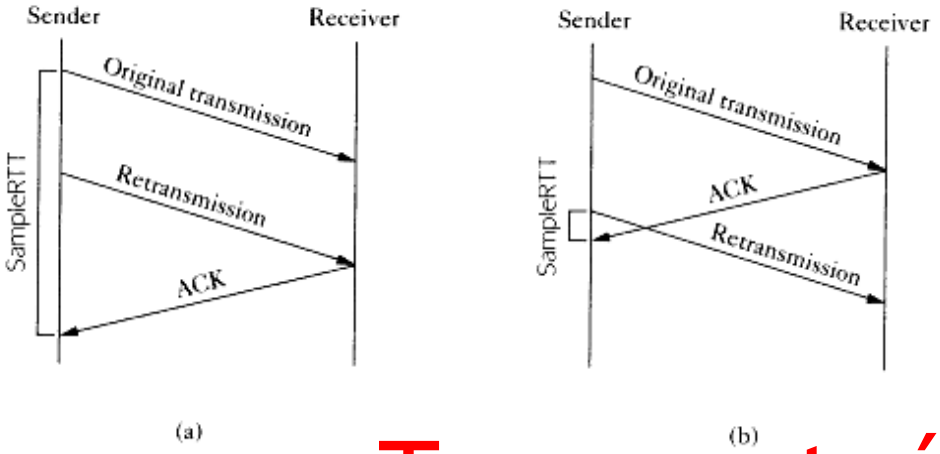
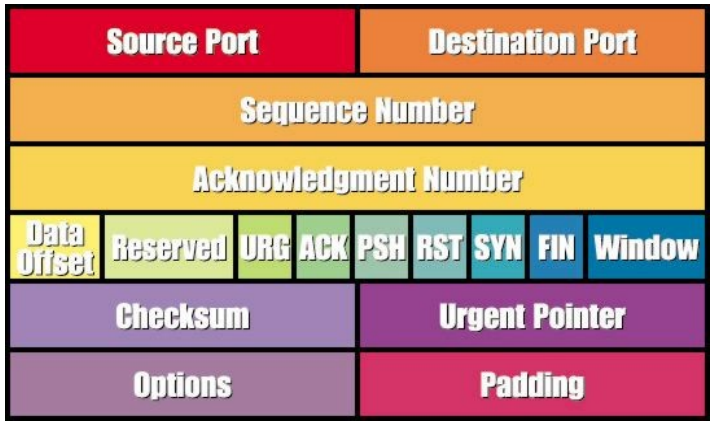
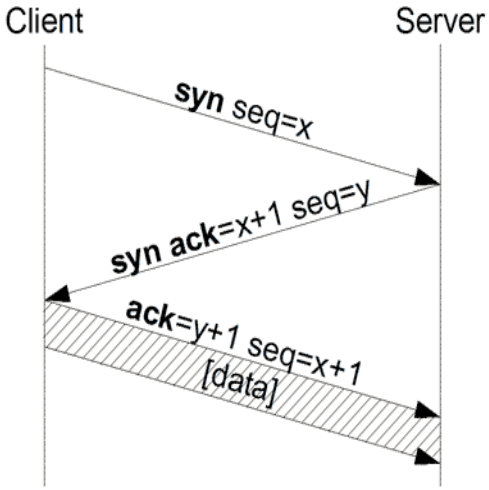
# 5. prednáška




# Transportná vrstva

# 2.časť




| Source Port (16 bits) | Destination Port (16 bits) |
|---|---|
| Length (16 bits) | Checksum (16 bits) |
| Data.... | |

# Transportná vrstva: čo už vieme

- transportná vrstva, v prípade TCP, u odosielateľa delí správy (prúd dát) aplikačnej vrstvy na časti, z ktorých pridaním hlavičky vznikajú segmenty
- transportná vrstva príjemcu extrahuje dáta zo segmentov a poskytuje ich aplikačnej vrstve
- hlavnou úlohou transportných protokolov je umožniť odosielanie a prijímanie dát aplikácie cez sokety
- soket je otvorený procesom alebo vláknom procesu, pričom musí určiť port a rozhranie (jeho IP adresu), cez ktoré bude komunikovať
- soket je jednoznačne určený:
  - v UDP IP adresou príjemcu a portom príjemcu
  - v TCP IP adresami príjemcu a odosielateľa a portami príjemcu a odosielateľa

# Transportná vrstva: čo už vieme

- Vlastnosti protokolu UDP:
  - vysielanie bez nadviazania spojenia
  - "umožňuje" vyslanie dát k viacerým staniciam (napr. streaming multimédií)
  - počítanie kontrolných súčtov
  - nepotvrdzovaný prenos dát
    - proces príjemcu môže dostať dáta v inom poradí
    - stratené segmenty sa ignorujú
  - nemá kontrolu toku dát
  - nemá kontrolu zahltenia siete

# Potvrdzovaný prenos dát: čo už vieme

- Potvrdzovaný prenos dát cez nespoľahlivý kanál:
  - kontrolný súčet
  - sekvenčné čísla
  - potvrdenia (ACK) so sekvenčnými číslami
  - časovač
- Pipelining
  - zvýšenie efektivity
  - okno odosielateľných segmentov bez potvrdenia
  - okno prijatých segmentov
  - viac sekvenčných čísiel (≥ 2 * veľkosť okna)
  - kumulatívne potvrdenie

# Osnova rozprávania o transportnej vrstve

- 3.1 Služby transportnej vrstvy
- 3.2 Delenie správ a adresácia soketov
- 3.3 UDP: bezstavový transportný protokol
- 3.4 Princípy potvrdzovaného toku dát
- 3.5 TCP: stavový transportný protokol
  - štruktúra segmentu
  - pripájanie a odpájanie
  - potvrdzovaný tok dát
  - kontrola toku dát
- 3.6 Princípy zabezpečenia kontroly zahltenia
- 3.7 Kontrola zahltenia v protokole TCP

# TCP: vlastnosti

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - komunikácia vždy presne dvoch soketov
- potvrdzovaný prúd bajtov pre aplikáciu v správnom poradí
- pipelining:
  - posiela ďalšie segmenty bez okamžitého potvrdenia predchádzajúcich
- buffre odosielateľa a príjemcu
- full duplex:
  - obojsmerný tok dát v tom istom spojení
- so spojením:
  - handshaking (zahájenie spojenia cez riadiace segmenty) pred odosielaním prvých dát
- kontrola toku dát:
  - odosielateľ nezahltí príjemcu
- kontrola zahltenia siete:
  - odosielateľ nezahltí zariadenia na ceste k cieľu

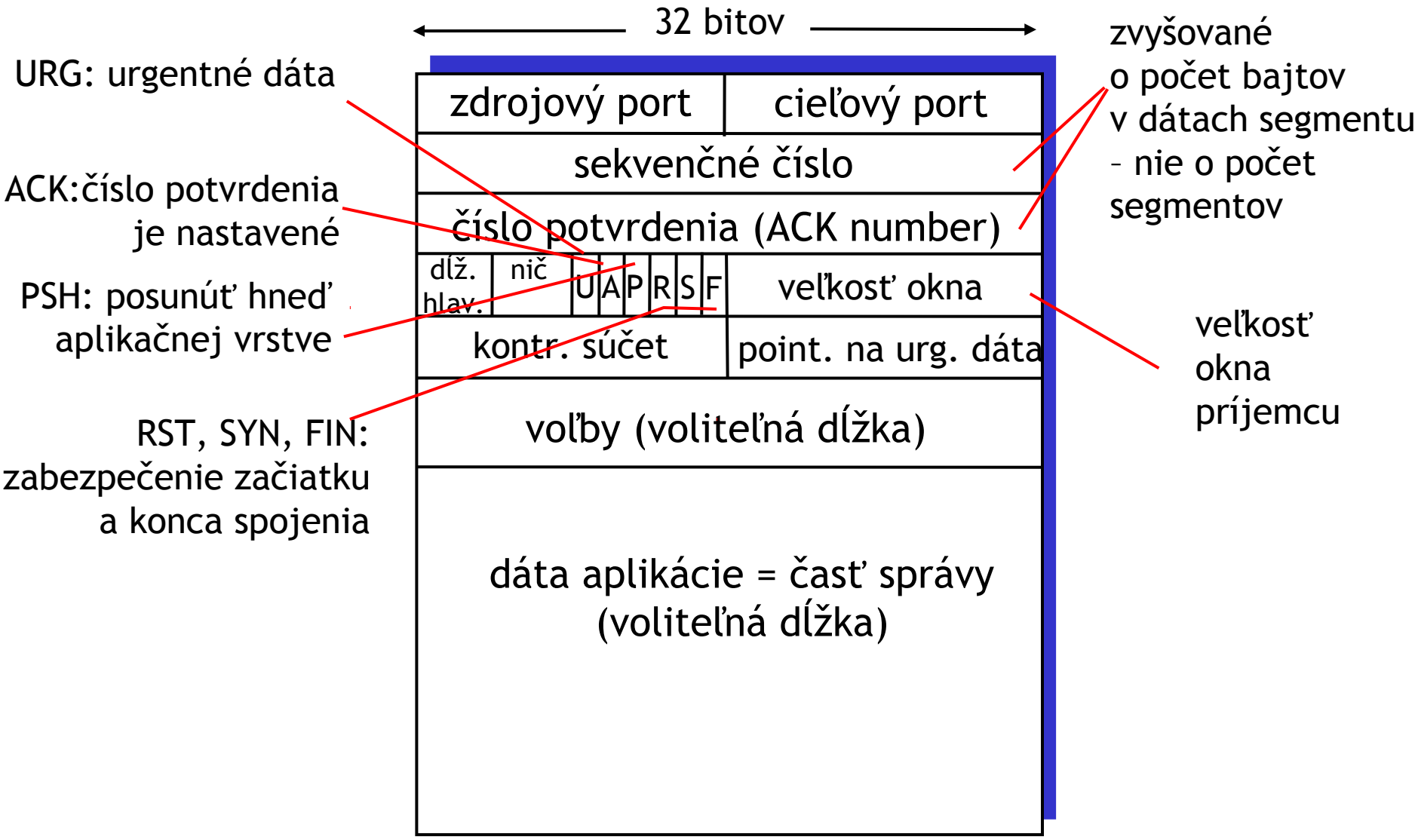
# Potvrdzovaný prenos dát cez TCP

- TCP využíva nespoľahlivý (best effort) prenos dát protokolom IP
- pipelining segmentov
  - okno príjemcu aj odosielateľa
- kumulatívne potvrdenia
- jediný časovač na preposielania
- opätovné posielanie je spôsobené:
  - timeout-om
  - viacnásobným potvrdením
- na začiatok uvažujme zjednodušeného odosielateľa:
  - ignorujme kontrolu toku dát a zahltenia

