

The background of the image is a high-angle aerial photograph of a massive, intricate highway interchange. The roads are a dark grey color with yellow dashed lines. There are several levels of elevated highways, some with traffic moving in both directions. Interspersed among the asphalt are large, vibrant green lawns and several circular flower beds filled with red flowers. The overall scene is one of a well-planned urban infrastructure.

IMD0043

---

CAMADA DE TRANSPORTE



# HOJE FALAREMOS SOBRE CONFIABILIDADE

- ▶ O que o **princípio fim-a-fim** no diz?
  - ▶ Coloque complexidade (como confiabilidade) no dispositivo final, e não na rede
- ▶ O **modelo em camadas** coloca confiabilidade em qual camada?
  - ▶ Acima da camada de rede
  - ▶ L4 foca na entrega processo-a-processo (fluxo)

# SERVIÇO DE MELHOR ESFORÇO (BEST EFFORT): L3

- ▶ Pacotes podem ser perdidos
- ▶ Pacotes podem ser corrompidos
- ▶ Pacotes podem chegar fora da ordem
- ▶ Pacotes podem atrasar
- ▶ Pacotes podem ser duplicados
- ▶ ...

# MISSÃO

- ▶ Construir uma pilha que suporte **comunicação confiável**
  - ▶ Para que aplicações individuais não precisem lidar com perda de pacotes, etc.
- ▶ Que mecanismos podemos incluir na camada de transporte para prover confiabilidade?
- ▶ Confiabilidade é baseada em um único "**fluxo**"
  - ▶ Fluxo: *Stream* de pacotes entre dois processos

# QUATRO OBJETIVOS PARA TRANSPORTE CONFIÁVEL

## Corretude

- ▶ Um mecanismo de transporte é confiável se e somente se:
  - ▶ Reenvia todos os pacotes corrompidos ou dropados
  - ▶ Tenta fazer progresso!
  - ▶ Se houver dados para enviar, o transporte eventualmente tentará enviar dados

## Equidade

- ▶ Todo fluxo deve ter uma parte justa dos recursos da rede

## Desempenho

- ▶ Latência, jitter, etc.

## Utilização

- ▶ Maximizar o uso da largura de banda

## NOTA

- ▶ Um mecanismo de transporte pode até “desistir”, mas terá que comunicar a aplicação
- ▶ Se o mecanismo de transporte tentou por algum tempo entregar os dados, e não teve sucesso:
  - ▶ Pode decidir que é melhor desistir
  - ▶ E as aplicações podem reiniciar sua transferência de dados
- ▶ **Mas nunca pode alegar falsamente que entregou um pacote!**

# SOLUÇÃO #1

- ▶ Enviar os pacotes sempre e o mais rápido possível
- ▶ Lembrando que o mecanismo de transporte deve tentar fazer progresso
- ▶ Não temos como conferir se um pacote foi dropado ou corrompido
  - ▶ Então ficaria enviando sempre o mesmo pacote? sem progresso em outros fluxos!
- ▶ O que faltaria? ***feedback*** do destinatário
- ▶ Se o destino não responder, a origem nunca vai saber quando parar de retransmitir!

# FORMAS DE FEEDBACK

- ▶ ACK: Sim, recebi o pacote.
- ▶ NACK: Não, não recebi o pacote.
- ▶ Quando o NACK é uma boa ideia?
  - ▶ Pacote corrompido (*Recebi o pacote #5, mas estava corrompido!*)
- ▶ Vamos ignorar o NACK...

# SOLUÇÃO #2

- ▶ Retransmitir até receber um ACK do destinatário
  - ▶ E destino irá enviar ACKs por pacote até os dados finalmente terminarem
- ▶ **Funciona (corretude)?**
  - ▶ Todos os pacotes dropados/descartados serão retransmitidas
  - ▶ O mecanismo de transporte fará progresso
- ▶ **Justo?**
  - ▶ Ao longo prazo, sim. Todas as origens terão sua chance de usar o recursos de rede
- ▶ **Desempenho?**
  - ▶ Bom, mas não necessariamente ótimo, pacotes retransmitidos desnecessariamente podem aumentar a latência
- ▶ **Utilização?**
  - ▶ Sub-ótimo, pacotes retransmitidos desnecessariamente

# SOLUÇÃO #3

- ▶ Envia pacote, e agora, **define um *timer***
- ▶ Quando o destino recebe o pacote, envia ACK
- ▶ Se a origem recebe o ACK, transmissão com sucesso
- ▶ Se a origem não recebe o ACK e o *timer* expira, retransmite
- ▶ **Corretude OK, equidade OK**
- ▶ Para **desempenho**, pediria pequenos *timeouts*
- ▶ Para **utilização**, pediria grandes *timeouts*
  - ▶ Aumentar o timer a cada tentativa
  - ▶ Limitar a quantidade máxima de tentativas

# CHEGAMOS ENTÃO A UMA POSSÍVEL SOLUÇÃO

- ▶ Envia pacote, e agora, **define um *timer***
- ▶ Se a origem não recebe o ACK e o *timer* expira, retransmite
  - ▶ e reseta o *timer*
- ▶ *Trade-off* entre desempenho e utilização na seleção do *timeout*
  - ▶ Muito pequeno, retransmissões desnecessárias (utilização ineficiente)
  - ▶ Muito grande, espera desnecessária (baixo desempenho)

# ALGORITMOS BASEADOS EM JANELA

- ▶ Conceitos bastante simples: envia  $W$  pacotes
  - ▶ Quando a origem recebe um ACK de um pacote, envia o próximo pacote da fila
- ▶ Querem que  $W$  seja tal que:
  - ▶ se estou enviando na taxa do enlace (largura de banda do *link*), então
  - ▶ o ACK do primeiro pacote chega exatamente quando eu termino de enviar o último dos meus  $W$  pacotes
- ▶ Ou seja, me deixe enviar tão rápido quanto o caminho pode suportar

# CONSIDERAÇÕES DE PROJETO

- ▶ Natureza do *feedback*
  - ▶ O que o ACK deve nos contar quando temos muitos pacotes sendo transmitidos?
- ▶ Detecção de perdas
- ▶ Resposta a perdas

# ACK INDIVIDUAL POR PACOTE

O destino envia ACK para cada pacote individual que chega

- ▶ Exemplo:
- ▶ Assuma que o pacote #5 é perdido, mas nenhum outro. Stream de ACKs será:
  - ▶ |
  - ▶ 2
  - ▶ 3
  - ▶ 4
  - ▶ 6
  - ▶ 7
  - ▶ 8
  - ▶ ...

# ACK INDIVIDUAL POR PACOTE

## Natureza do *feedback*

- ▶ Simples, o receptor manda ACK para cada pacote

## Detecção de perdas

- ▶ Simples, os ACKs ditam a situação de cada pacote para a origem

## Resposta a perdas

- ▶ Moderado, perda do ACK requer retransmissão

# ACK COM FEEDBACK COMPLETO

Lista todos os pacotes que foram recebidos

- ▶ Envia o maior ACK cumulativo adicionando qualquer outro pacote adicional
- ▶ Exemplo:
  - ▶ Assuma que o pacote #5 é perdido, mas nenhum outro. Stream de ACKs sera:
    - ▶ Até o 1
    - ▶ Até o 2
    - ▶ Até o 3
    - ▶ Até o 4
    - ▶ Até o 4, e o 6
    - ▶ Até o 4, e o 6,7
    - ▶ Até o 4, e o 6,7,8
    - ▶ ...

# ACK COM FEEDBACK COMPLETO

## Natureza do *feedback*

- ▶ Complexo, o feedback pode ter muito overhead ( $ACK(1, 2, 3, 4, 5, \dots, 100)$ )

## Detecção de perdas

- ▶ Simples, os ACKs ditam a situação de cada pacote para a origem

## Resposta a perdas

- ▶ Simples, perda do ACK não necessariamente requer retransmissão
- ▶ O próximo ACK vai dizer se o pacote foi realmente recebido

# ACK CUMULATIVO

- ▶ ACKs individuais podem ser perdidos, e requerem retransmissão desnecessária
- ▶ ACKs com *feedback* completo podem lidar com ACKs perdidos mas tem altos overheads

**ACK cumulativo: um meio termo entre as duas abordagens**

- ▶ Só a primeira parte do ACK com *feedback* completo
- ▶ ACK o maior número de sequência de todos os pacotes previamente recebidos
- ▶ A implementação na verdade normalmente indica o próximo pacote esperado!

