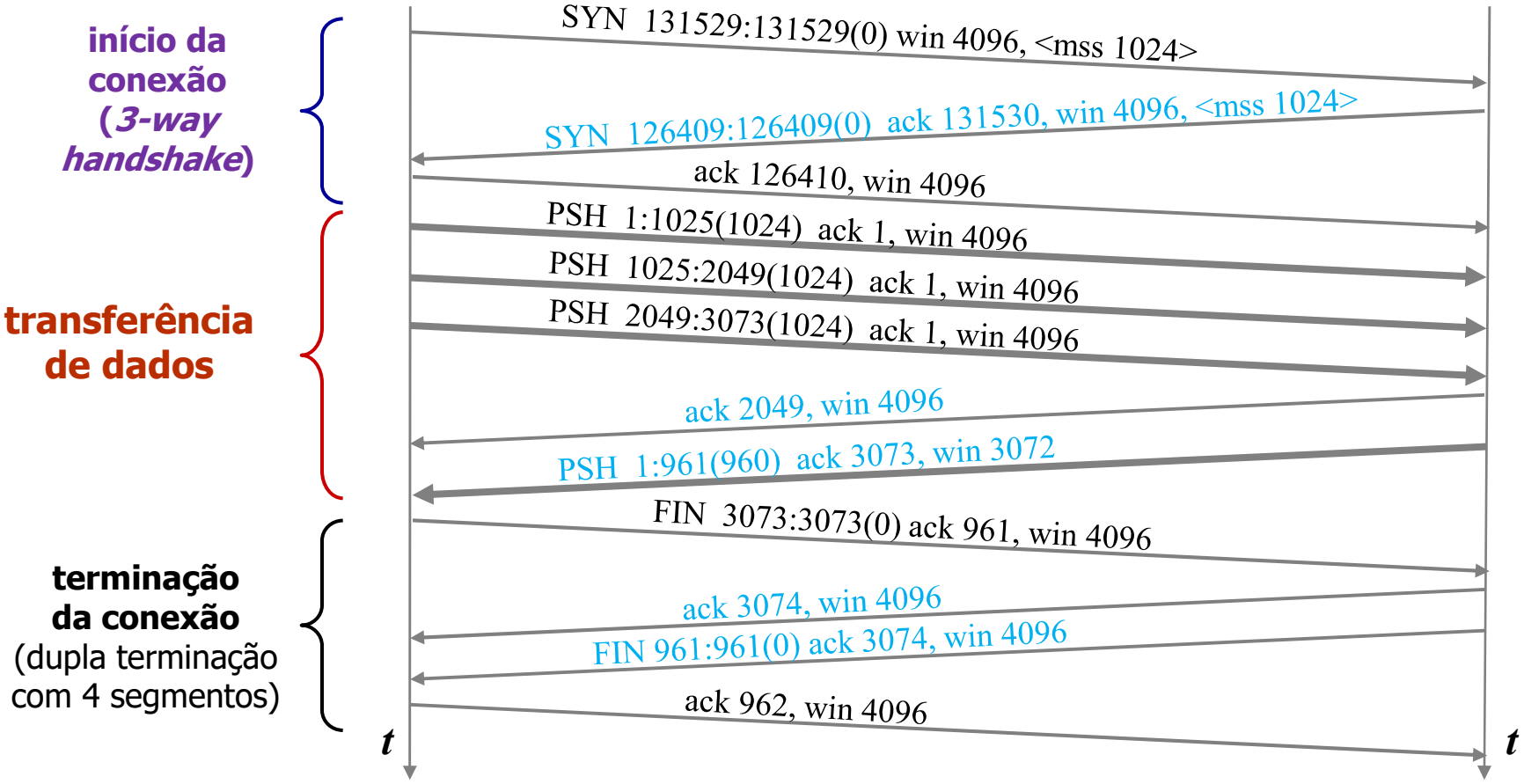
Flags TCP (1 bit por *flag*):

- URG – indica se o apontador de urgência é válido
- ACK – indica se o nº de sequência de confirmação é válido
- PSH – o recetor deve passar imediatamente os dados à aplicação (aparece nos segmentos de transferência de dados)
- RST – indica que a conexão TCP vai ser reinicializada
- SYN – indica que os números de sequência devem ser sincronizados para se iniciar uma conexão
- FIN - indica que o transmissor terminou o envio de dados

Os segmentos SYN e FIN consomem um número de sequência

TCP

Operação



TCP

Discussão...