# Osnova rozprávania o transportnej vrstve

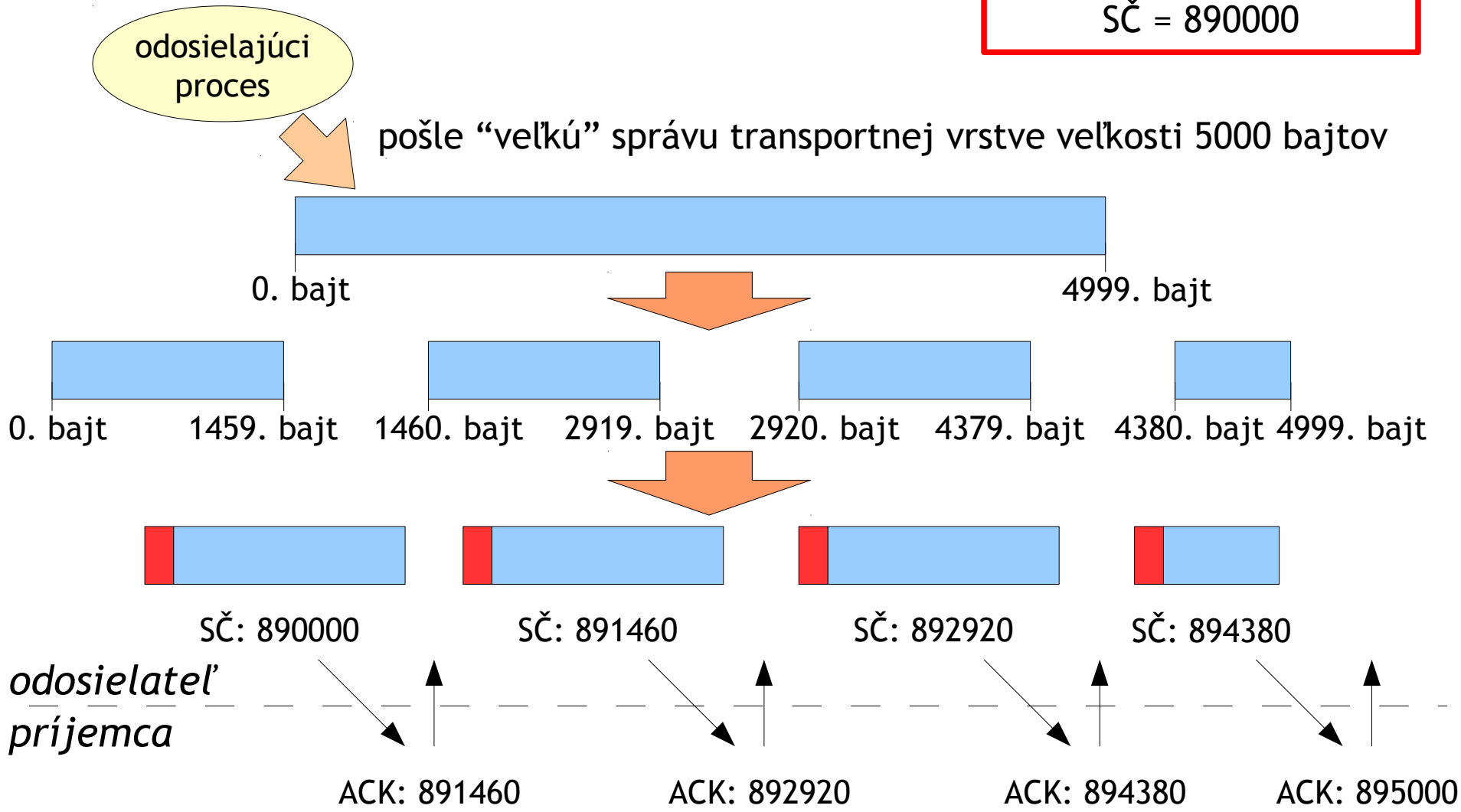
- 3.1 Služby transportnej vrstvy
- 3.2 Delenie správ a adresácia soketov
- 3.3 UDP: bezstavový transportný protokol
- 3.4 Princípy potvrdzovaného toku dát
- 3.5 TCP: stavový transportný protokol
  - štruktúra segmentu
  - pripájanie a odpájanie
  - potvrdzovaný tok dát
  - kontrola toku dát
- 3.6 Princípy zabezpečenia kontroly zahltenia
- 3.7 Kontrola zahltenia v protokole TCP

# Štruktúra TCP segmentu



# TCP sekvenčné čísla a potvrdenia

Vygenerované sekvenčné číslo = 889999, Prvé dátové SČ = 890000



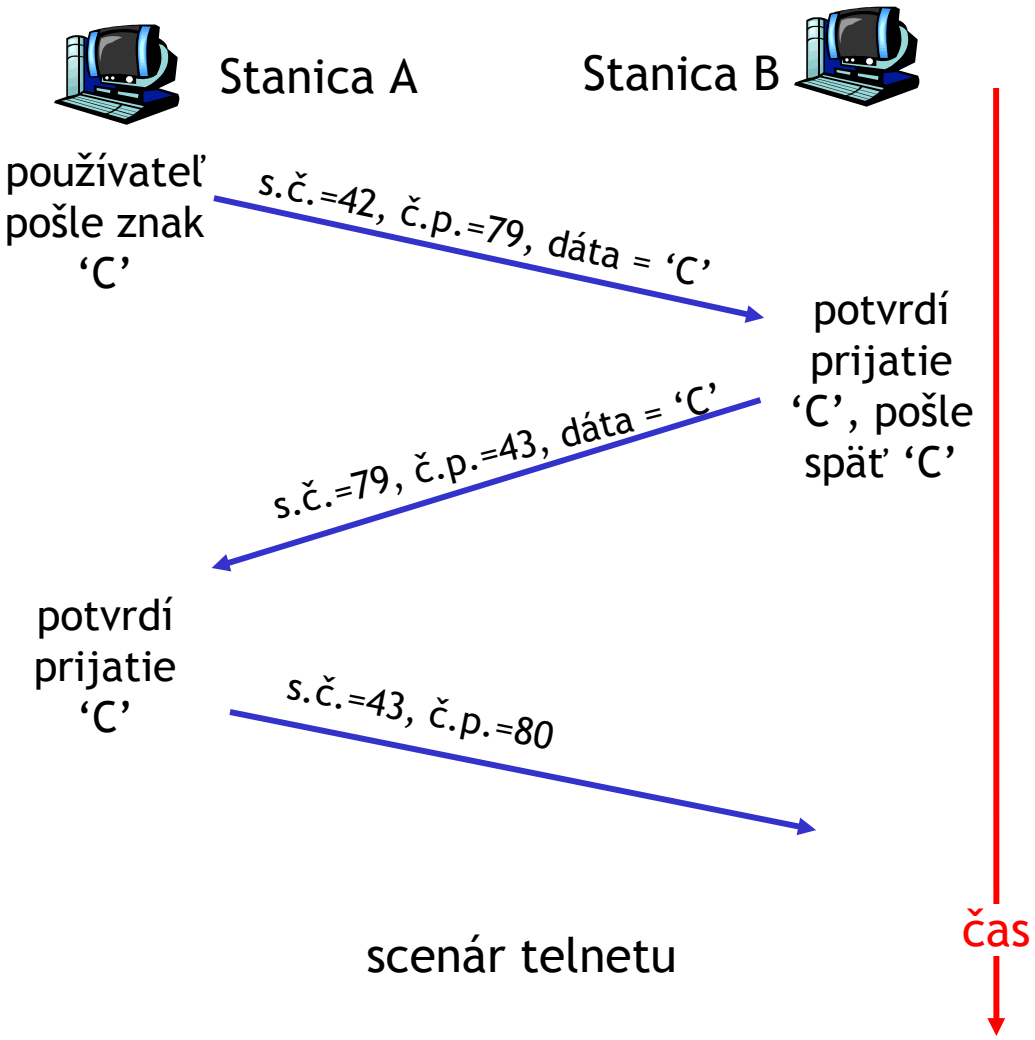
# TCP sekvenčné čísla a potvrdenia

Sekvenčné čísla:

- poradové číslo prvého dátového bajtu v segmente

Čísla potvrdenia (ACK):

- sekvenčné číslo dátového bajtu, ktorý je očakávaný ako ďalší v poradí na prijatie
- kumulatívne potvrdenie




scenár telnetu

# TCP: Manažment napojenia

- pred výmenou dát sa musí inicializovať spojenie
- úvodné nastavenie premenných:
  - sekvenčné číslo - náhodne
  - buffer, premenné kontroly toku dát (napr. **`RcvWindow`**)
- server: čaká na klienta

```
ServerSocket serverSocket = new ServerSocket(port);
Socket connectionSocket = serverSocket.accept();
```

- klient: iniciátor spojenia

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

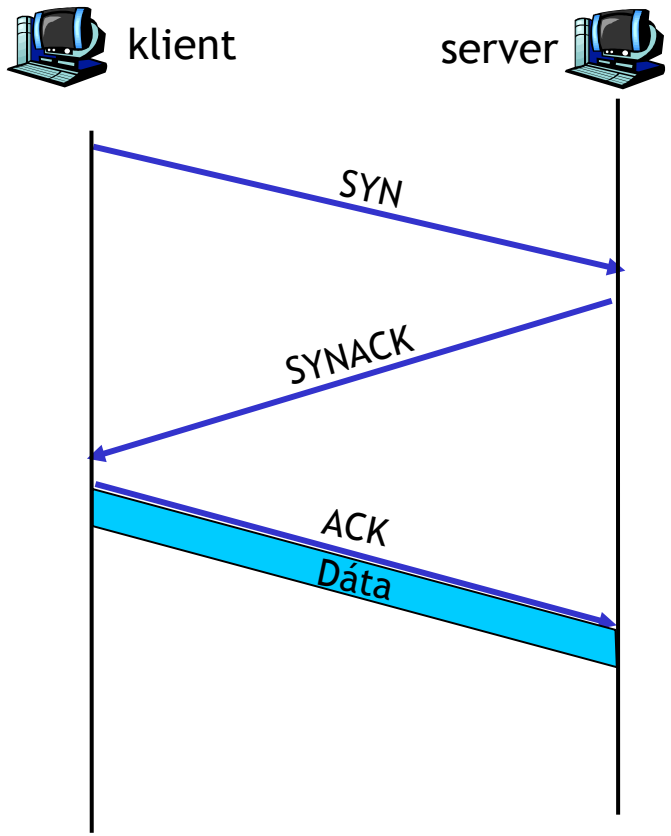
# TCP: Manažment napojenia

## Napojenie (handshake):

krok 1: klient pošle SYN segment na server
- vygeneruje svoje úvodné sekvenčné číslo
- žiadne dáta
- klient alokuje buffer

krok 2: server prijme SYN, odpovie SYNACK segmentom
- vygeneruje svoje úvodné sekvenčné číslo
- žiadne dáta
- server alokuje buffer

krok 3: klient prijme SYNACK, odpovie bežným ACK segmentom, ktorý už môže obsahovať dáta