# ACK CUMULATIVO (MESMO EXEMPLO, PACOTE #5 PERDIDO)

ACKs com *feedback* completo:

- ▶ Stream de ACKs
  - ▶ Até o 1
  - ▶ Até o 2
  - ▶ Até o 3
  - ▶ Até o 4
  - ▶ Até o 4, e o 6
  - ▶ Até o 4, e o 6, 7
  - ▶ Até o 4, e o 6, 7, 8
  - ▶ ...
- ▶ Informa quais pacotes chegaram, e qual pacote não chegou

ACKs cumulativos:

- ▶ Stream de ACKs
  - ▶ Até o 1
  - ▶ Até o 2
  - ▶ Até o 3
  - ▶ Até o 4
  - ▶ ...
- ▶ Informa que alguns pacotes chegaram, e qual pacote não chegou

# ACK CUMULATIVO (COMO A REORDENAÇÃO É VISTA)

Eventos no destino:

- ▶ Pacote 1 recebido
- ▶ Pacote 2 recebido
- ▶ Pacote 3 recebido
- ▶ Pacote 4 recebido
- ▶ **Pacote 6 recebido**
- ▶ Pacote 7 recebido
- ▶ **Pacote 5 recebido**
- ▶ Pacote 8 recebido
- ▶ ...

ACKs cumulativos enviados:

- ▶ Até o 1
- ▶ Até o 2
- ▶ Até o 3
- ▶ Até o 4
- ▶ Até o 4
- ▶ Até o 4
- ▶ Até o 7
- ▶ Até o 8
- ▶ ...

# O QUE VIMOS ATÉ ENTÃO...

- ▶ Corretude para transporte confiável!
- ▶ ... entender por que ***feedback do receptor*** é necessário (solução #1)
- ▶ ... entender por que ***timers*** podem ser necessários (solução #2)
- ▶ ... entender por que projeto baseado em ***janelamento*** pode ser necessário (solução #3)
- ▶ ... entender por que ***ACKs cumulativos*** podem ser uma boa ideia
- ▶ Muito próximo ao protocolo **TCP**!

# CAMADA DE TRANSPORTE

- ▶ Camada de transporte oferece a abstração de um "*pipe*" para as aplicações
- ▶ Dados entram num lado do *pipe* e saem do outro
- ▶ **Pipes são entre processos, e não entre hosts**
- ▶ Existem basicamente dois tipos de abstrações de *pipes*

# DUAS ABSTRAÇÕES DE PIPES

## Entrega de pacotes não confiável (UDP)

- ▶ não confiável (aplicação responsável por retransmissões)
- ▶ mensagens limitadas a um único pacote

## Entrega confiável de fluxo (*stream*) de bytes (TCP)

- ▶ Bytes inseridos no *pipe* pelo transmissor
- ▶ Eles chegam, em ordem, no receptor (para a aplicação)

# UDP (USER DATAGRAM PROTOCOL)

- ▶ Origens enviam pacotes
- ▶ Destinos não fazem nada, somente recebem os pacotes
- ▶ Se pacotes atrasam/fora de ordem/perdidos:
  - ▶ ...
  - ▶ Deixa a aplicação resolver!
  - ▶ Se a aplicação precisa de entrega confiável, deve usar transporte confiável!
- ▶ Basicamente um extensão mínima do IP...

# TCP (TRANSMISSION CONTROL PROTOCOL)

- ▶ Origem enviam **segmentos**
- ▶ Destinos enviam ACKs
- ▶ Origem retransmite segmentos perdidos e/ou corrompidos
- ▶ Origem realiza **controle de fluxo** (para não sobrecarregar (overflow) o receptor)
- ▶ Origem realiza **controle de congestionamento** (para não sobrecarregar a rede)
- ▶ Origem e destino participam de um processo estabelecimento (e encerramento) de "**conexão**"

# CONEXÕES (OU SESSÕES)

- ▶ Confiabilidade exige armazenar o **estado**
  - ▶ Transmissor: pacotes enviados mas ainda não foram ACKed, e respectivos timers
  - ▶ Receptor: pacotes que chegarem fora da ordem
- ▶ Cada fluxo de bytes é chamado de **conexão** ou sessão
  - ▶ Cada um com seu estado de conexão
  - ▶ Estados estão nos hosts, e não na rede

# PORTAS

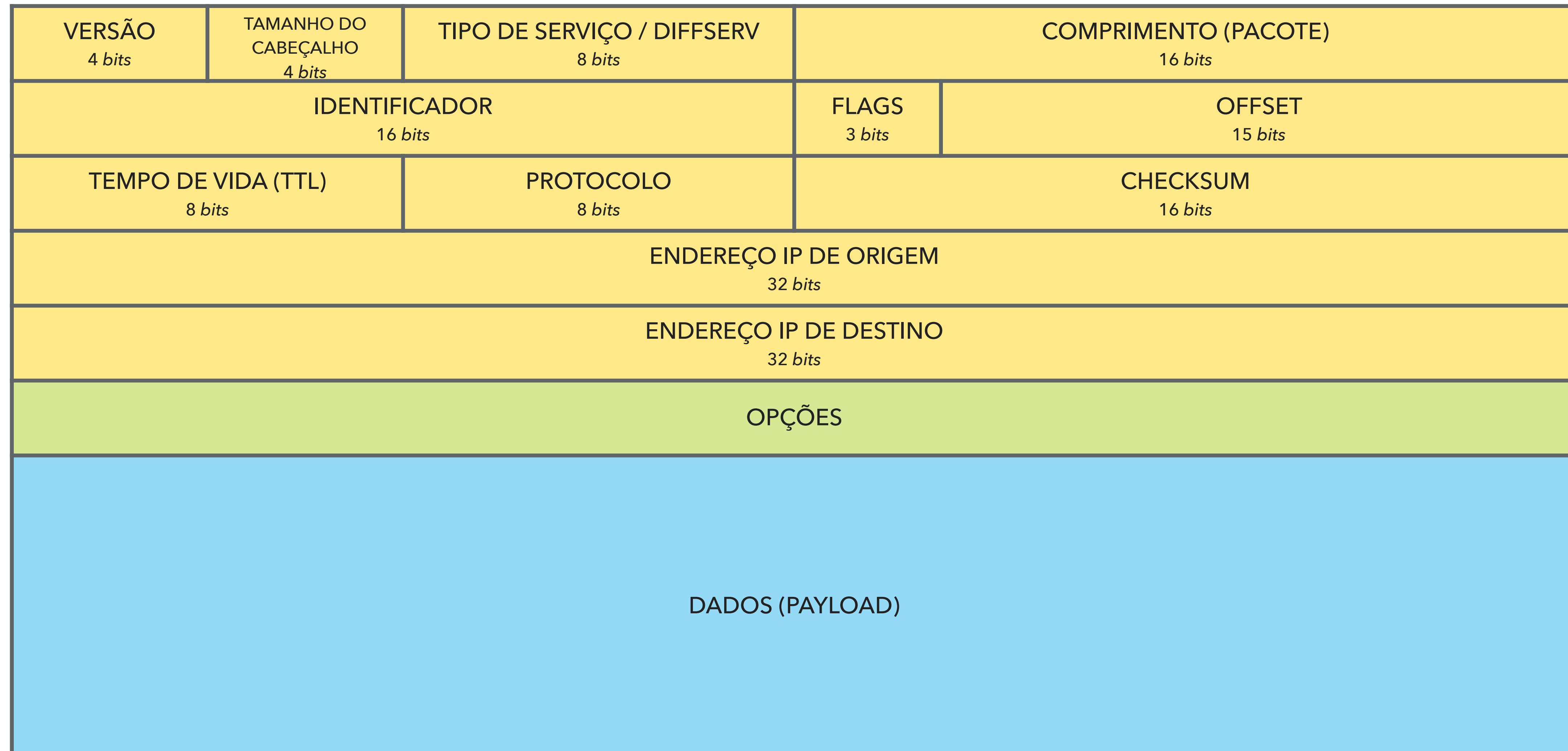
- ▶ Espaço de endereçamento de 16 bits
  - ▶ TCP tem um espaço, UDP tem outro
- ▶ Portas "conhecidas" (well known) (0-1023):
  - ▶ Acordo sobre quais serviços rodam em quais portas
    - ▶ ex: SSH 22, HTTP 80, ...
  - ▶ Cliente (aplicação) sabe a porta apropriada no servidor
  - ▶ Serviços podem escutar nas portas conhecidas