- Que acontece se o servidor não responde ao SYN inicial?
- Que acontece se o servidor não enviar um FIN de volta?
  Ou o cliente não enviar o ACK final?
- Quais as questões de segurança associadas ao início e fim de conexão?
- O que acontece se, em vez de *3-way,* se usasse *2-way handshake*?

TCP

Estabelecimento da Ligação

O emissor e o recetor TCP estabelecem uma ligação antes de iniciarem a troca de dados, gerando várias interações protocolares entre cliente e servidor:

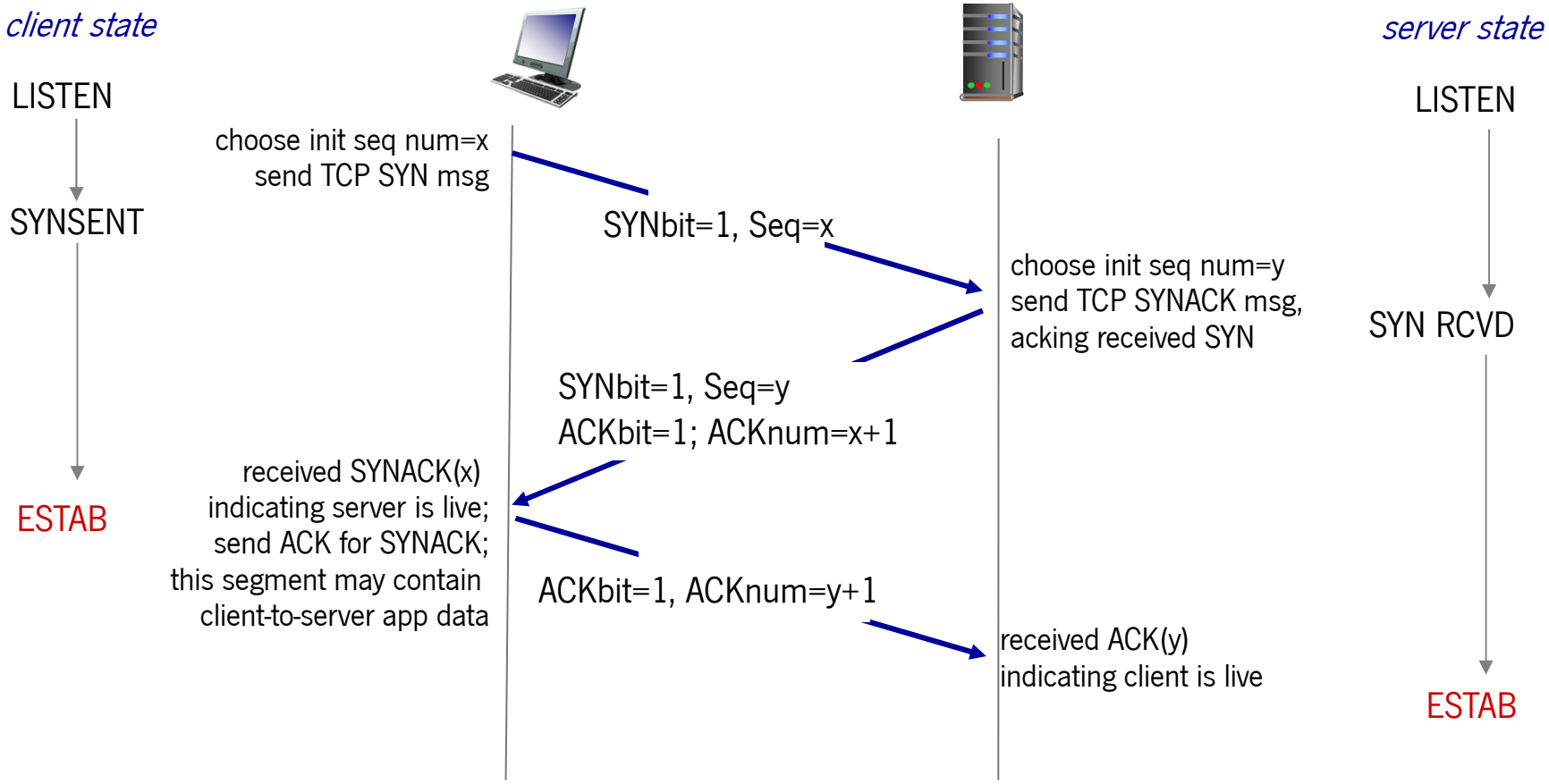
- Inicialização de variáveis...
  - números de sequência
  - *buffers*, controlo de fluxo (e.g. `RcvWindow`)
- Cliente inicia o pedido de ligação...
  ```
  Socket clientSocket = new
  Socket("hostname","port")
  ```
- Servidor aceita o pedido de ligação...
  ```
  Socket connectionSocket =
  welcomeSocket.accept()
  ```