Po vytvorení spojenia už je z pohľadu TCP jedno, ktorý z nich je klient a ktorý je server

# TCP: Manažment odpojenia

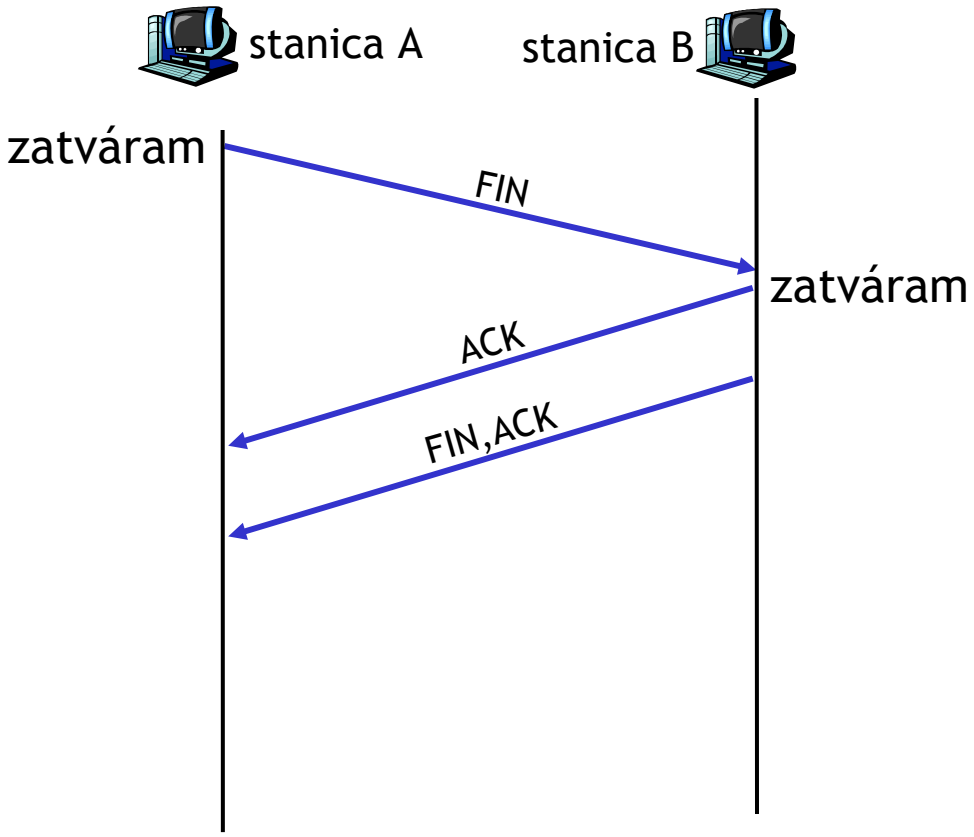
## Uzavretie spojenia:

jedna zo staníc chce ukončiť spojenie (pôvodný klient alebo server): `socket.close();`

krok 1: stanica A pošle FIN (alebo FINACK) segment stanici B

krok 2: stanica B prijme FIN, odpovie ACK segmentom.

krok 3: stanica B ak už nemá ďalšie dáta, pošle FINACK segment stanici A.

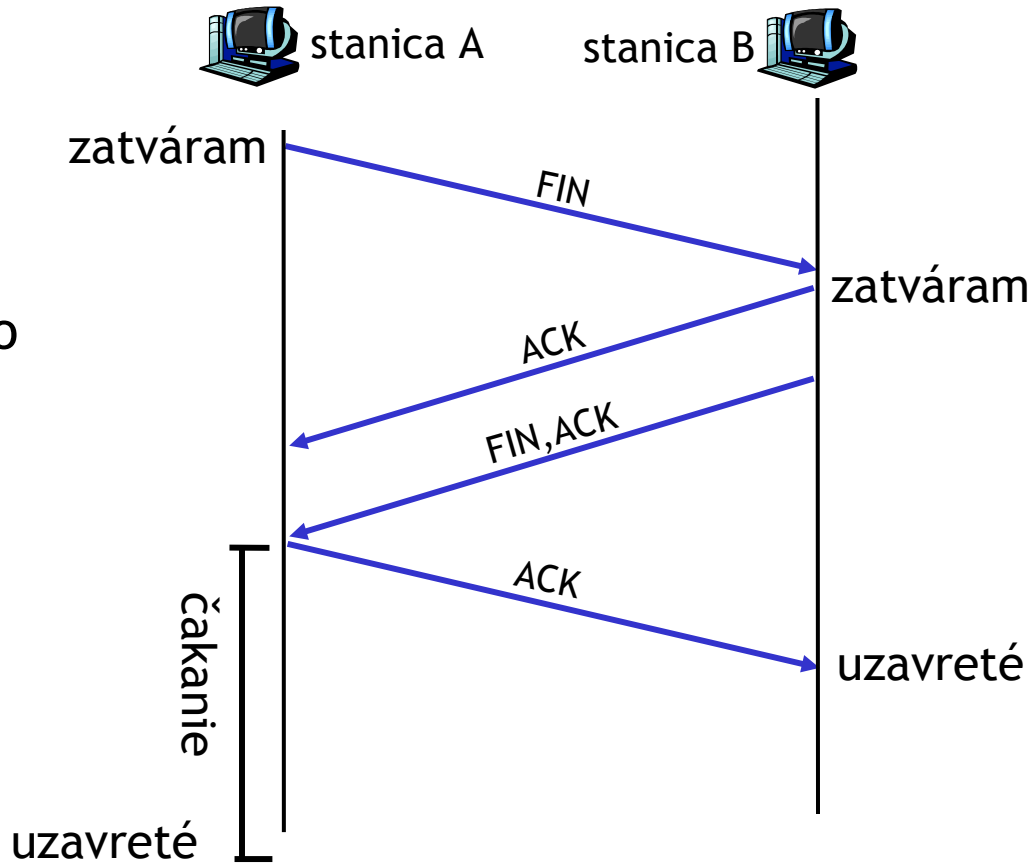


# TCP: Manažment odpojenia

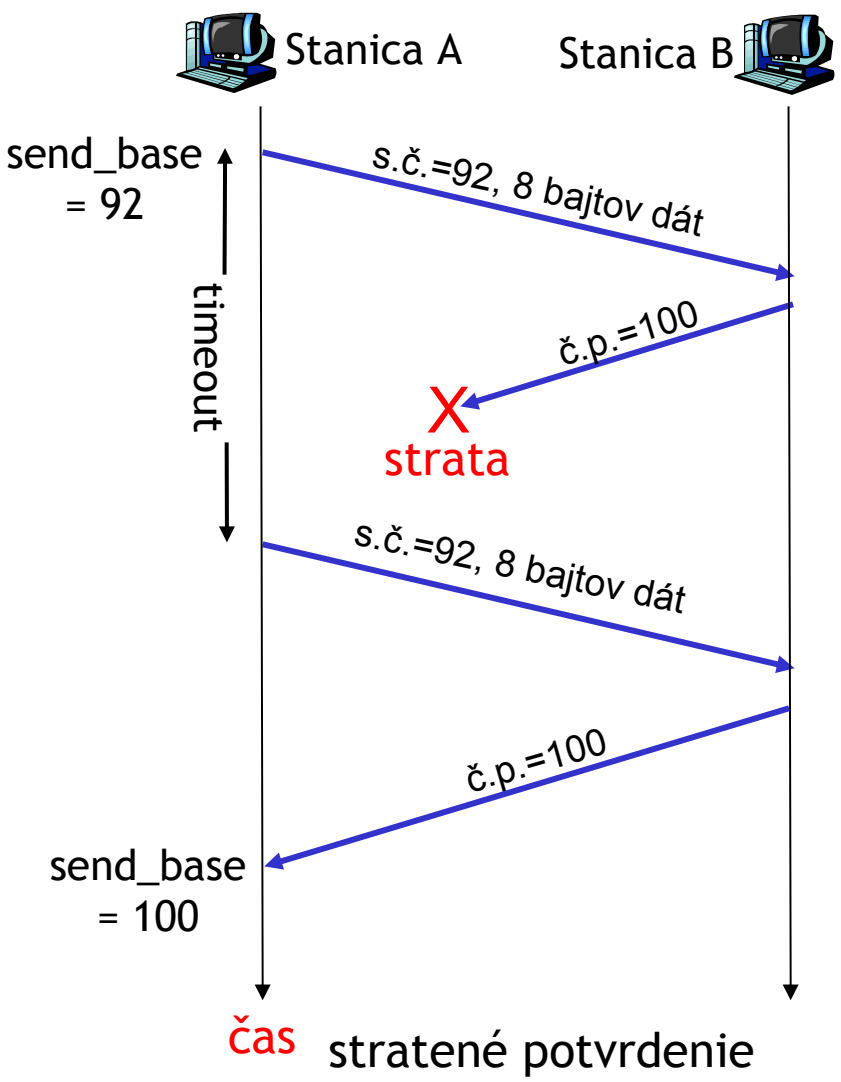
krok 4: stanica A prijme FINACK, odpovie ACK segmentom.