# MULTIPLEXAÇÃO E DEMULTIPLEXAÇÃO

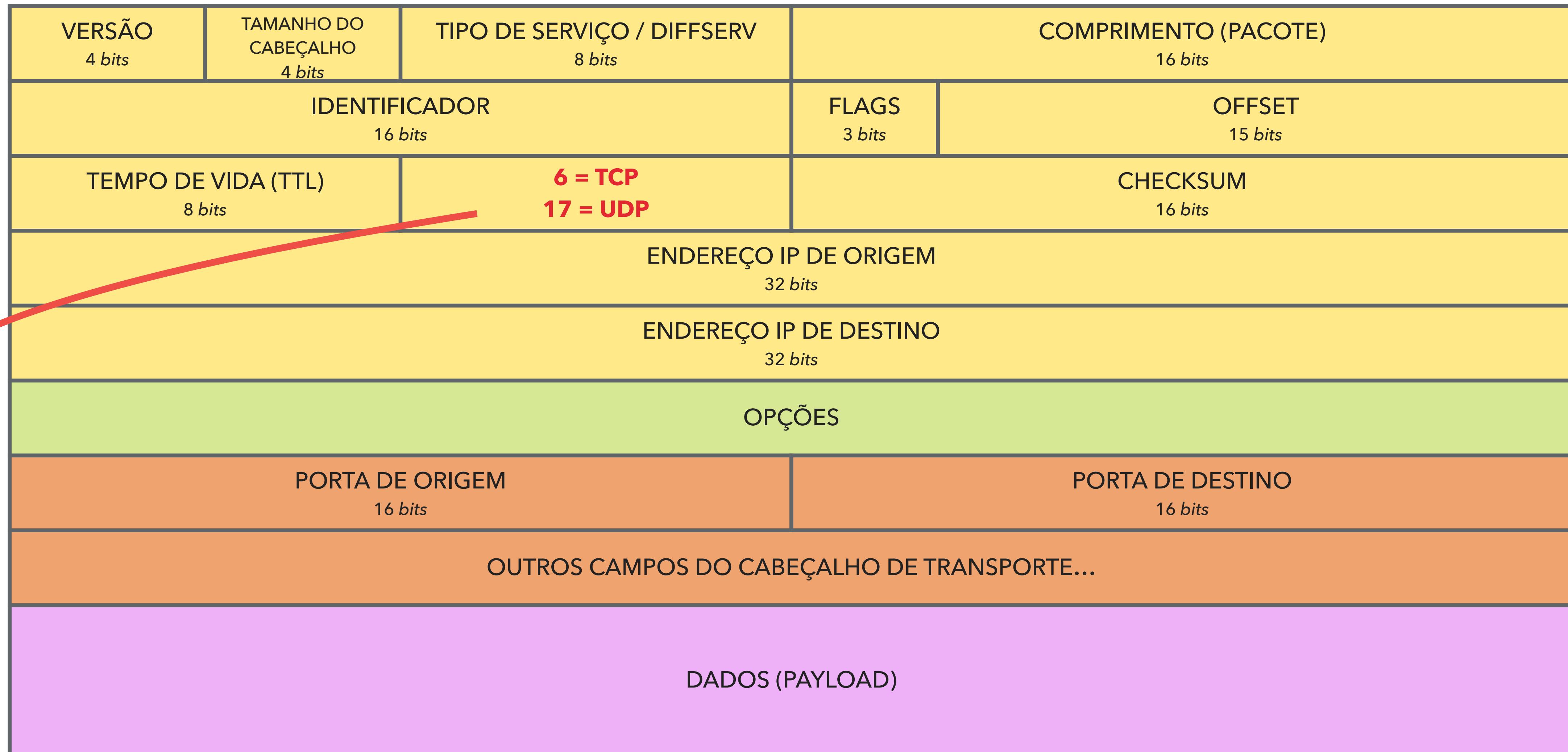
- ▶ Host recebe pacote IP
  - ▶ Cada pacote tem um endereço IP de origem e de destino
  - ▶ Cada segmento tem uma porta de origem e de destino
- ▶ Host utiliza o endereço IP e os números de porta para direcionar o segmento para o socket apropriado

Porta de Origem	Porta de Destino
Outros campos do cabeçalho	
Dados da aplicação (mensagem)	

# ESTRUTURA DO PACOTE IP



# ESTRUTURA DO PACOTE IP



# TCP (TRANSMISSION CONTROL PROTOCOL)

Confiável, entrega ordenada

- ▶ Garante que o fluxo de *bytes* eventualmente é entregue íntegro

Orientado a conexão

- ▶ Estabelecimento e encerramento explícito de uma sessão TCP (handshakes)

Fluxo de *bytes* bidirecional

- ▶ Envia e recebe *stream* de *bytes*, e não mensagens

Controle de fluxo e congestionamento

- ▶ Garante que o remetente não sobrecarregue o destinatário
- ▶ Adaptação dinâmica à capacidade do caminho

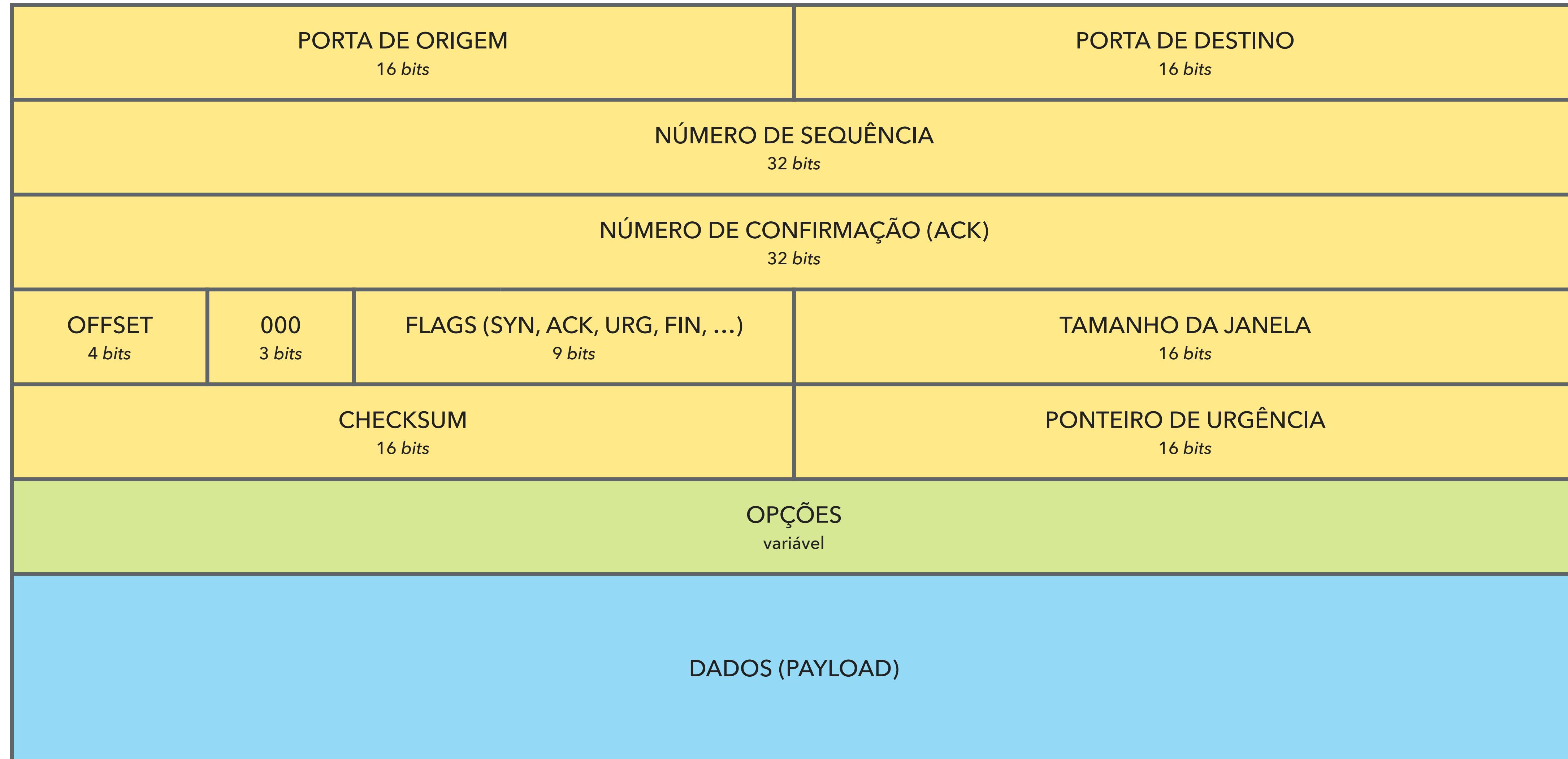
# COMPONENTES BÁSICOS PARA PROVER CONFIABILIDADE

- ▶ ACKs
  - ▶ TCP utiliza o número de sequência dos *bytes* para identificar o *payload*
  - ▶ ACKs se referem a esses números
- ▶ *Timeouts* e retransmissões
  - ▶ TCP retransmite baseado em *timeouts*
  - ▶ *Timeouts* baseados no RTT (estimativa)

