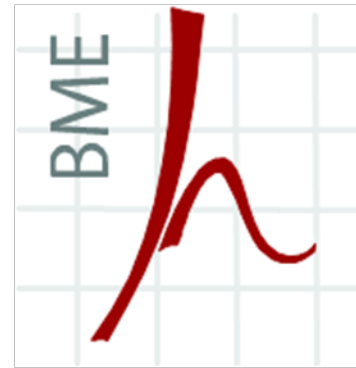




SZÁLLÍTÁSI (TRANSPORT, HOST-TO-HOST) PROTOKOLLOK

UDP és TCP

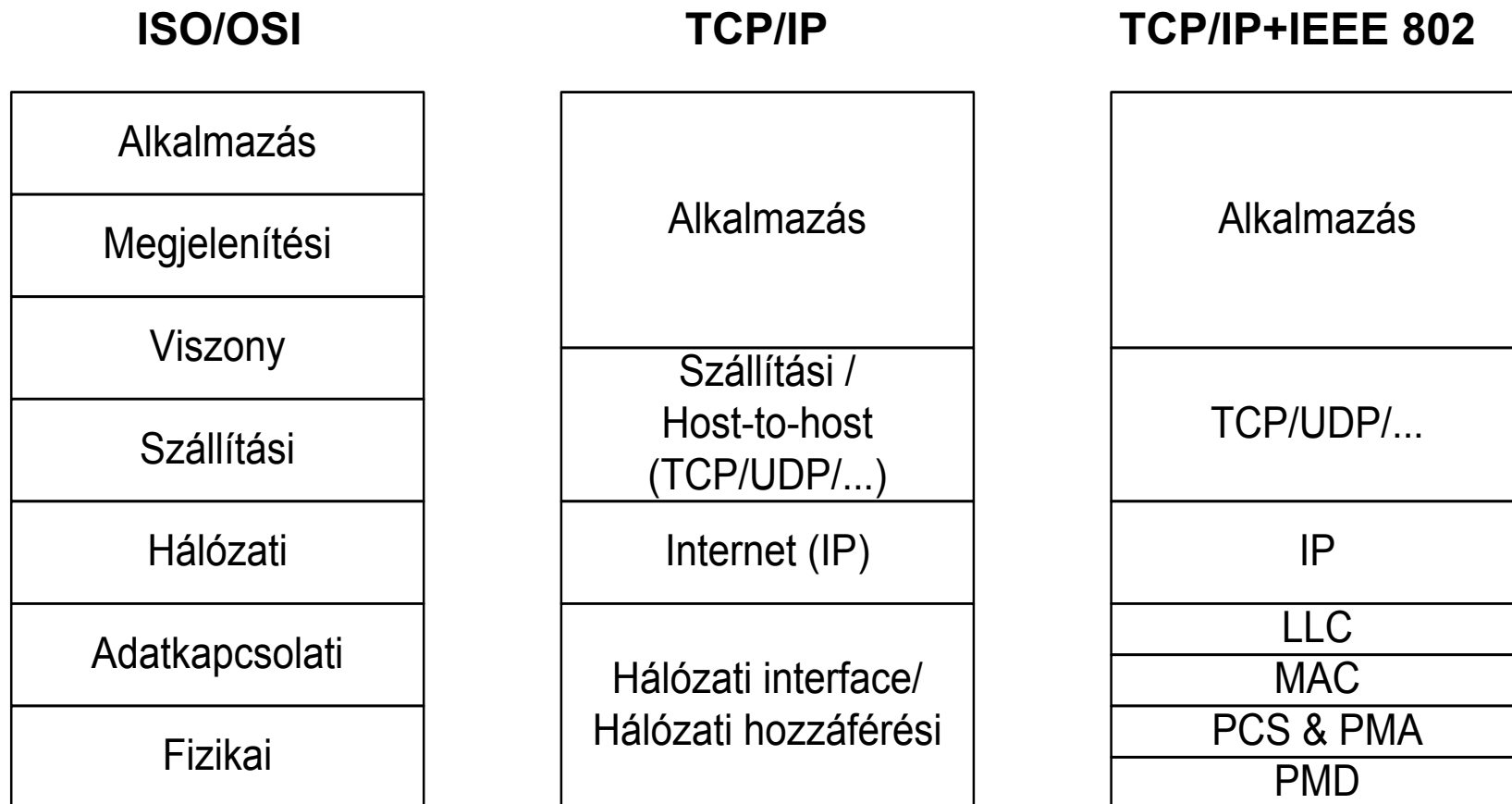
Dr. Simon Vilmos
docens

BME Hálózati Rendszerek és Szolgáltatások Tanszék

svilmos@hit.bme.hu

2016.november 16.