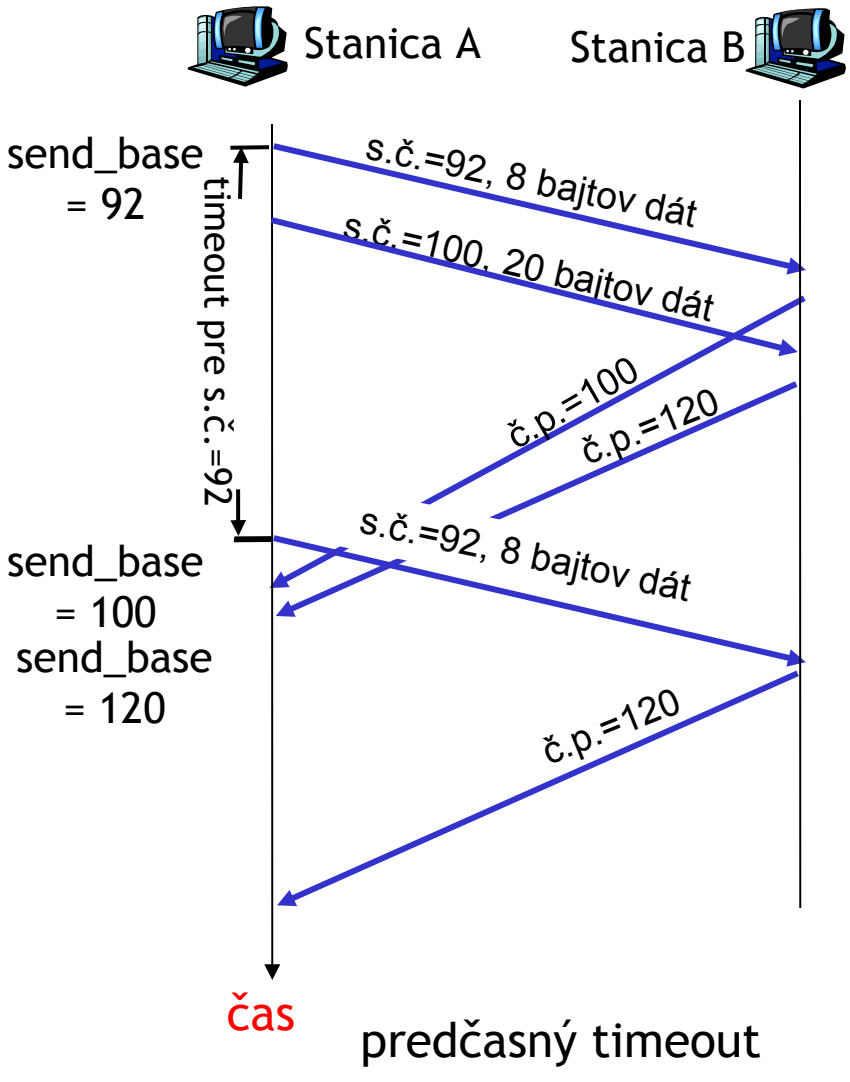
- začne čakať, či nepríde ďalší FINACK (ak sa jeho posledný ACK stratil)
- obvykle sa čaká 30 s, 1 min alebo 2 min

krok 5: stanica B prijme ACK. Uzatvorenie spojenia.

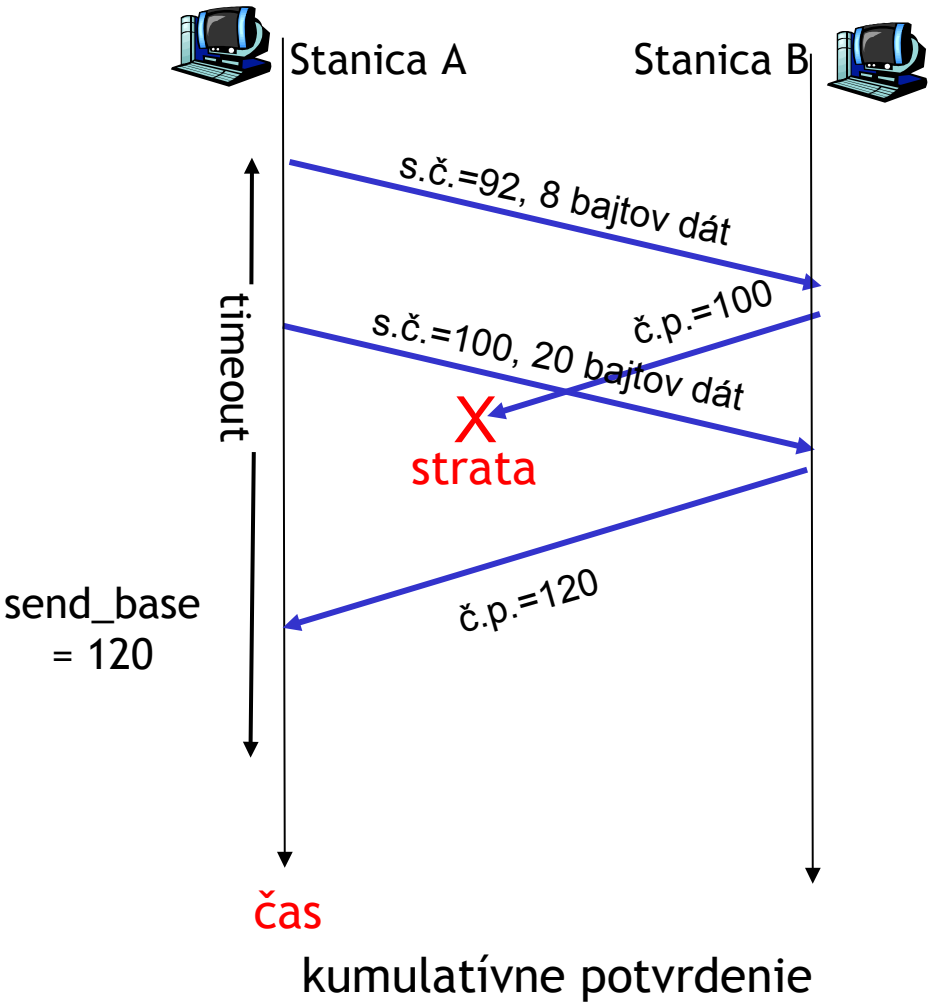


# TCP: príklad komunikácie



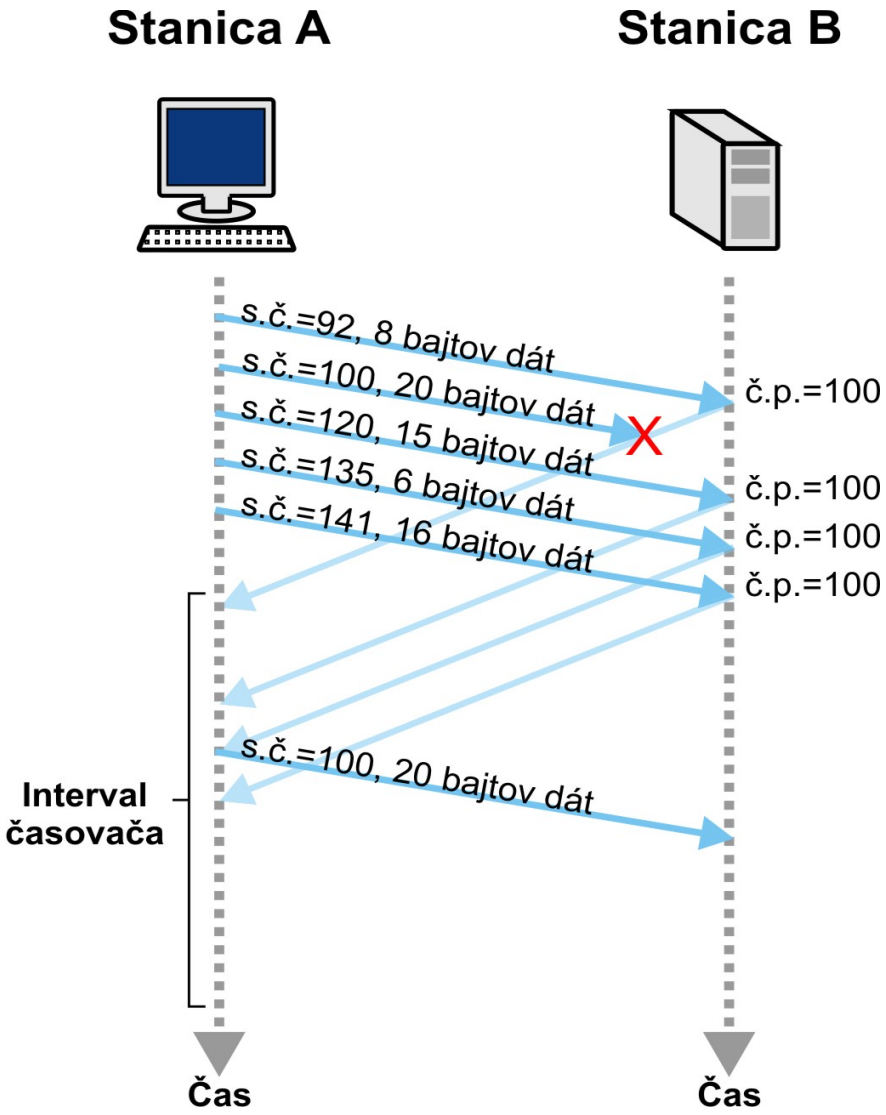



# TCP: príklad komunikácie




kumulatívne potvrdenie

# TCP: viacnásobné potvrdenie



# Udalosti odosielateľa:

došli dáta aplikačnej vrstvy:

- Vytvor segment so sekvenčným číslom *nextSeqNum*
  - nastav *nextSeqNum* na svoje sekvenčné číslo plus veľkosť dát v svojom segmente
- Ak bolo okno odosielateľa prázdne, zapni časovač
- expiračný interval: *TimeOutInterval*

timeout:

- prepošli najstarší nepotvrdený segment
- reštartuj časovač

prišlo potvrdenie:

- ak sa týka doteraz nepotvrdených segmentov (číslo potvrdenia > send_base)
  - nastav segment s dôjdeným číslom potvrdenia a všetky od neho staršie ako potvrdené a odstráň ich z okna odosielateľa
  - ak sú ešte nepotvrdené segmenty, zapni časovač
- ak prišli 3 potvrdenia s rovnakým číslom, prepošli najstarší nepotvrdený segment znova

# TCP: generovanie potvrdení

[RFC 1122, RFC 2581]

| udalosť TCP príjemcu | akcia |
|---|---|
| Príchod segmentu s očakávaným sekvenčným číslom, všetky dáta s menším číslom už potvrdené. | Nastav oneskorené potvrdenie. Čakaj max. 500 ms na ďalší segment. Ak nedôjde, odošli toto potvrdenie. |
| Príchod segmentu s očakávaným sekvenčným číslom. Predchádzajúci segment nebol potvrdený. | Odošli kumulatívne potvrdenie za oba segmenty. |
| Príchod segmentu mimo poradie s vyšším ako očakávaným sekvenčným číslom. Zistená diera ! | Odošli (zdupľované) potvrdenie s číslom potvrdenia najstaršieho neprijatého segmentu. |
| Príchod segmentu, ktorý čiastočne alebo úplne vypĺňa dieru. | Odošli potvrdenie s číslom potvrdenia najstaršieho neprijatého segmentu. |

# Rýchle preposlanie

- Timeout nastáva až po relatívne dlhom čase:
  - dlhá prestávka pred preposlaním strateného paketu
- Zistenie stratených segmentov cez viacnásobné potvrdenie
  - Odosielateľ odošle mnoho dát, pokiaľ sa nezistí strata
  - Ak sa segment stratí, začne prichádzať veľa rovnakých potvrdení
- Ak odosielateľ dostane 3 potvrdenia s rovnakým číslom, predpokladá, že dáta sa stratili:
  - rýchle preposlanie: pošlem dáta znova bez toho, aby som čakal na timeout a reštartujem časovač

# Round Trip Time (RTT) a timeout

Aká je správna hodnota pre interval timeoutu časovača?