# OUTRAS CONSIDERAÇÕES

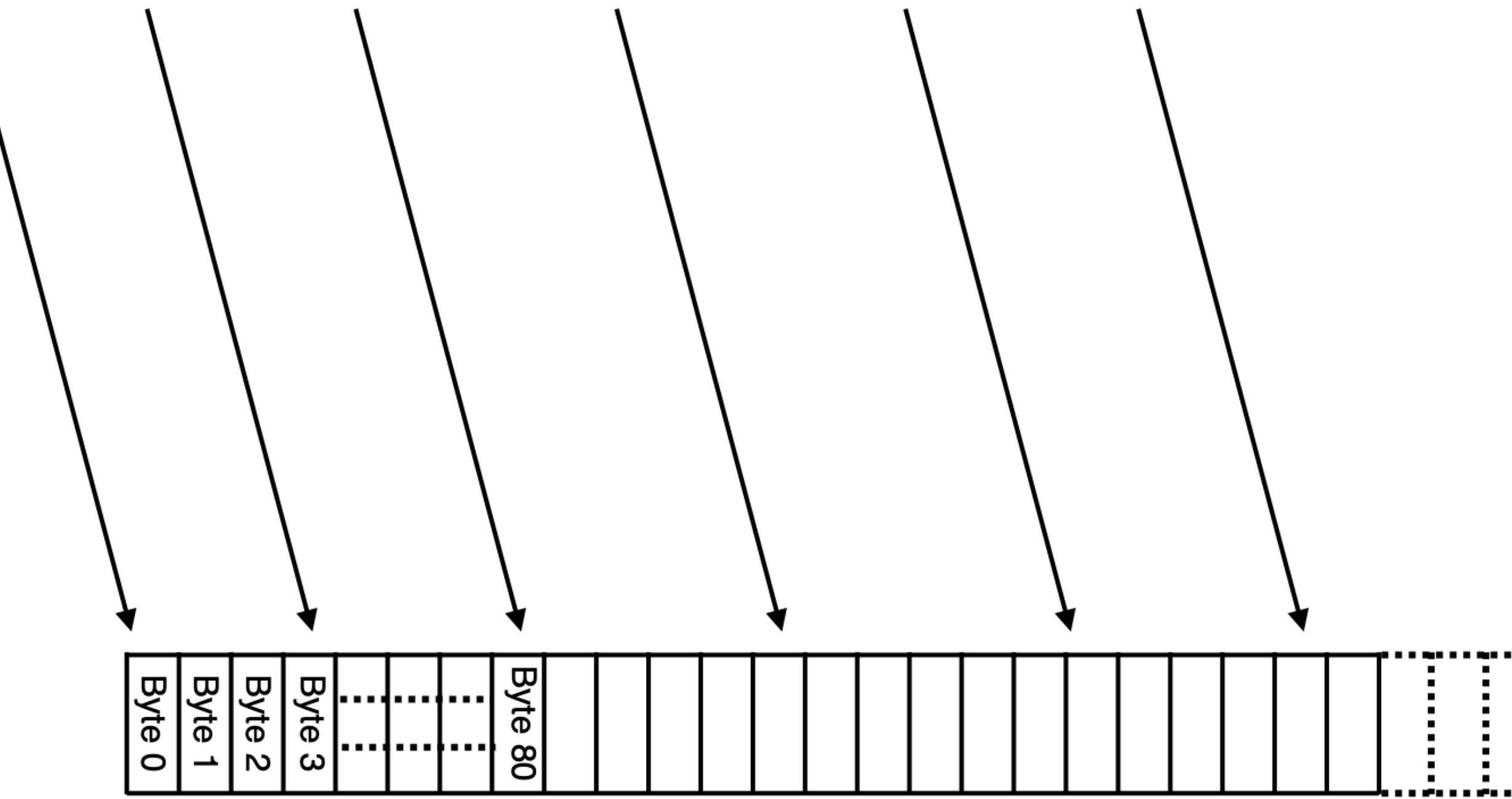
- ▶ Controle de fluxo com o janela deslizante
  - ▶ Permite que  $W$  bytes sequenciais estejam “viajando”
- ▶ ACKs cumulativos
- ▶ Define timer após cada *payload* é ACKed
  - ▶ O ACK indica qual o próximo byte esperado
  - ▶ Quando o timer expira, reenvia o *payload* e espera...
  - ▶ Duplica o período de timeout

# CABEÇALHO TCP



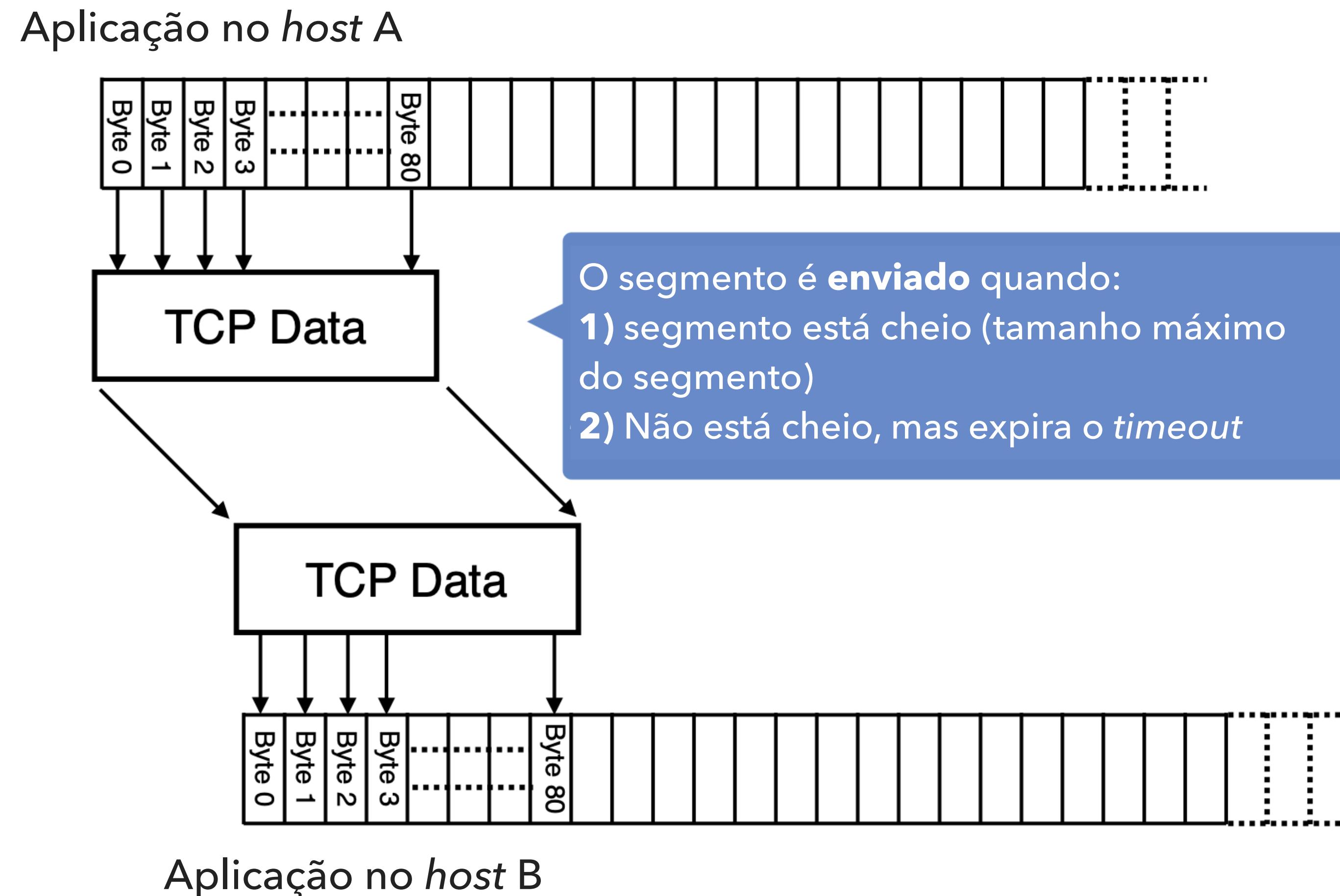
# SERVIÇO DO TCP: STREAM DE BYTES

Aplicação no host A



Aplicação no host B

# SERVIÇO DO TCP: STREAM DE BYTES



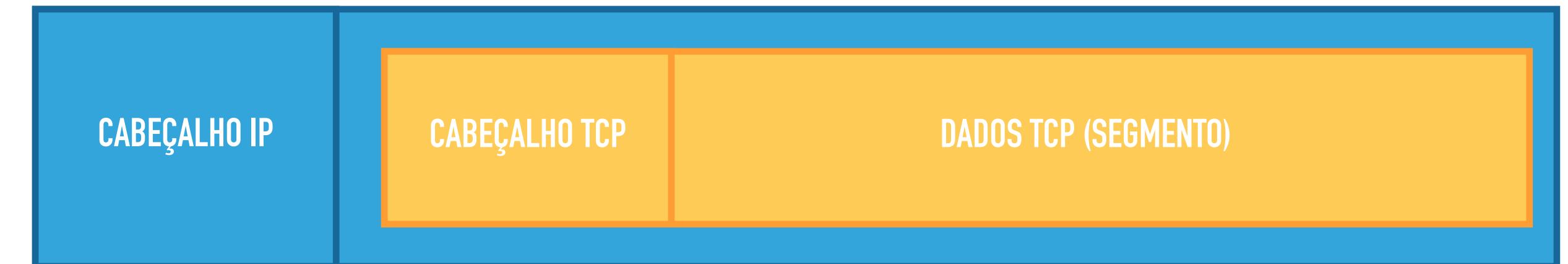
# SEGMENTO TCP

## Pacote IP

- ▶ Não pode ser maior que o MTU (*Maximum Transmission Unit*)
- ▶ Ex: 1500 bytes para Ethernet