Interações respetivas:

1. O cliente envia segmento SYN para o servidor e especifica o número de sequência inicial para aquele sentido da comunicação, sem enviar dados aplicacionais
2. O servidor recebe o SYN e responde com um segmento SYN ACK e aloca espaço de armazenamento para os dados aplicacionais e especifica o número de sequência inicial para aquele sentido da comunicação
3. O cliente recebe o segmento SYN ACK e responde com um segmento ACK que pode conter dados

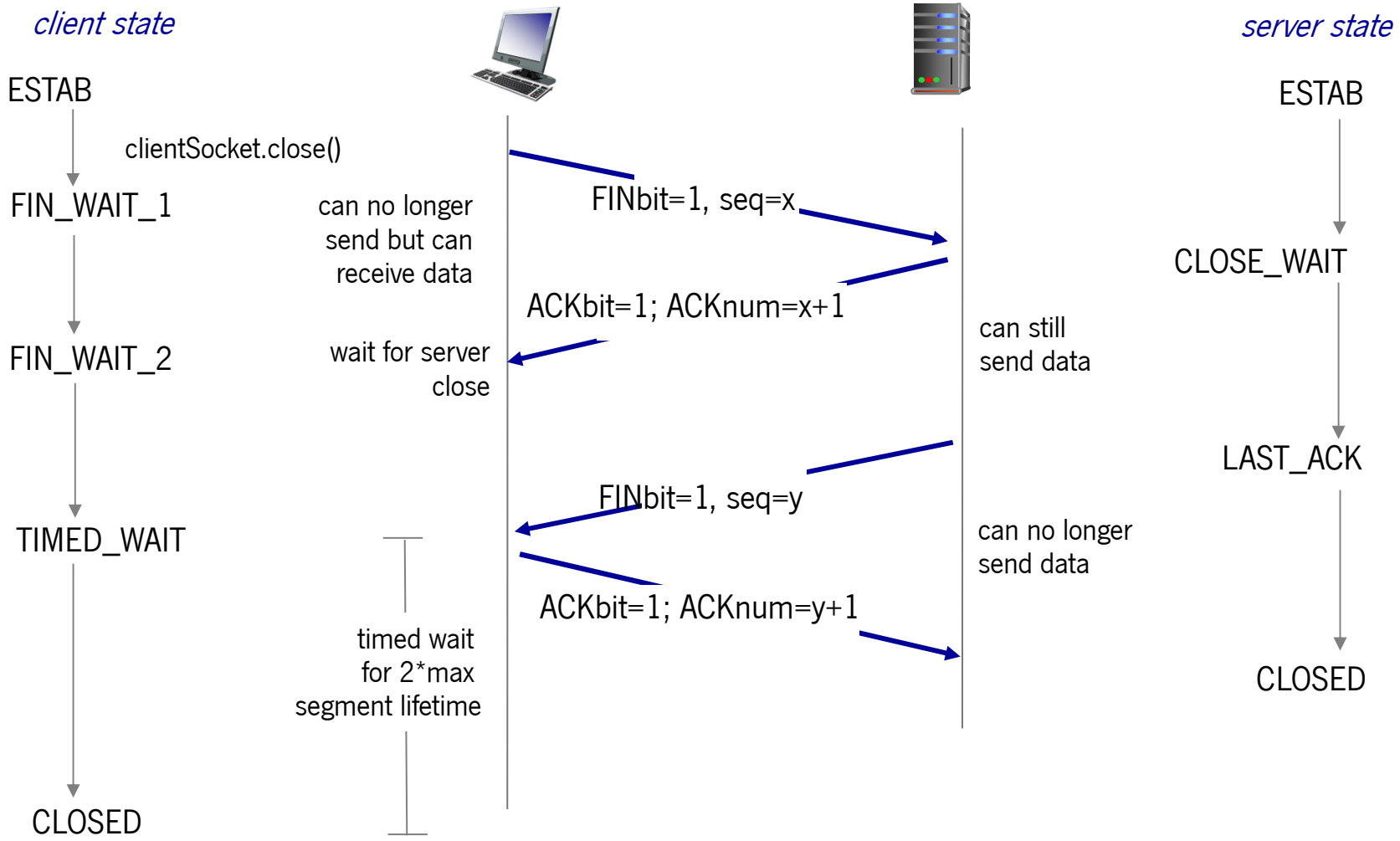
TCP

Estabelecimento da Ligação



TCP

Finalizar da Ligação



TCP

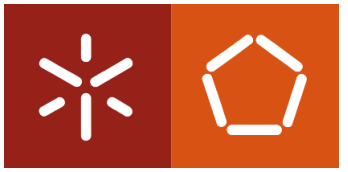
Maximum Segment Size – MSS

Significado do MSS do TCP

- opção TCP que apenas aparece em segmentos SYN
- o MSS define o maior bloco de dados aplicacionais que a entidade enviará na conexão num único segmento
- ao iniciar-se uma conexão, cada lado tem a opção de anunciar ao outro o MSS que pode receber
- para minimizar a fragmentação IP, não se deve definir um MSS maior que o MTU da tecnologia de ligação mais o tamanho dos cabeçalhos TCP e IP (por exemplo, sobre *Ethernet,* o MSS não deve ultrapassar 1460 bytes)

TCP

Estabelecimento da Ligação



```
> Frame 1516: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface en6, id 0
> Ethernet II, Src: Tp-LinkT_dd:71:32 (c4:e9:84:dd:71:32), Dst: Xensourc_83:7b:15 (00:16:3e:83:7b:15)
> Internet Protocol Version 4, Src: 193.136.9.175, Dst: 193.136.9.241
v Transmission Control Protocol, Src Port: 56843, Dst Port: 8080, Seq: 0, Len: 0
    Source Port: 56843
    Destination Port: 8080
    [Stream index: 54]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 1125210519