- viac ako RTT
  - ale RTT je vždy iné
- ak je malá: predčasné opätovné poslania
  - zbytočné preposielania
- ak je veľká: pomalá reakcia na stratu segmentu

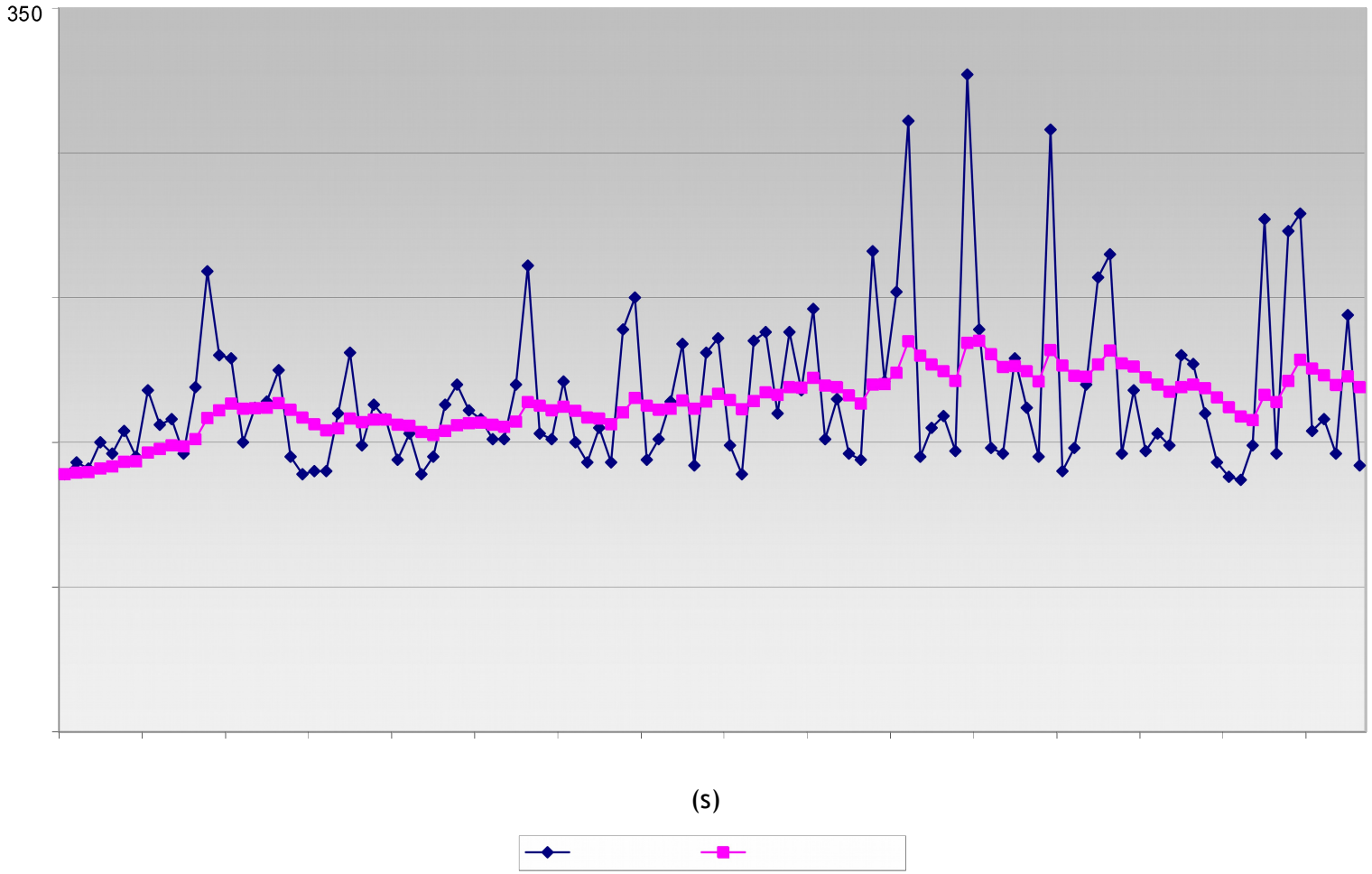
- **`SampleRTT:`** čas medzi odoslaním segmentu a príchodom jeho potvrdzovacieho segmentu
  - ignorujeme preposielania
- **`SampleRTT`** sa dosť mení, potrebujeme niečo stabilnejšie
  - vezmeme do úvahy niekoľko posledných hodnôt **`SampleRTT`**

# Očakávané RTT (EstimatedRTT)

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

- Predchádzajúca hodnota **`EstimatedRTT`** ovplyvňuje novú
- Exponential weighted moving average (exponenciálny vážený pohybujúci sa priemer ?)
- Vplyv posledného merania je oveľa menší ako vplyv pôvodnej hodnoty **`EstimatedRTT`**
- typicky $\alpha = 0.125$

# Príklad vývoja `SampleRTT` a `EstimatedRTT` v reálnom príklade



# Nastavenie timeout intervalu

- **`EstimatedRTT`** plus "ochranný pás"
  - veľká zmena **`SampleRTT`** -> väčší ochranný pás
- najprv vyjadrime, ako sa **`SampleRTT`** odchyľuje od **`EstimatedRTT`**:

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

typicky $\beta = 0.25$

TimeoutInterval nastavíme nasledovne:

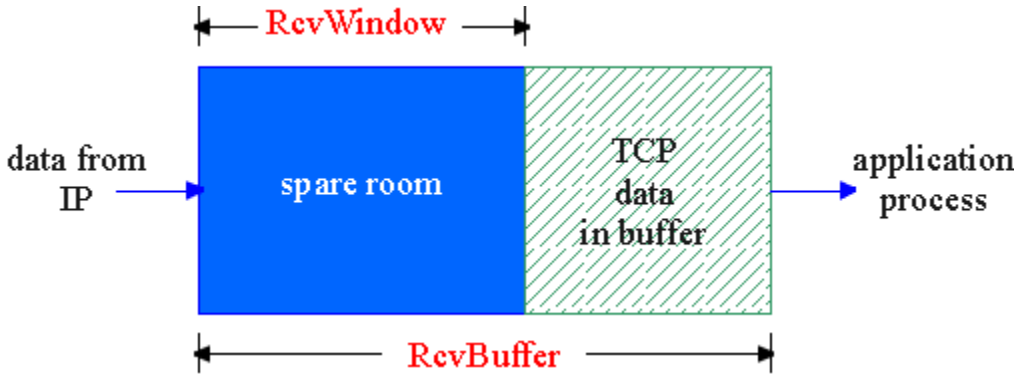
$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

# Osnova rozprávania o transportnej vrstve

- 3.1 Služby transportnej vrstvy
- 3.2 Delenie správ a adresácia soketov
- 3.3 UDP: bezstavový transportný protokol
- 3.4 Princípy potvrdzovaného toku dát
- 3.5 TCP: stavový transportný protokol
  - štruktúra segmentu
  - pripájanie a odpájanie
  - potvrdzovaný tok dát
  - kontrola toku dát
- 3.6 Princípy zabezpečenia kontroly zahltenia
- 3.7 Kontrola zahltenia v protokole TCP

# TCP: Kontrola toku dát

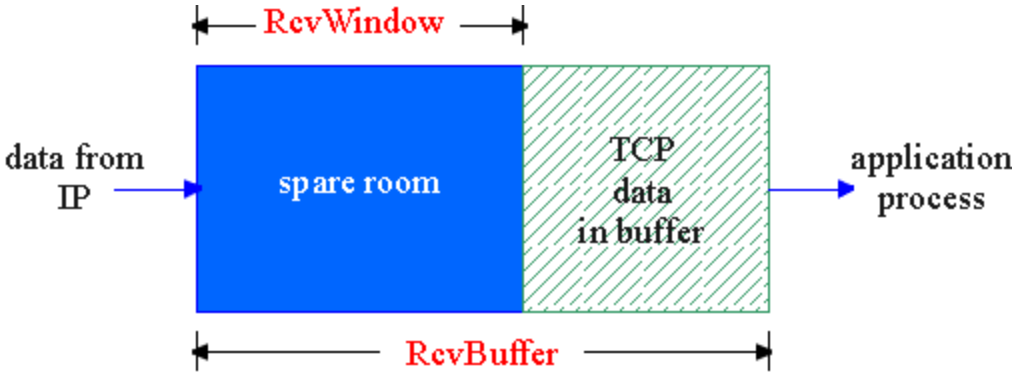
- na prijaté dáta máme
  buffer = okno príjemcu
  + nespracované dáta:




- proces aplikácie nemusí stíhať čítať z buffra

- odosielateľ nechce zahltiť príjemcu rýchlym posielaním správ
- snažíme sa odosielať takou rýchlosťou, akou to stíha príjemca spracúvať