## Pacote TCP

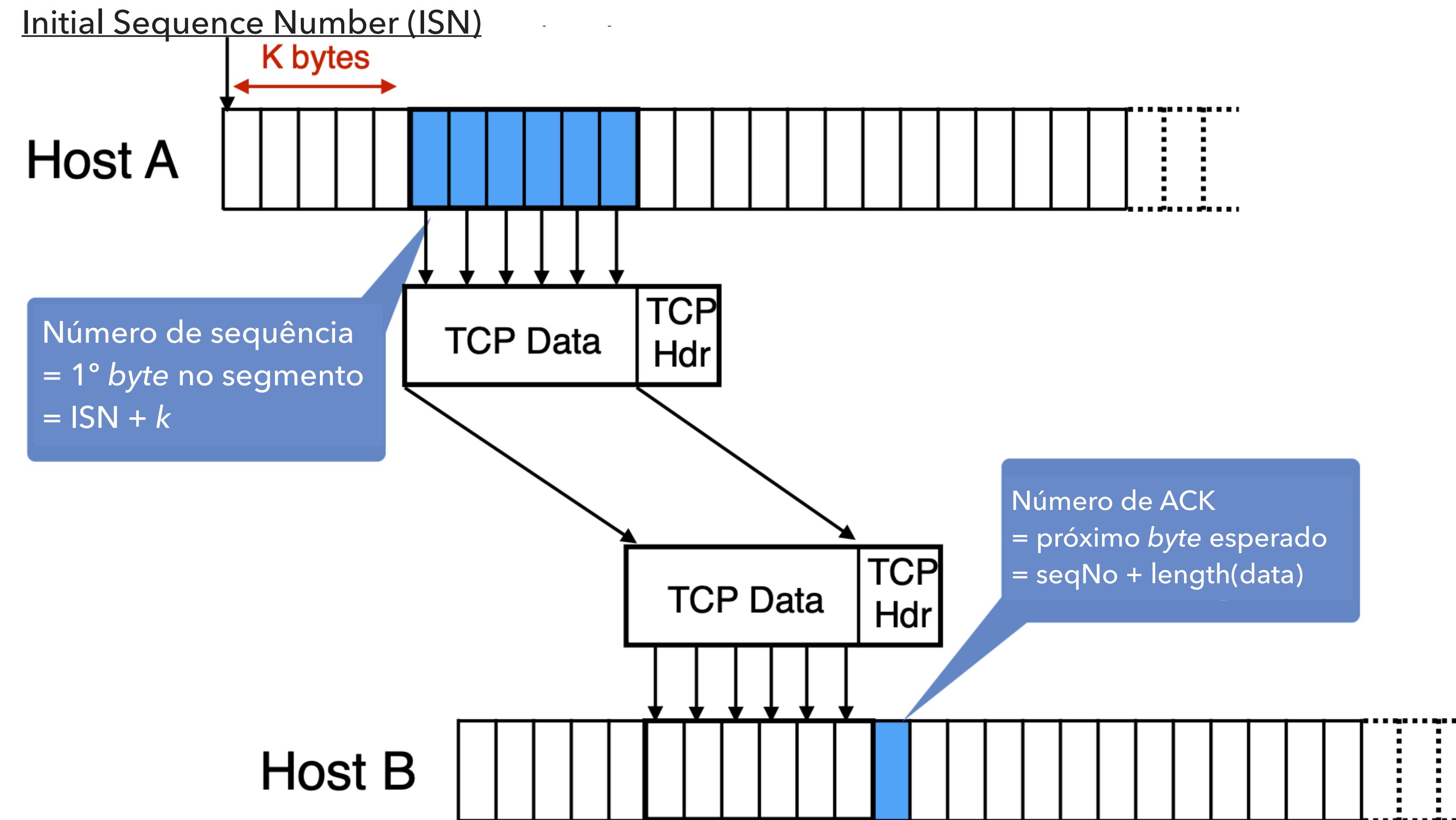
- ▶ Pacote IP com um cabeçalho TCP e dados
- ▶ Cabeçalho TCP  $\geq 20$  bytes



## Segmento TCP

- ▶ Não pode ser maior que o MSS (*Maximum Segment Size*)
- ▶ Ex: até 1460 bytes consecutivos do fluxo
- ▶  $MSS = MTU - \text{cabeçalho IP} - \text{cabeçalho TCP}$

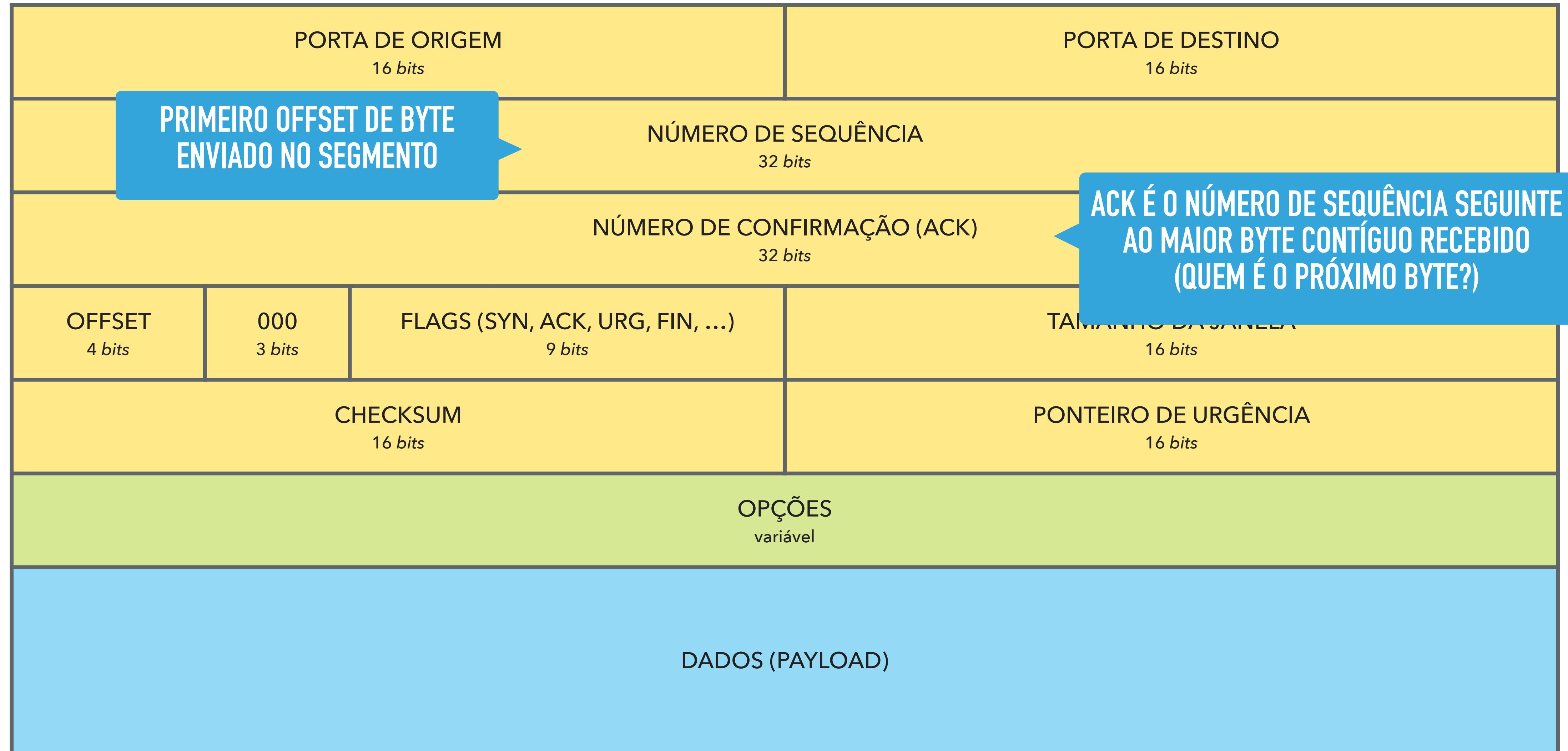
# SERVIÇO DO TCP: STREAM DE BYTES



# NÚMEROS DE RECONHECIMENTO E SEQUÊNCIA

- ▶ Transmissor envia segmentos (fluxo de *bytes*)
  - ▶ Dados começam com o número de sequência X
  - ▶ Pacote contém B *bytes*
    - ▶ X, X+1, X+2, ..., X+B-1
- ▶ No recebimento de um segmento, o receptor envia um ACK
  - ▶ Se todos os dados antes de X já foram recebidos:
    - ▶ ACK reconhece X+B (que é o próximo byte esperado)
  - ▶ Se o byte contíguo recebido mais alto é um valor menor Y
    - ▶ ACK reconhece Y+1
  - ▶ Mesmo que esse byte já tenha sido reconhecido (ACK) antes