A TCP/IP architektúra és az ISO/OSI rétegmodell



IP: Internet Protocol

TCP: Transmission Control Protocol

UDP: User Datagram Protocol

LLC: Logical Link Control

MAC: Medium Access Control

PCS: Physical Coding Sublayer

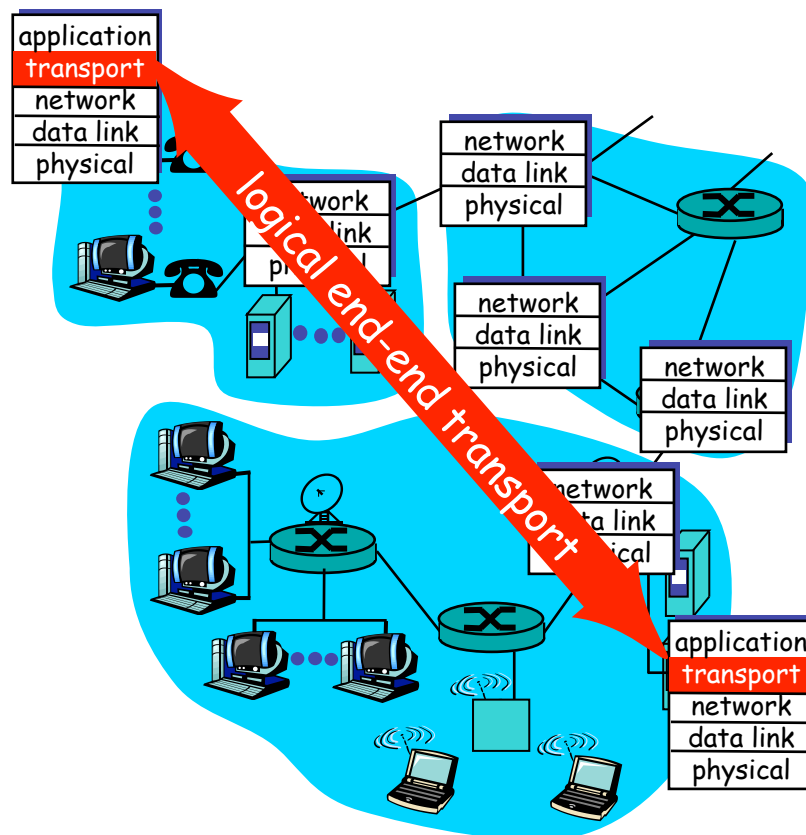
PMA: Physical Medium Attachment

PMD: Physical Medium Dependent

A hálózati és a szállítási réteg

- *hálózati réteg*: végpontok („host”-ok) közötti logikai kapcsolatok
- *szállítási réteg*: alkalmazások (process) közötti logikai kapcsolatok
 - a hálózati réteg szolgáltatásainak igénybevételére alapozva **megbízható** átvitel a **transzport entitások** között (transport entity)
 - Feladatai:
 - forgalomszabályozás
 - multiplexelés
 - hibadetektálás, javítás (pl. Automatic Repeat reQuest – ARQ: nyugta, timer, újraküldés)
 - sorrendhelyes átvitel

A szállítási réteg



Logikai kapcsolatok a
végpontokban futó
alkalmazások között

A szállítási protokollok a
végpontokban futnak, a
csomópontokban nem

Az alkalmazások
adategységeit szállítási
protokoll-adategységekbe
tördeljük, a kapottakból
pedig összerakjuk