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1011 .... = Header Length: 44 bytes (11)
  > Flags: 0x0c2 (SYN, ECN, CWR)
    Window: 65535
    [Calculated window size: 65535]
    Checksum: 0xfdbb [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  v Options: (24 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), Timestamps, SACK permitted…
    > TCP Option - Maximum segment size: 1460 bytes
    > TCP Option - No-Operation (NOP)
    > TCP Option - Window scale: 6 (multiply by 64)
    > TCP Option - No-Operation (NOP)
    > TCP Option - No-Operation (NOP)
    > TCP Option - Timestamps: TSval 314438861, TSecr 0
    > TCP Option - SACK permitted
    > TCP Option - End of Option List (EOL)
  v [Timestamps]
      [Time since first frame in this TCP stream: 0.000000000 seconds]
      [Time since previous frame in this TCP stream: 0.000000000 seconds]
```

Segmento sem dados (LEN=0)
Mas conta um byte na stream que precisa ser confirmado com ACK

Inicio de conexão (SYN)

Opções negociadas no início da conexão: MSS e Window Scale

TCP

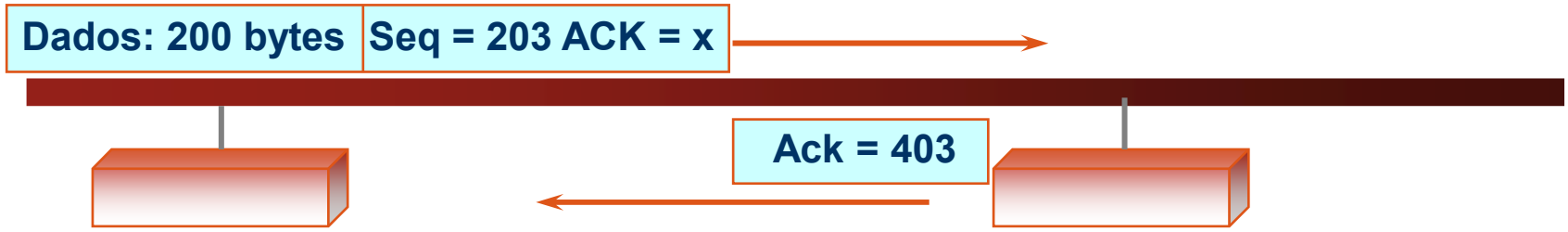
Operação

Segmentos TCP

- a sua sequenciação/numeração é necessária para permitir uma ordenação/re-ordenação dos dados à chegada e para permitir a sua referenciação nos ACK
- o *número de sequência* é baseado no número de bytes do campo de dados
- cada segmento TCP tem de ser confirmado (ACK), contudo é válido o ACK de múltiplos segmentos duma só vez
- o campo ACK indica o próximo byte (*sequence*) que o receptor espera receber (mecanismo *piggyback*)
- o emissor pode retransmitir após *timeout:* o protocolo define o tempo máximo de vida dos segmentos ou *Maximum Segment Lifetime* (MSL)