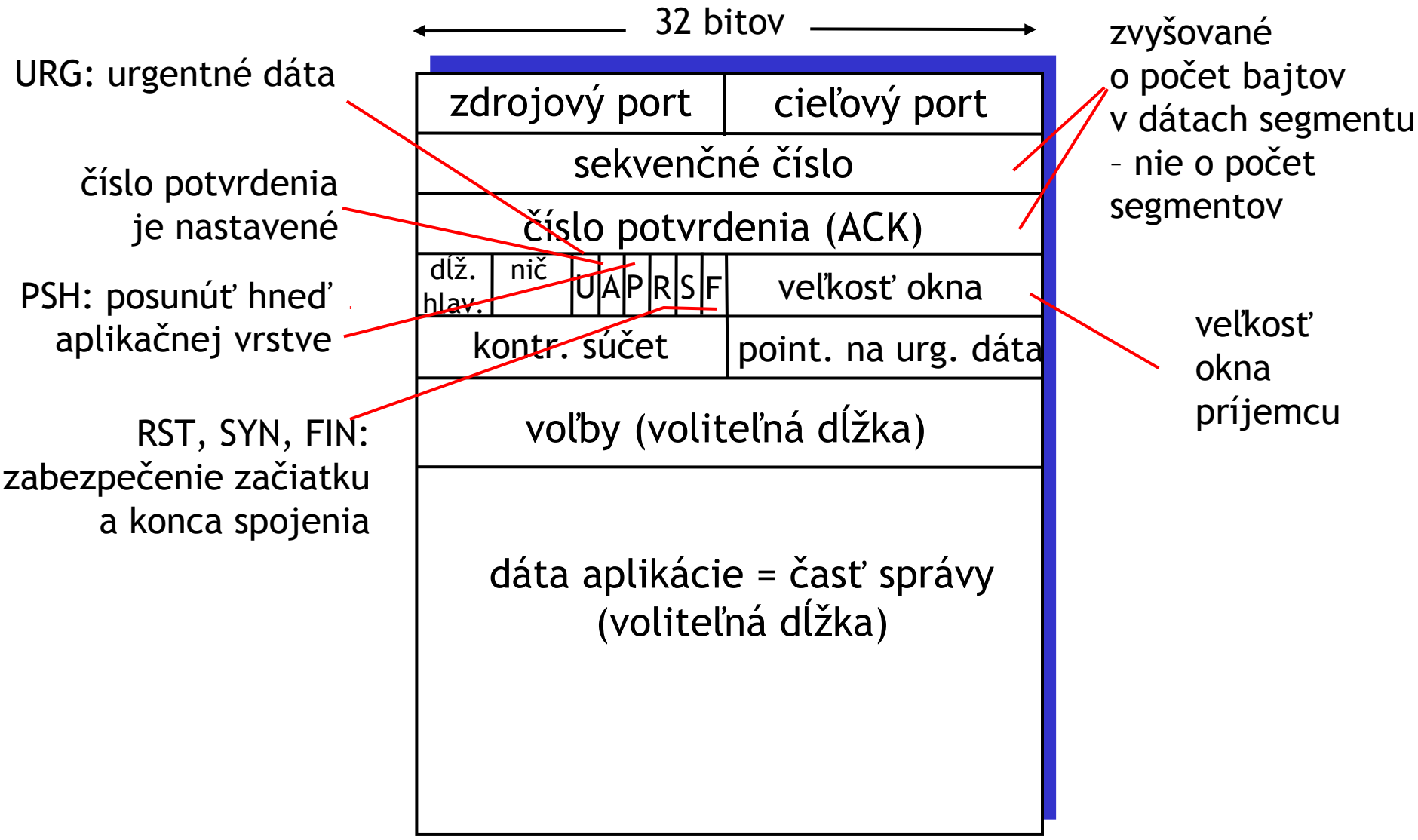
# TCP: Kontrola toku dát




- voľné miesto v buffri príjemcu
  - = **`RcvWindow`**
  - = **`RcvBuffer`** - dáta nespracované aplikáciou

- Príjemca informuje odosielateľa o veľkosti **`RcvWindow`** v hlavičke segmentu
- Odosielateľ nastavuje veľkosť svojho okna podľa hodnoty **`RcvWindow`**
  - iba toľko nepotvrdených segmentov môže mať odoslaných
  - garantujeme, že príjemcu nezahltíme

# Štruktúra TCP segmentu



# Osnova rozprávania o transportnej vrstve

- 3.1 Služby transportnej vrstvy
- 3.2 Delenie správ a adresácia soketov
- 3.3 UDP: bezstavový transportný protokol
- 3.4 Princípy potvrdzovaného toku dát
- 3.5 TCP: stavový transportný protokol
  - štruktúra segmentu
  - pripájanie a odpájanie
  - potvrdzovaný tok dát
  - kontrola toku dát
- 3.6 Princípy zabezpečenia kontroly zahltenia
- 3.7 Kontrola zahltenia v protokole TCP

# Princípy zabezpečenia kontroly zahltenia

## Zahltenie:

- neformálne: “veľa zdrojov generuje tak veľa dát, že ich *sieť* nestíha všetky spracovať a preposielať k cieľom”
- rôzne od kontroly toku dát!
- prejavy:
  - stratené pakety (pretečenie buffrov na routroch)
  - veľké zdržania (čakanie v radoch paketov v routroch)

# Prístupy ku kontrole zahltenia

Dva hlavné prístupy:

## Kontrola zahltenia pozorovaním vlastnej TCP komunikácie:

- žiadne explicitné informácie z jadra siete
- zahltenie sa odhalí stratou a zdržaním segmentov
- podpora priamo v TCP protokole

## Kontrola zahltenia s asistenciou siete:

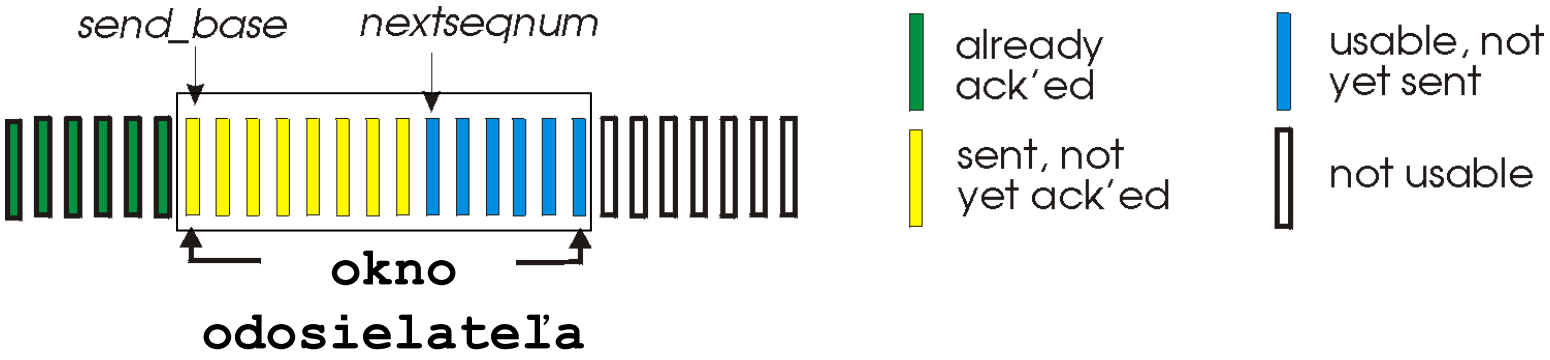
- routre poskytujú spätnú väzbu cieľovým staniciam
  - jednobitová informácia o zahltení
  - explicitná hodnota rýchlosti, ktorou má odosielateľ vysielať

# Osnova rozprávania o transportnej vrstve

- 3.1 Služby transportnej vrstvy
- 3.2 Delenie správ a adresácia soketov
- 3.3 UDP: bezstavový transportný protokol
- 3.4 Princípy potvrdzovaného toku dát
- 3.5 TCP: stavový transportný protokol
  - štruktúra segmentu
  - pripájanie a odpájanie
  - potvrdzovaný tok dát
  - kontrola toku dát
- 3.6 Princípy zabezpečenia kontroly zahltenia
- 3.7 Kontrola zahltenia v protokole TCP

# Kontrola zahltenia v TCP

- bez spolupráce so "sieťou", teda s routrami
- rýchlosť odosielania je limitovaná veľkosťou okna **`Congwin`** (congestion=zahltenie)




- Okno odosielateľa = min {**`Congwin, RcvWindow`**}

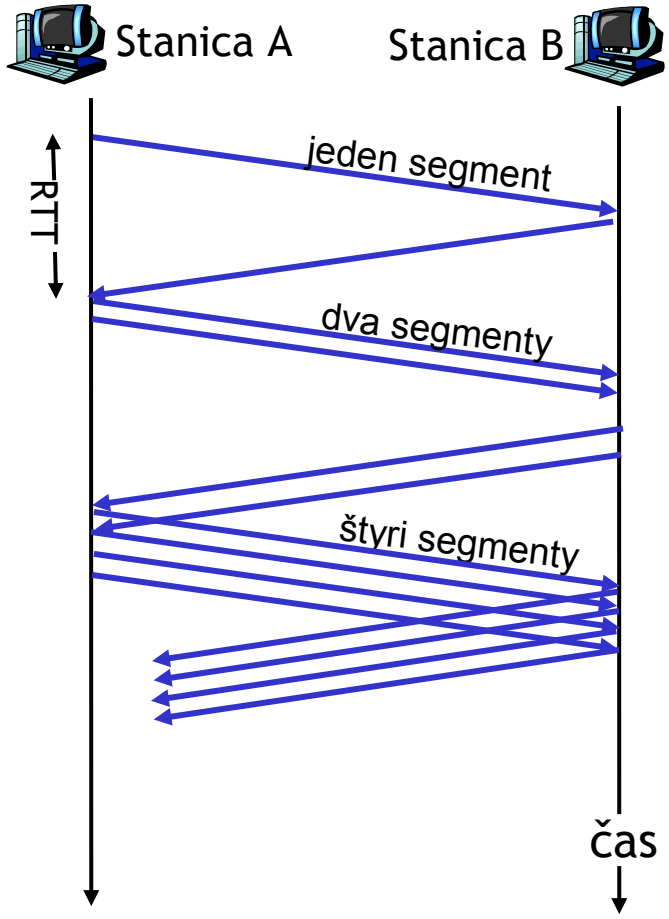
# Kontrola zahltenia v TCP

- “skúšanie” použiteľnej šírky pásma:
  - ideálne: odosielam tak rýchlo, ako mi umožňuje pripojenie (**`Congwin`** je také veľké ako treba), pričom nedochádza k stratám ani zdržaniu
  - *zvyšuj* hodnotu **`Congwin`**, pokiaľ nedôjde k strate (pokiaľ nezahlcujeme)
  - strata paketu: *zníž* hodnotu **`Congwin`**, a opäť skús zvyšovať
- dve “fázy”
  - pomalý štart (slow start)
  - predchádzanie zahlteniu (congestion avoidance)
- dôležité premenné:
  - **`Congwin`**
  - **`threshold`**: definuje hranicu “správania” pri ďalšom odosielaní po zistení straty paketu