Szállítási protokollok: UDP és TCP

- **UDP – User Datagram Protocol**
- **TCP – Transmission Control Protocol**

- Az UDP és TCP közös képességei:
 - Portok kezelése
 - Multiplexelési képesség

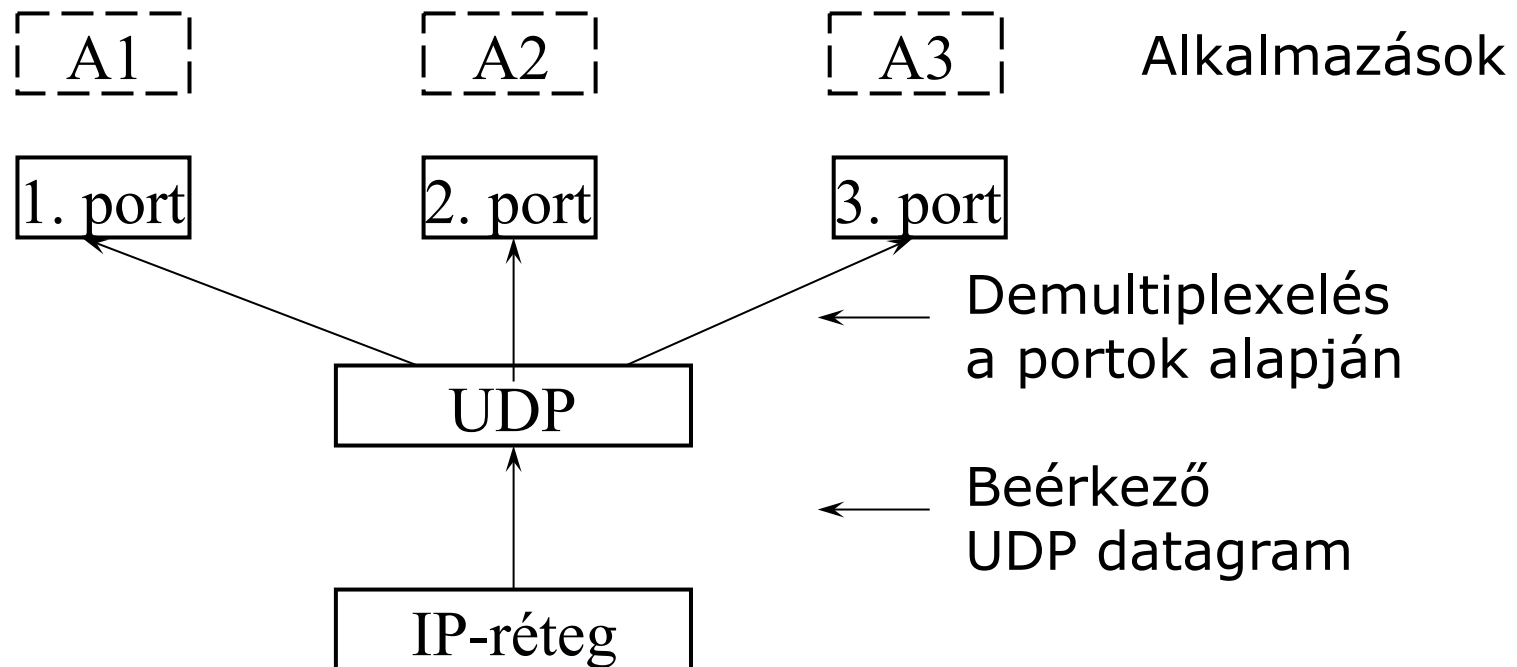
- Alapvető különbség az UDP és a TCP között:
 - o UDP összeköttetésmentes (connectionless),
 - o TCP összeköttetés-alapú (connection-oriented)transzport-szolgáltatást nyújt

Az UDP és TCP közös képességei (1)

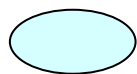
- Portok kezelése:
 - Az IP-rétegben a csomagok végpontnak, „host”-nak vannak címezve
 - A végpontokon belül: több alkalmazás, folyamat
 - Megkülönböztetésük: portok használatával (16 biten)
 - **Foglalt** (reserved, „well-known”, 0-1023), **regisztrált** (registered, 1024-49151) és **dinamikus** (dynamic, 49152-65535) portszámok
 - Foglalt portok: ide mindig lehet küldeni datagrammokat
 - pl. 80: HTTP, 21: FTP (ezek főként TCP-re)
 - Az UDP-en belül megállapításra kerülnek az alkalmazandó portszámok
- Multiplexelés /demultiplexelés
 - A portmechanizmus segítségével