# CABEÇALHO TCP



# CONTROLE DE FLUXO

- ▶ Janela anunciada:  $W$ 
  - ▶ Pode enviar  $W$  bytes além do byte esperado
- ▶ Receptor utiliza  $W$  para prevenir que o transmissor "estoure" seu buffer
- ▶ Limita o número de bytes o transmissor pode ter “viajando”

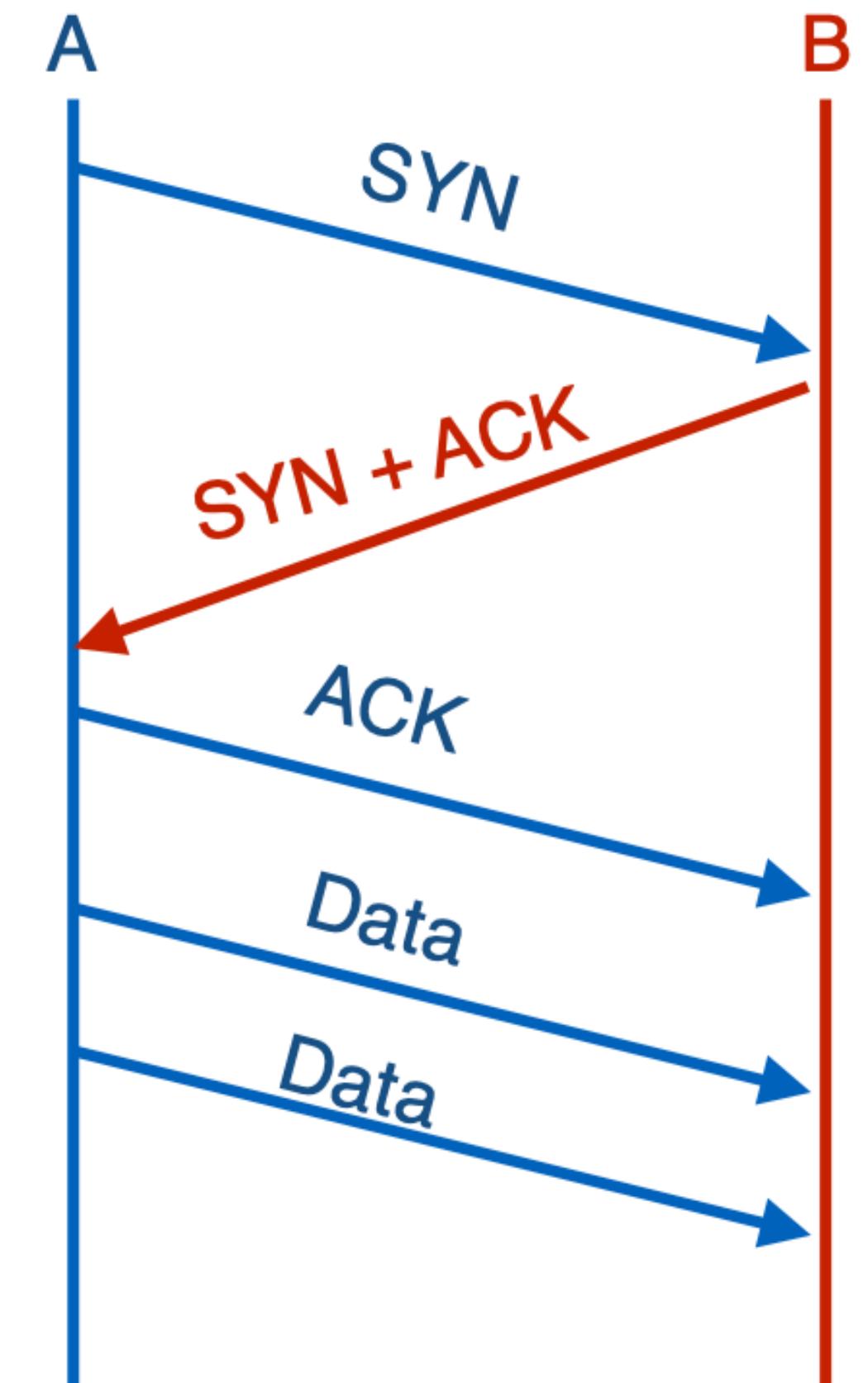
# ESTABELECIMENTO DE CONEXÃO TCP

- ▶ Número de sequência para o primeiro byte (*Initial Sequence Number*)
  - ▶ Por que não usar  $ISN = 0$  ?
- ▶ Utilização prática:
  - ▶ Endereços IP e números de porta identificam unicamente uma conexão
  - ▶ Eventualmente, essas portas **são utilizadas novamente**
  - ▶ ... pequena chance de um pacote antigo ainda estar **a caminho do destino**
- ▶ TCP quer que se mude o ISN
  - ▶ Utiliza relógio como fonte de origem
- ▶ Para estabelecer uma conexão, os hosts trocam ISNs