# Pomalý štart (Slow Start)

## SlowStart algoritmus

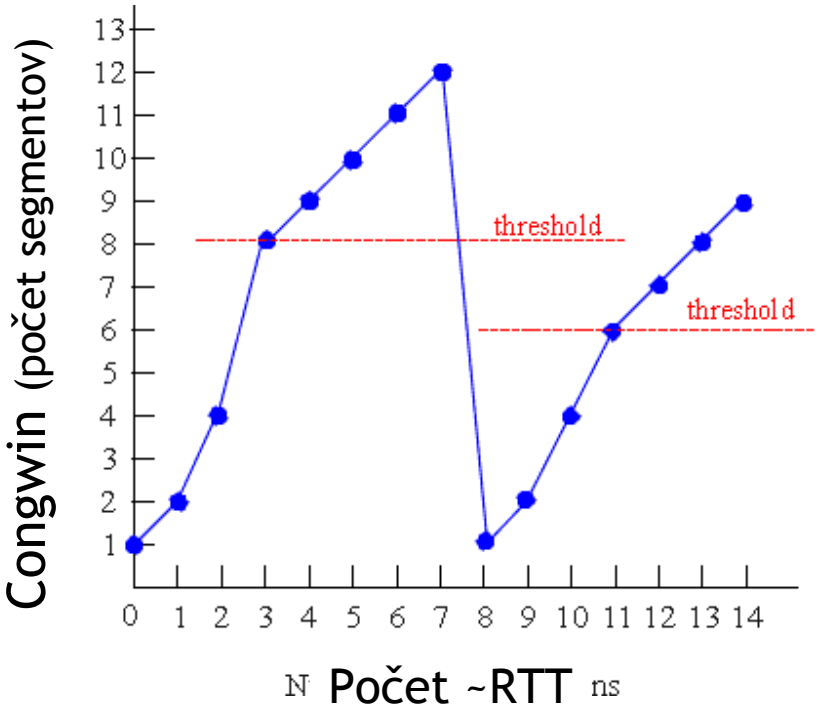
```
Congwin = 1
do {
    pre každý ACK segment bez
       duplicít Congwin++
} while (! strata paketu AND
            !(CongWin > threshold))
```




- exponenciálny nárast veľkosti okna s každým ~RTT (t.j. potvrdením celého predchádzajúceho okna)

# Predchádzanie zahlteniu: TCP Tahoe

```
/* SlowStart skončil      */
/* Congwin > threshold */
while (nie je strata počas ~RTT) {
   Congwin++
}
threshold = Congwin/2
Congwin = 1
začni SlowStart
```



# TCP AIMD = additive increase, multiplicative decrease

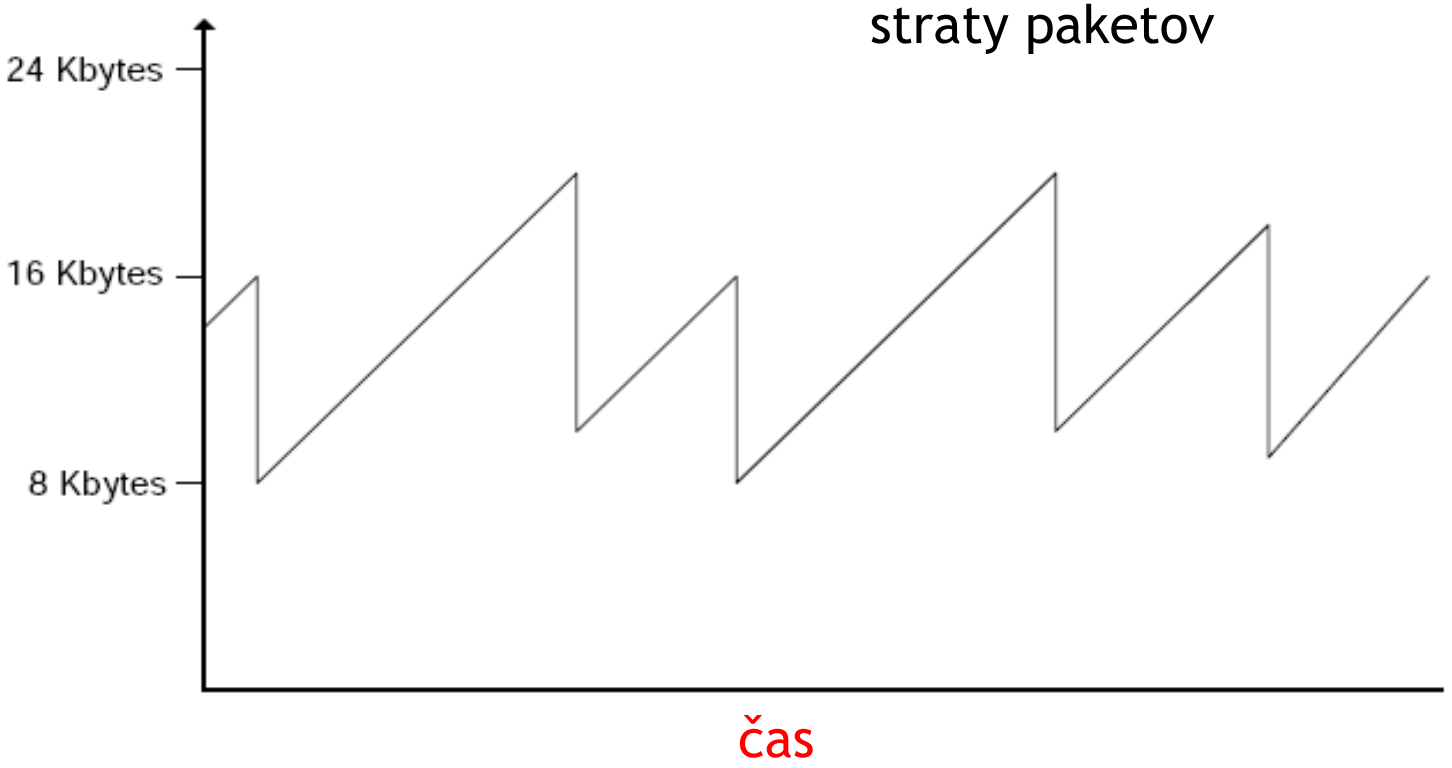
zmenšovanie delením:

ignorujeme SlowStart =>

**`CongWin = CongWin/2`**

zväčšovanie pripočítavaním:

zväčšíme **CongWin** o 1 maximálnu veľkosť segmentu (MSS) po každom ~RTT bez straty paketov



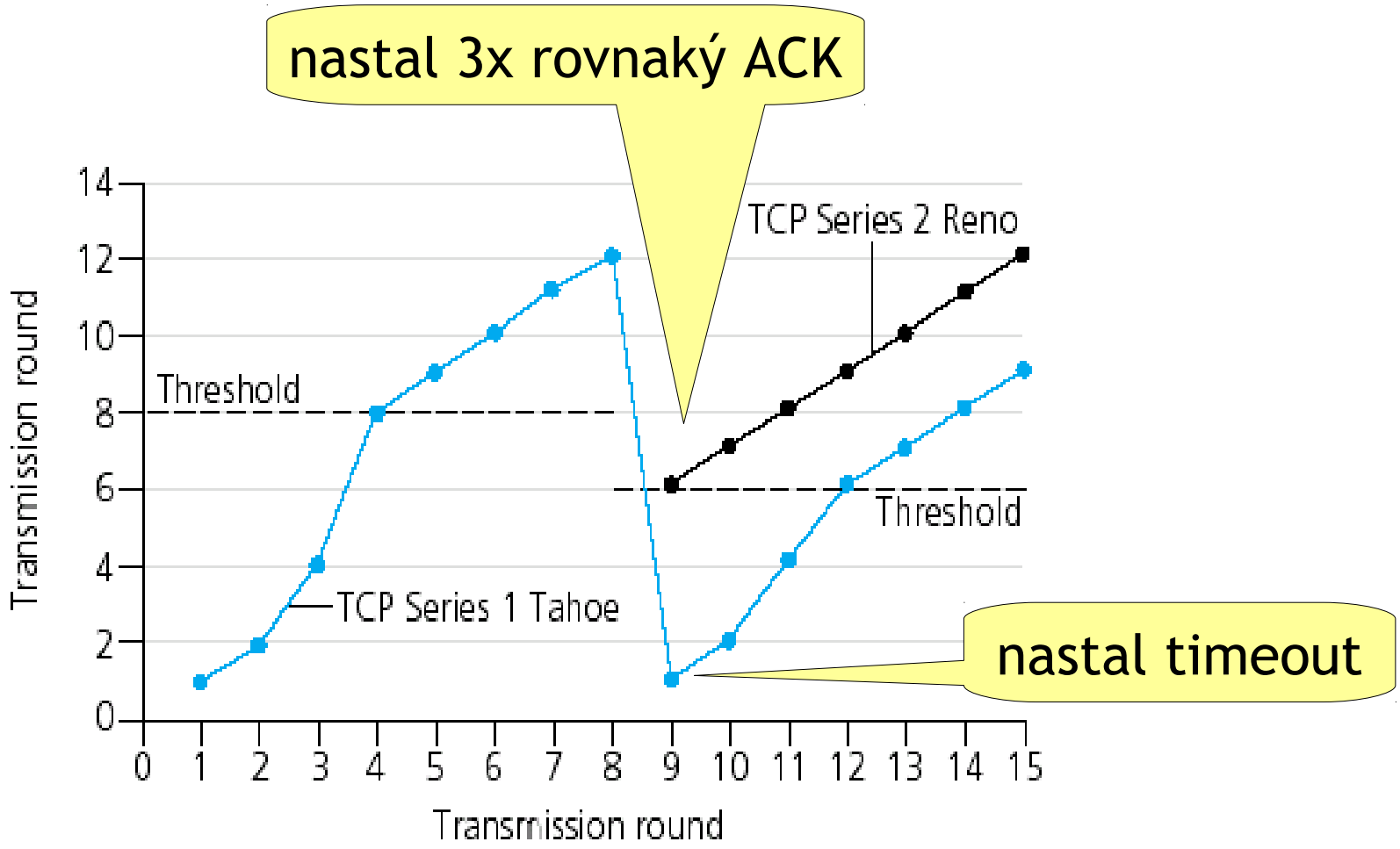
# Vylepšenie: TCP Reno

- Po troch 3 rovnakých potvrdeniach:
  - veľkosť **`CongWin`** sa delí na polovicu
  - následne rastie lineárne
- ALE ak nastal timeout:
  - **`CongWin`** sa nastaví na veľkosť 1 segmentu (1 MSS)
  - začne SlowStart, ktorý beží až do thresholdu