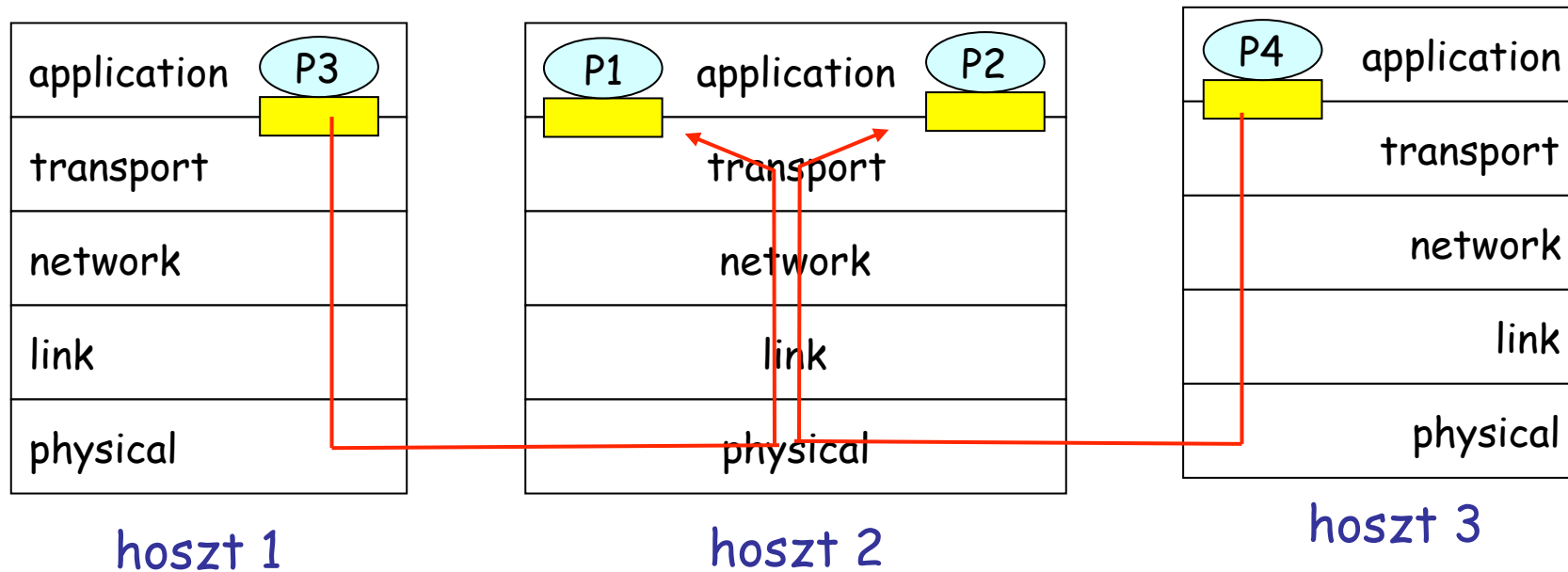
Az UDP és TCP közös képességei (2)

Multiplexelés és demultiplexelés
Példa:



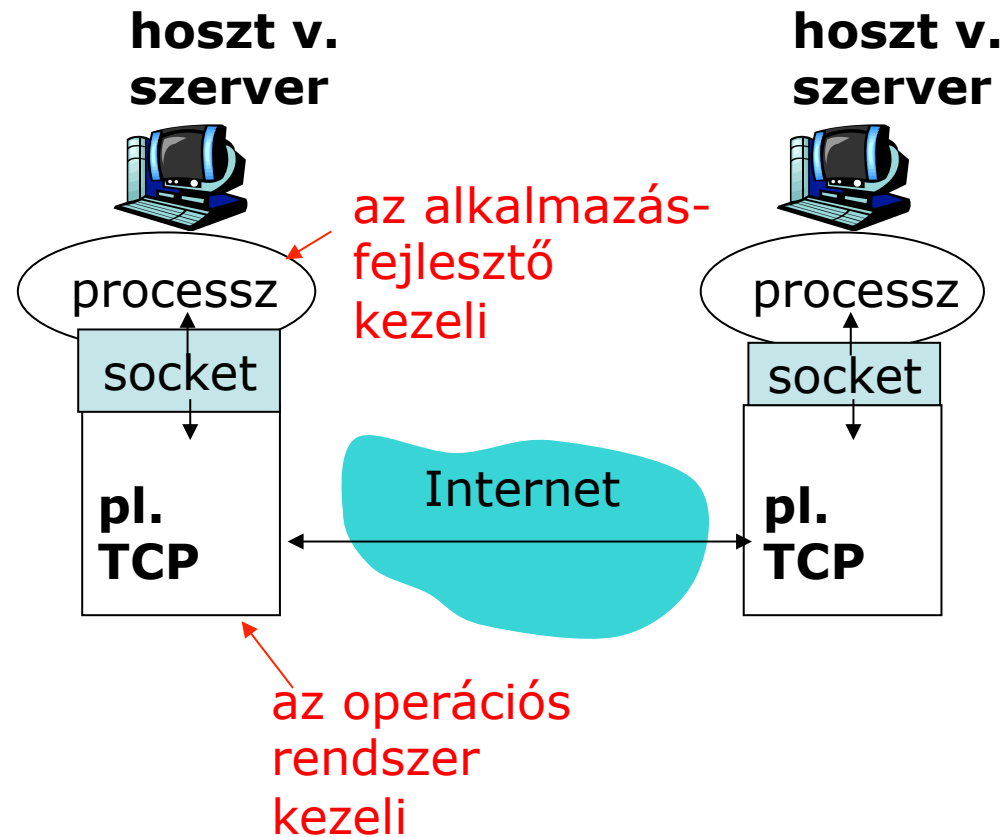
Multiplexelés-demultiplexelés

 = socket  = processz



- Socket: interfész, „ajtó” az alkalmazás és a hálózat között
 - A socket-et az alábbiak jellemzik:
 - transzport protokoll (UDP v. TCP)
 - saját IP cím
 - saját portszám
 - *(opcionális) távoli host IP címe*
 - *(opcionális) távoli alkalmazás portszáma*
- helyi socket cím
- távoli socket cím
- socket pár
- Leegyszerűsítve: socket = IP cím + portszám
 - Az operációs rendszer a bejövő IP csomagokat a fentiek alapján továbbítja az alkalmazásnak, kiszedve ezeket a megfelelő PDU-kból
 - Netstat paranccsal lehet kilistázni őket

Socket: illusztráció



Socket típusok

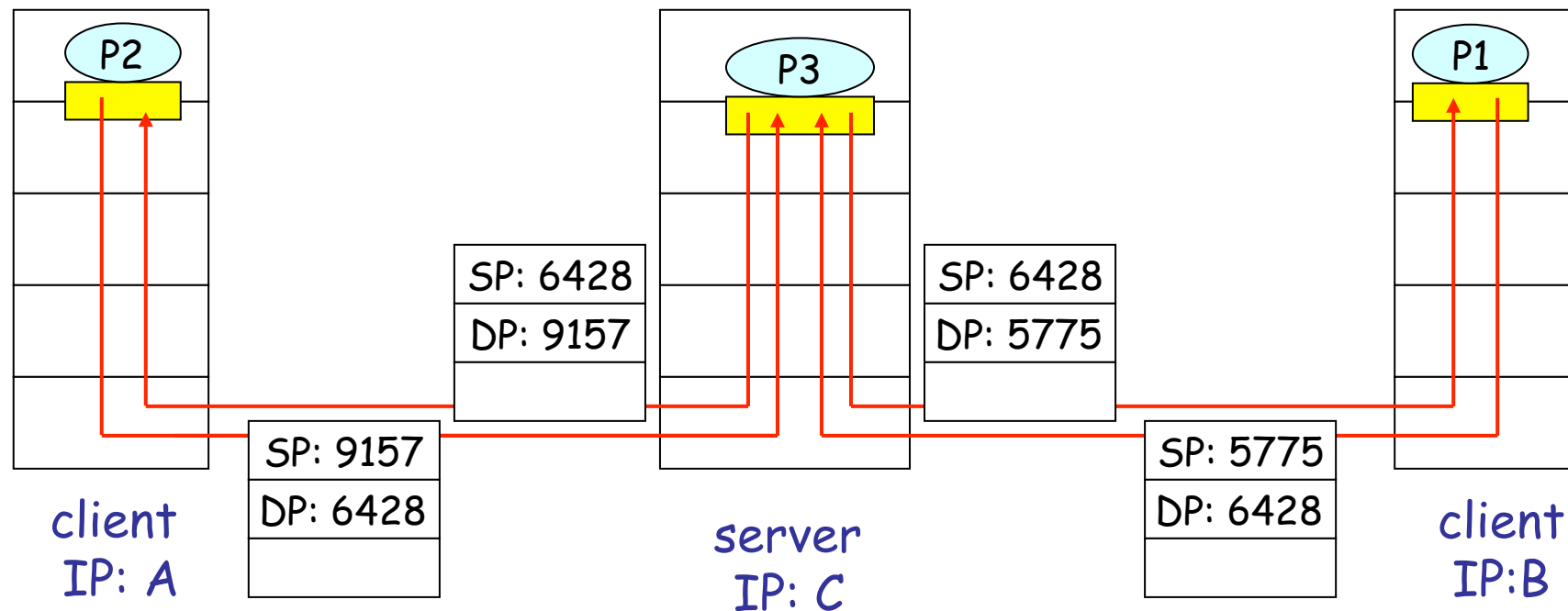
- Datagramm socket
 - Összeköttetésmentes socket, UDP használja