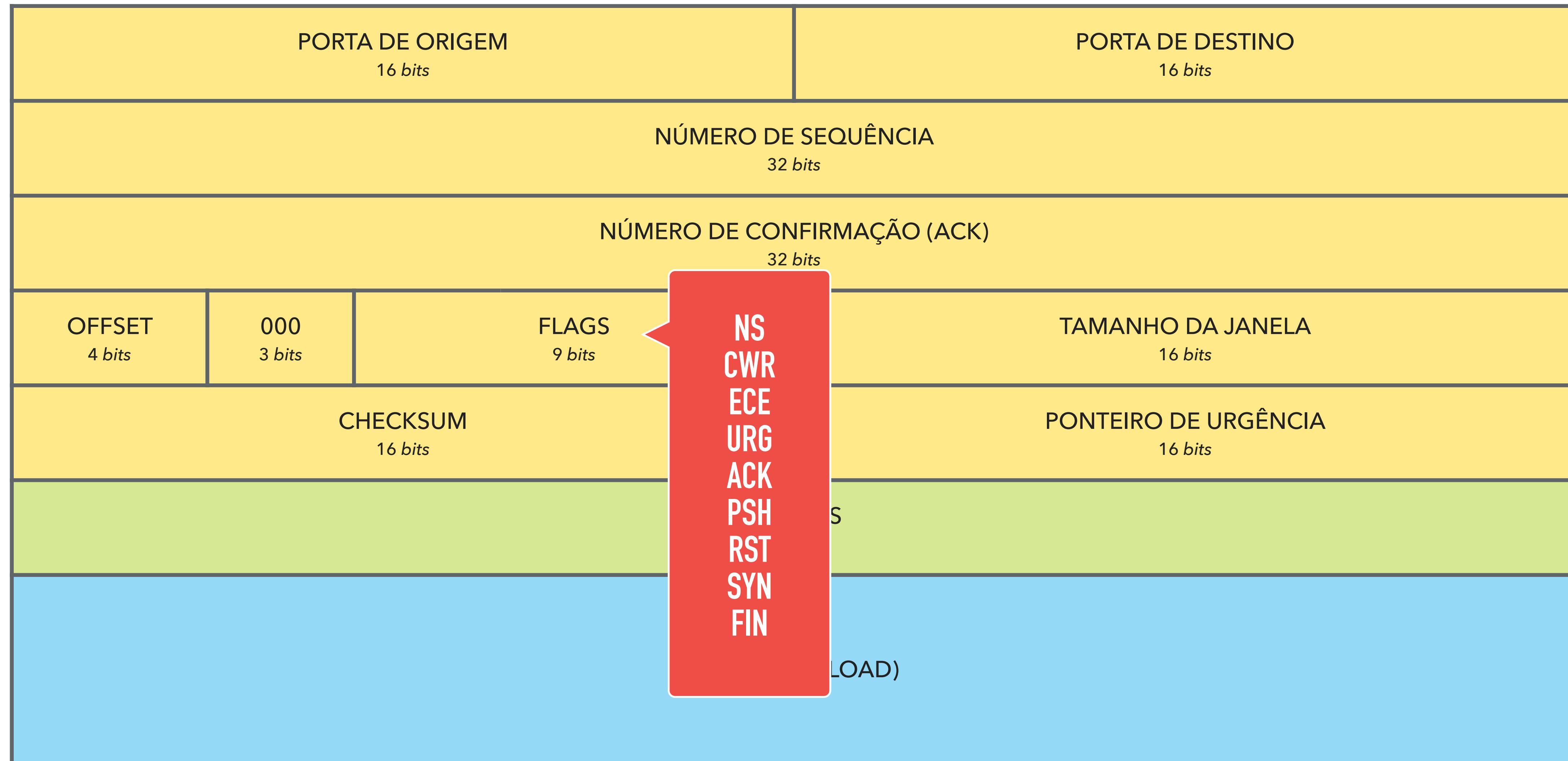
# ESTABELECIMENTO DE CONEXÃO TCP

Cada host informa o seu ISN para o outro

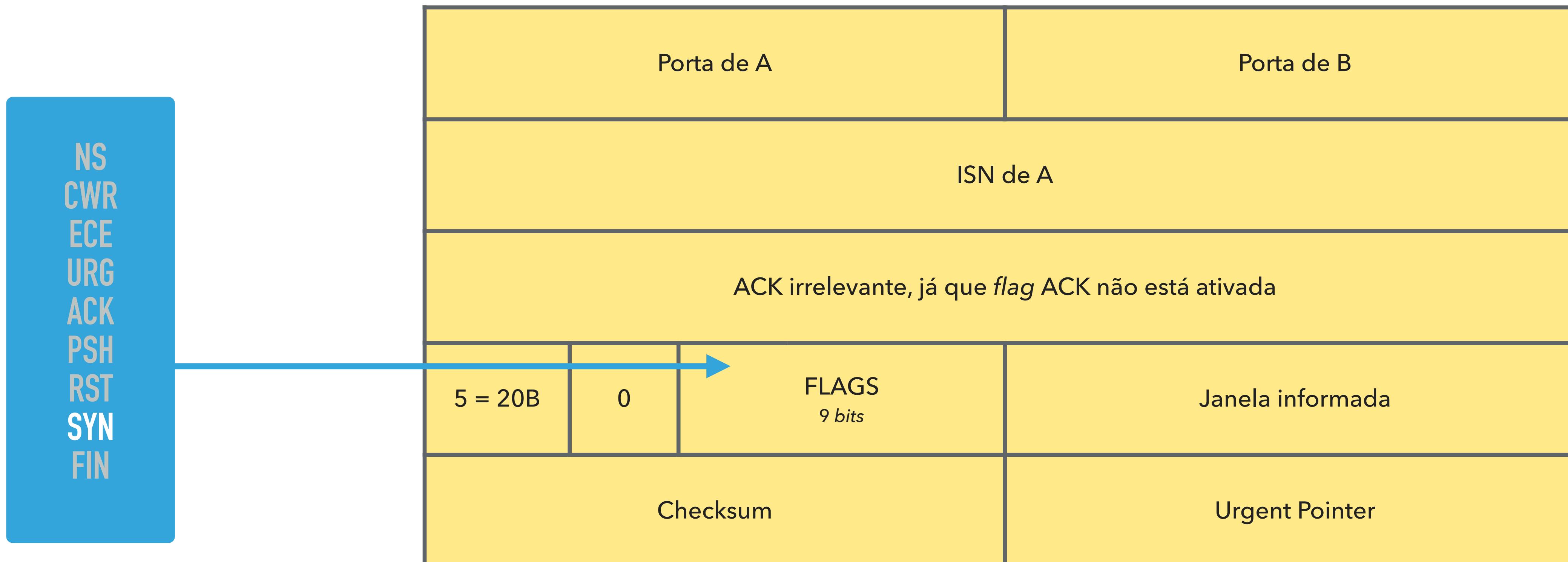
- ▶ *Three-way handshake* para estabelecimento de conexão TCP
- ▶ Host A envia um SYN
- ▶ Host B retorna um reconhecimento SYN (SYN-ACK)
- ▶ Host A envia um ACK para reconhecer o SYN-ACK recebido



# FLAGS

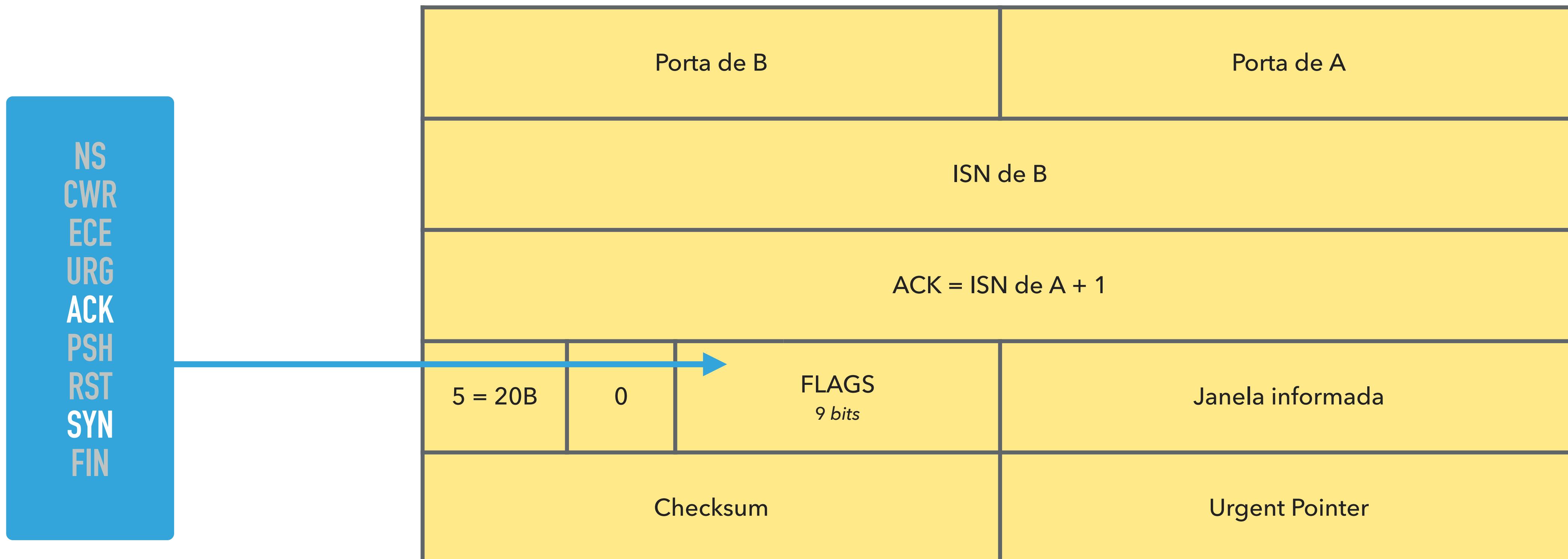


# THREE WAY HANDSHAKE: PASSO 1



A informa B que quer estabelecer uma conexão...