Filozofia:

- 3 rovnaké potvrdenia hovoria, že sieť je schopná prenášať nejaké segmenty
- timeout bez toho, aby nastali 3 rovnaké potvrdenia je "vážnejšia udalosť"

# Vylepšenie: TCP Reno



# Zhrnutie: kontrola zahltenia v TCP

- Ak **`CongWin`** je menšie ako **`threshold`**, odosielateľ je vo fáze slowStart, okno rastie exponenciálne.
- Ak **`CongWin`** je väčšie ako **`threshold`**, odosielateľ je vo fáze ”predchádzanie zahlteniu”, okno rastie lineárne, ale:
- Ak príde 3x rovnaký ACK, **`threshold`** aj **`CongWin`** sa nastaví na **`CongWin/2`**
- Ak nastane timeout, **`threshold`** sa nastaví na **`CongWin/2`** a **`CongWin`** sa nastaví na 1 MSS.

# Priepustnosť TCP

- určme priemernú priepustnosť TCP ako funkciu veľkosti okna a ~RTT
  - ignorujme SlowStart
- označme W ako veľkosť okna pri odhalení straty.
- keď veľkosť okna je w, priepustnosť je w/~RTT
- hneď po strate sa w zmenší na W/2, priepustnosť na W/(2*~RTT).
- predpokladajme, že W sa nemení, potom priemerná priepustnosť je 0.75 W/~RTT

# Budúcnosť TCP: “dlhé, široké rúry”

- Príklad: 1500 bytové segmenty, 100ms RTT, chceme priepustnosť 10 Gb/s
- Potrebujeme veľkosť okna W = 83 333 MSS
- Vzorec pre priepustnosť ako funkcia frekvencie strát (L):

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

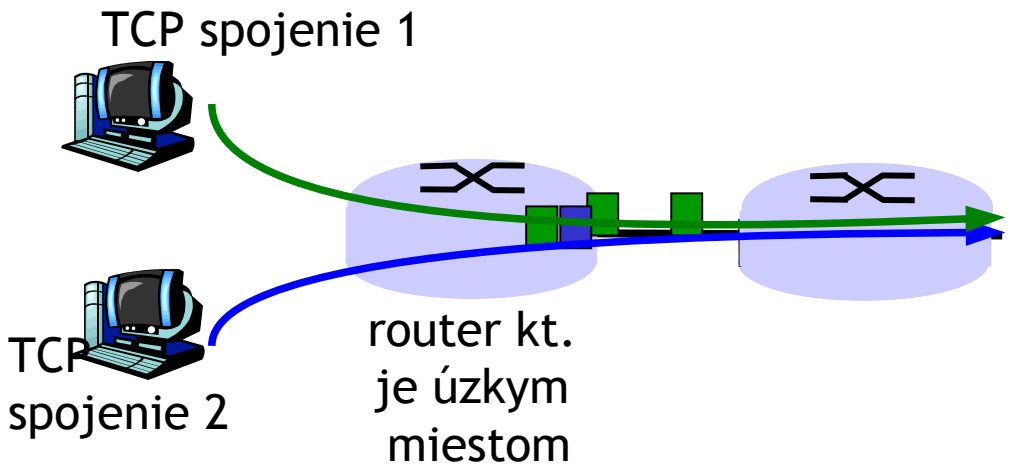
- ➜ L = $2 \cdot 10^{-10}$ *t.j. 1 strata po* $4{,}666 \cdot 10^{9}$ *segmentoch*
- Vývoj nových TCP algoritmov na kontrolu zahltenia pre veľké rýchlosti a vzdialenosti

# Užitočné voľby v TCP hlavičke

- Hlavička obsahuje miesto pre ďalšie voľby
- Štandardná veľkosť MSS je 536 bajtov. Bežne sa však používa 1460 bajtov.
  - Počas handshaku vieme nastaviť premennú MSS na vyššiu alebo nižšiu
- Maximálne číslo v TCP hlavičke pre veľkosť okna príjemcu je 65535 bajtov
  - pri veľkosti MSS 1460 B máme iba okno iba do 44 MSS !
  - Počas handshaku vieme nastaviť premennú WSCALE, ktorou vykonáme bitový posun čísla veľkosti okna v hlavičke.
    - Určíme tak násobok dvojky, ktorým sa bude násobiť hodnota veľkosti okna v hlavičke počas celej komunikácie → okno môže mať pokojne až 1 GB

# Spravodlivosť TCP

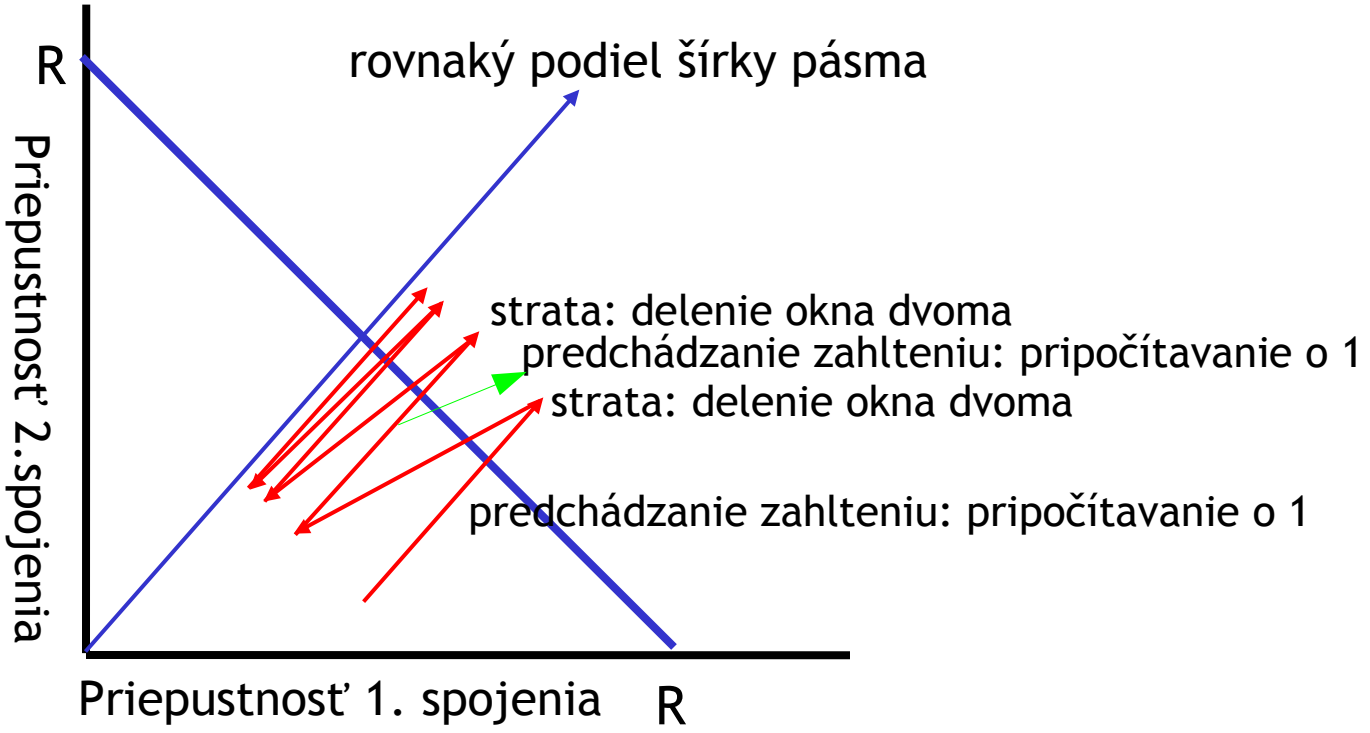
Cieľ spravodlivosti: ak *k* TCP spojení zdieľa rovnaké úzke miesto s prenosovou rýchlosťou *R*, každý by mal mať priemernú rýchlosť spojenia *R*/*k*



# Prečo je TCP spravodlivé

Dve súťažiace spojenia:

- fáza “predchádzanie zahlteniu” zväčšuje okno o 1 u oboch
- pri strate je delenie dvoma proporčné (u pomalšieho menej)



# Spravodlivosť

## UDP:

- Multimediálne aplikácie často nepoužívajú TCP
  - rýchlosť sa pri zahltení neznižuje
  - posielajú dáta konštantnou rýchlosťou
  - nevadí im strata paketov
- Miesto pre výskum: urobiť UDP tak, aby bolo priateľské k TCP

## Paralelné TCP spojenia

- nič nebráni tomu, aby aplikácie otvorili viac paralelných spojení medzi 2 stanicami
- toto robia webové prehliadače / sťahovače
- príklad: cez úzke miesto prechádza práve 9 TCP spojení
  - moja nová aplikácia s 1 TCP spojením dosiahne rýchlosť R/10
  - ak si otvorím 9 TCP spojení, dosiahnem R/2 !

# Zhrnutie

- Princípy fungovania služieb transportnej vrstvy:
  - delenie správ a adresácia soketov
  - spoľahlivý prenos dát
  - kontrola toku dát
  - kontrola zahltenia
- fungovanie implementácií protokolov transportnej vrstvy na internete:
  - UDP
  - TCP

V ďalšej prednáške:

- opustíme okraje siete (aplikačná a transportná vrstva)
- pozrieme sa na jadro siete

# Ďakujem za pozornosť

Modifikované slajdy z knihy:

*Computer Networking: A Top Down Approach* ,
4th edition.
Jim Kurose, Keith Ross
Addison-Wesley, July 2007.