TCP

Operação



- Cada sistema-final *(end-system)* mantém o seu próprio número de sequência [0, 2^{32} –1]
- Nº de ACK = Número de sequência + bytes corretos lidos no segmento.

TCP

Discussão...

“A” pretende enviar de forma fiável uma mensagem “m” para “B” usando uma ligação de “rede” não fiável...

- Como posso ter a certeza que B recebeu a mensagem “m”?
- O que pode correr mal no envio de “m” (tendo em atenção que estamos na camada de transporte)?
- Como lidar com possíveis erros?

TCP

Controlo de Erros

Nas redes de dados, o mecanismo preferencial para controlo de erros é o mecanismo de *Automatic Repeat reQuest* (ARQ).

O TCP implementa algumas estratégias simples do ARQ e que permite:

- A deteção de erros
- *O feedback* do recetor
- A retransmissão de PDUs com erros ou de PDUs não recebidos

Por estratégia, no TCP não há notificações negativas!

Só há notificações/confirmações positivas através de ACKs. Por esse motivo, o emissor pode apenas *desconfiar* que um determinado segmento enviado não chegou ao destino e repete a emissão.

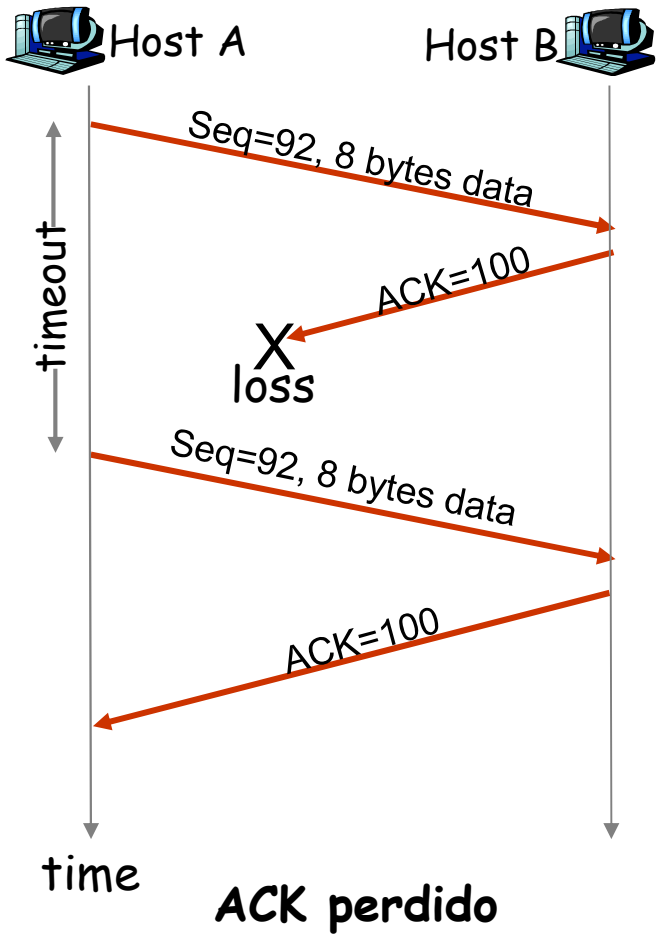
TCP

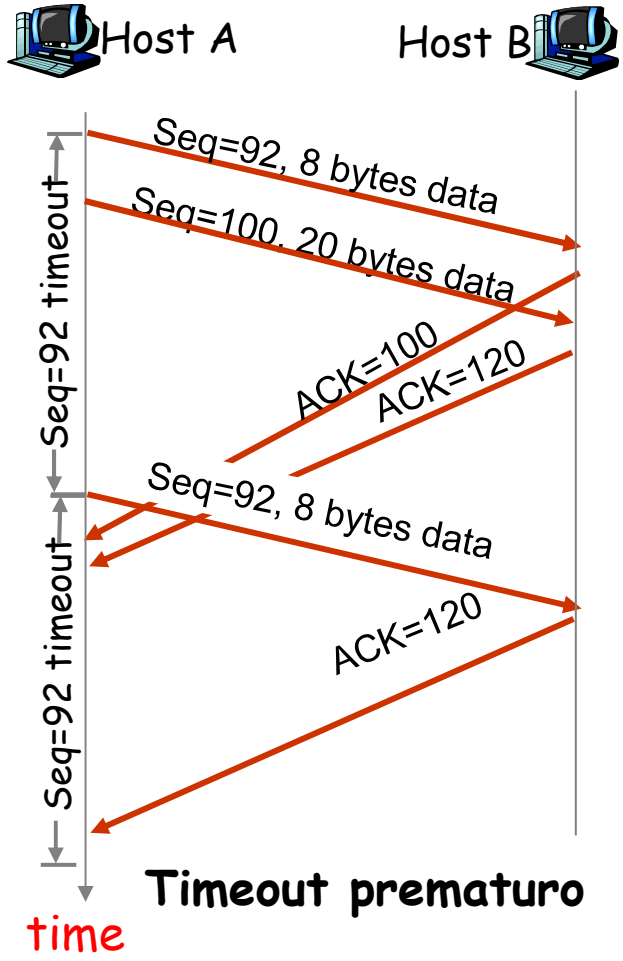
Discussão...

- Que deve fazer o receptor quando recebe um segmento com erros?
- Como pode o emissor saber que o segmento enviado estava com erros?
- E se o segmento se perder mesmo? O emissor pode ter a certeza que o segmento não chegou ao destino se não receber o ACK correspondente?

TCP

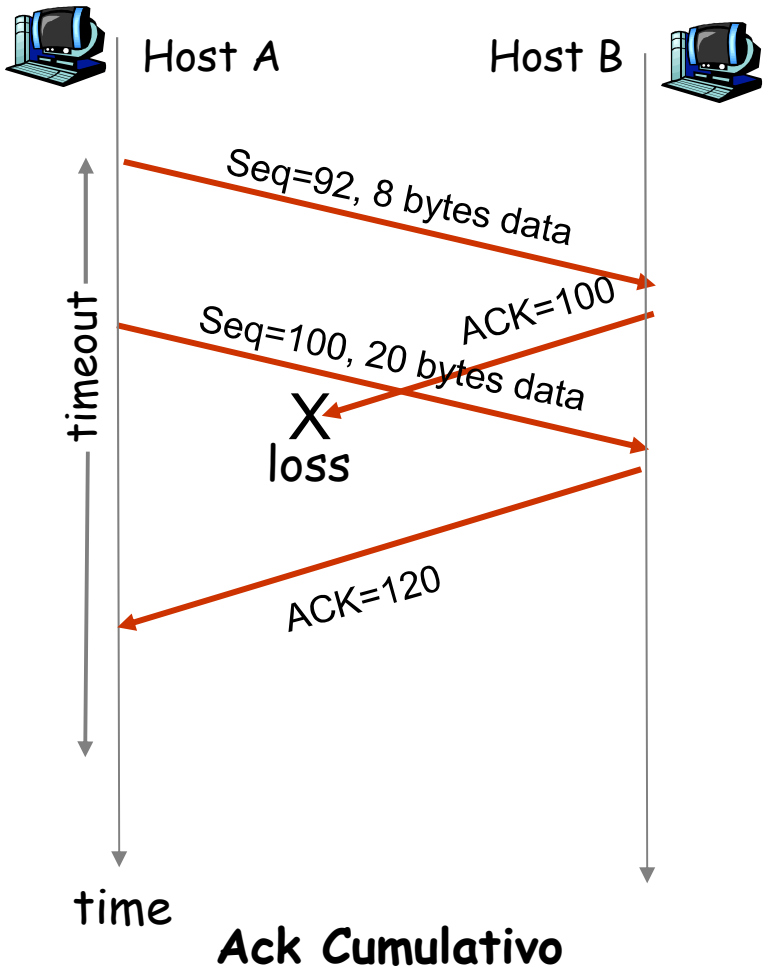
Retransmissões





TCP

Retransmissões



TCP

Timeout & Round Trip Time (RTT)

O valor do *Timeout* no TCP é dependente do valor do RTT:

- Demasiado curto aumenta o número de retransmissões desnecessárias e demasiado longo atrasa a reação a um segmento perdido...
- É necessário estimar bem o valor médio do RTT!
- Quanto maior for diferença entre os valores de RTT medidos e o valor médio estimado maior deverá ser o valor definido para o *Timeout*.

$$EstimatedRTT = (1-\alpha)*EstimatedRTT + \alpha*SampleRTT$$

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT - EstimatedRTT|$$

$$Timeout = EstimatedRTT + 4*DevRTT$$

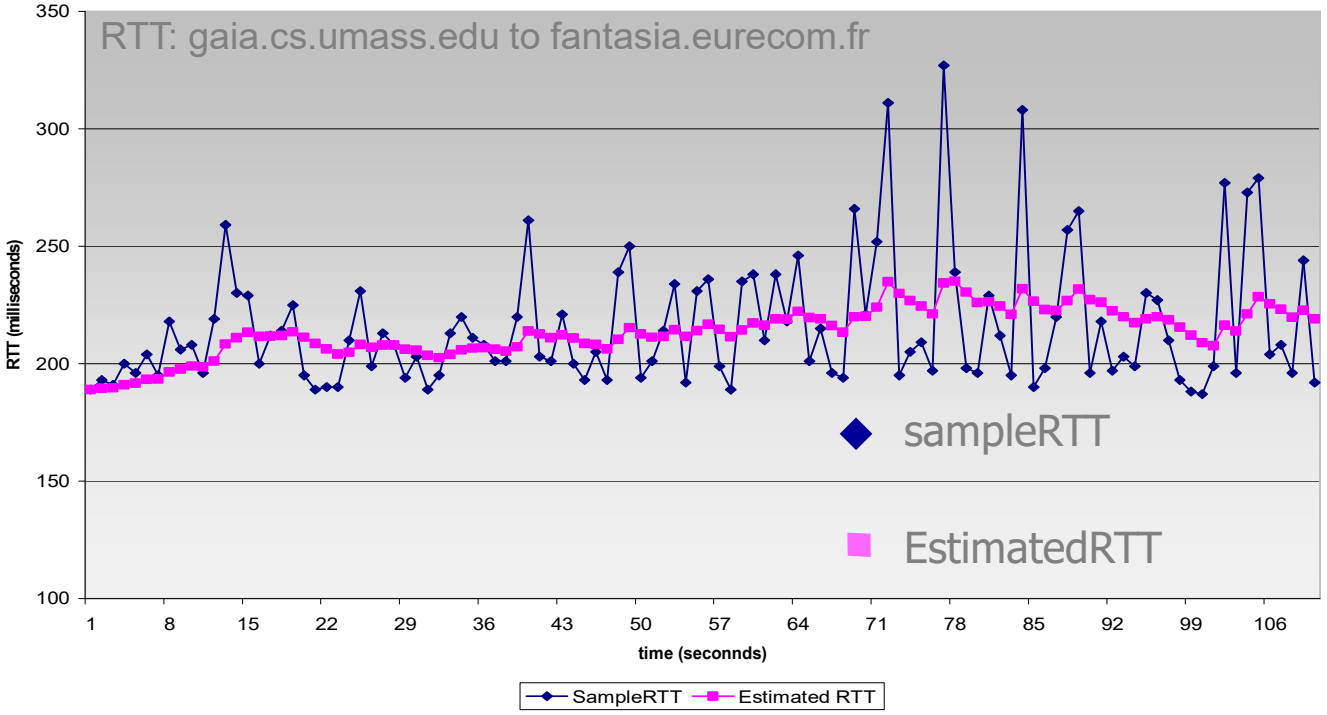
(typically, $\beta = 0.25$ & $\alpha = 0.125$)

TCP

RTT Estimation

`EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT`

- ✓ média móvel de peso exponencial, i.e, o peso do passado decresce exponencialmente...
- ✓ valor típico de $\alpha = 0.125$
- ✓ O SampleRTT é o tempo medido desde a transmissão do segmento até à recepção do ACK respetivo



TCP

Gestão de ACKs [RFC 1122, RFC 2581]

Evento no Recetor	Ação da outra 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… Se não 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 detetado	Repete imediatamente um ACK indicando o número de sequência esperado
Chegada de um segmento que preenche completa ou incompletamente um buraco	Se o número do segmento coincidir com o limite inferior do buraco envia ACK imediatamente.