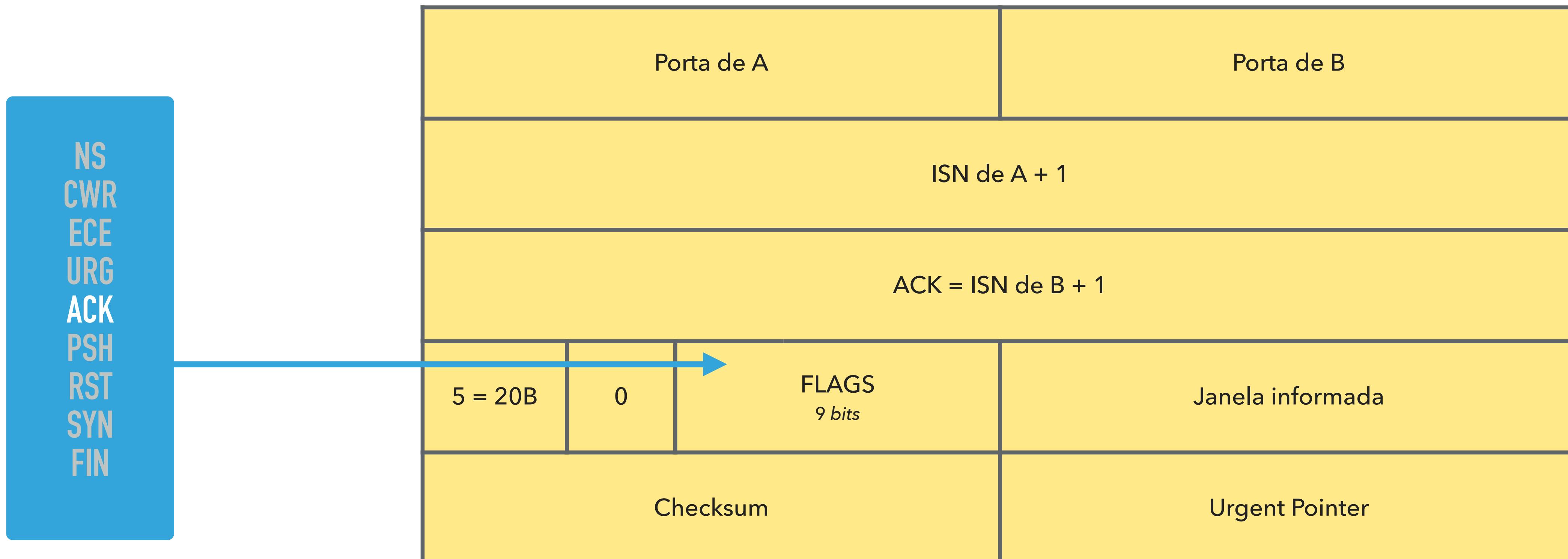
# THREE WAY HANDSHAKE: PASSO 2



B informa A que aceita e está pronto para receber o próximo *byte*...

... ao receber o pacote, A pode começar a enviar dados

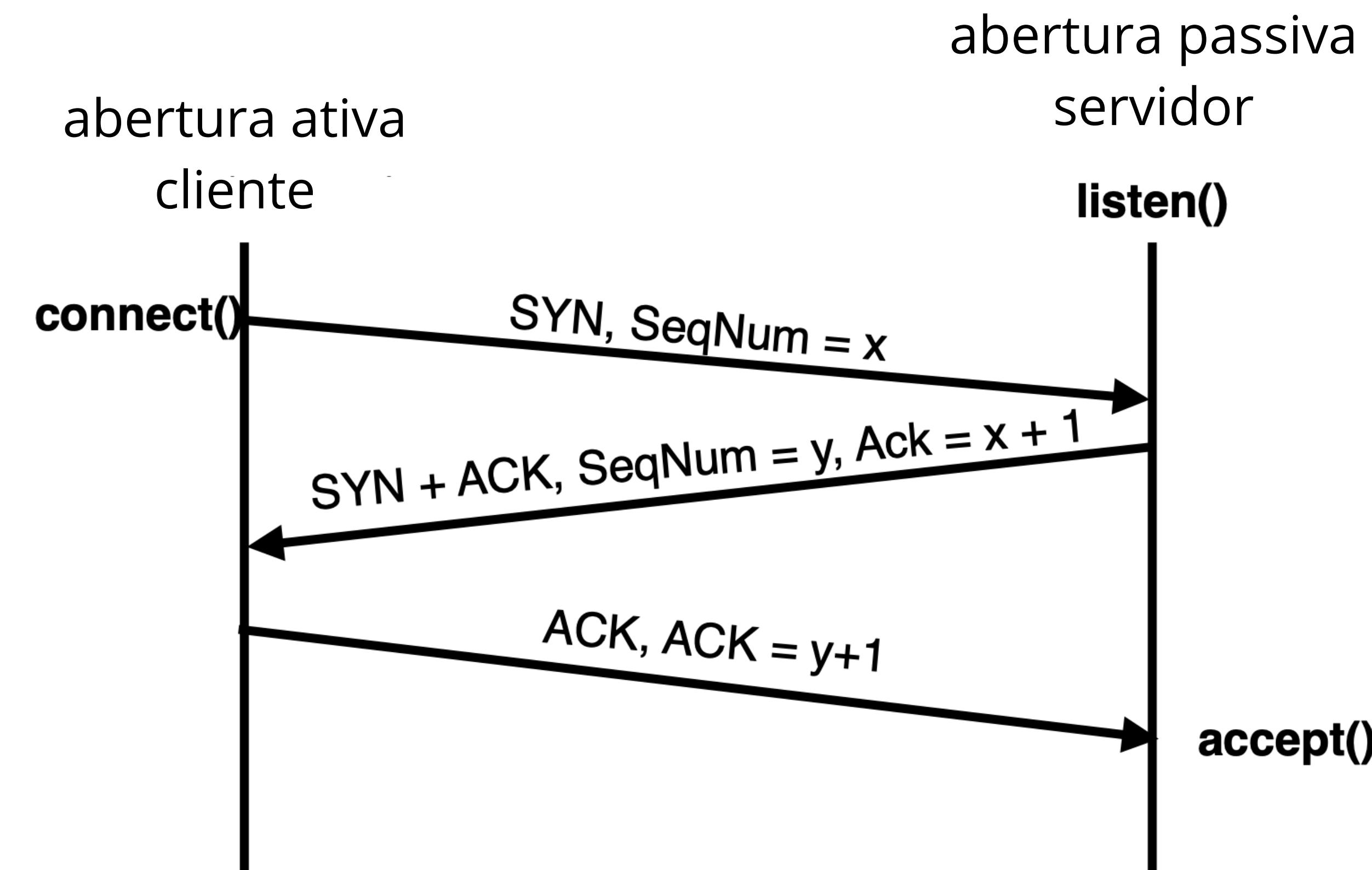
# THREE WAY HANDSHAKE: PASSO 3



A informa B que ele também está pronto para receber *bytes*...

... ao receber o pacote, B pode começar a enviar dados

# DIAGRAMA TEMPORAL

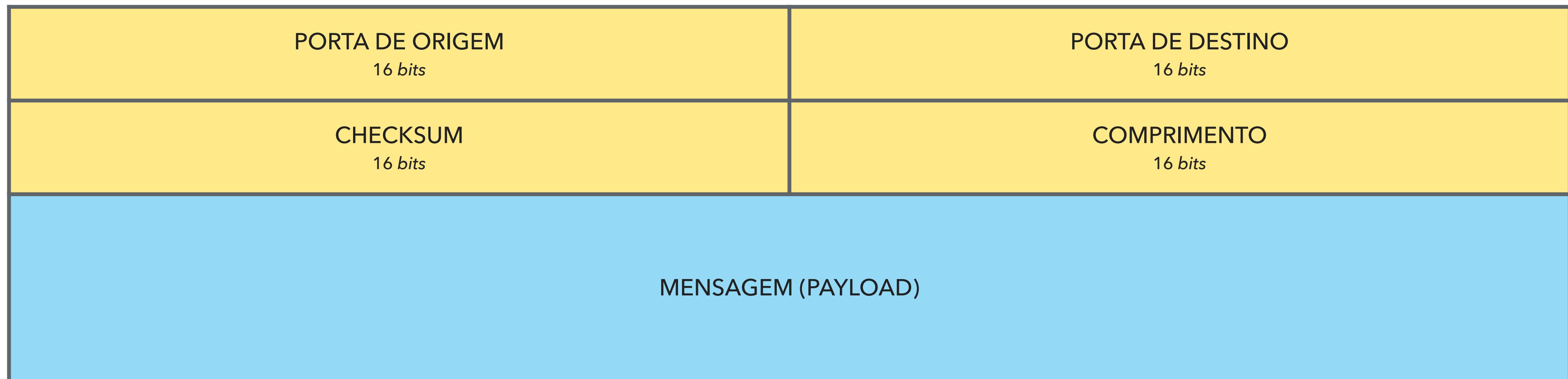


## NOTA: TCP É DUPLEX

- ▶ Uma conexão TCP entre A e B pode carregar dados em **ambas as direções**
- ▶ Pacotes **podem carregar dados e ACKs**
- ▶ Se a *flag* ACK está setada, então está reconhecendo dados

# UDP (USER DATAGRAM PROTOCOL)

- ▶ Comunicação "leve" entre processos
- ▶ Evita overhead e atrasos da entrega confiável e ordenada
- ▶ Envia e recebe mensagens por um socket



# APLICAÇÕES QUE USAM UDP

- ▶ Aplicações de streaming interativo
  - ▶ Retransmitir pacotes perdidos/corrompidos é irrelevante
    - ▶ Ex: chamadas telefônicas, video-conferência, jogos eletrônicos, ...
    - ▶ Apesar de que alguns protocolos modernos de *streaming* usam TCP (e HTTP)
  - ▶ Protocolos simples de consulta, como o DNS