- Stream socket
 - Összeköttetés-alapú socket, TCP használja

- Raw (IP) socket

Multiplexelés-demultiplexelés, összeköttetésmentes eset (UDP)

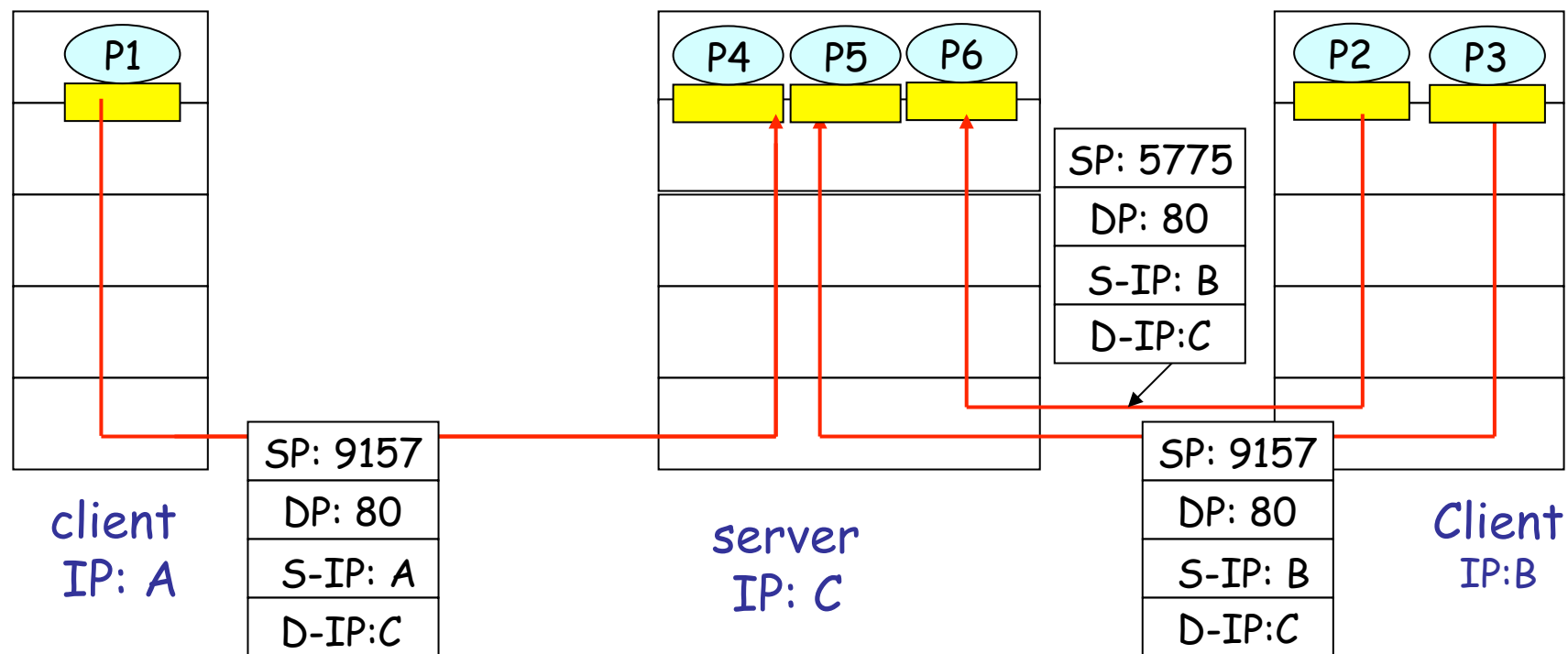
UDP socket: (cél IP cím, cél portszám) – “two-tuple”



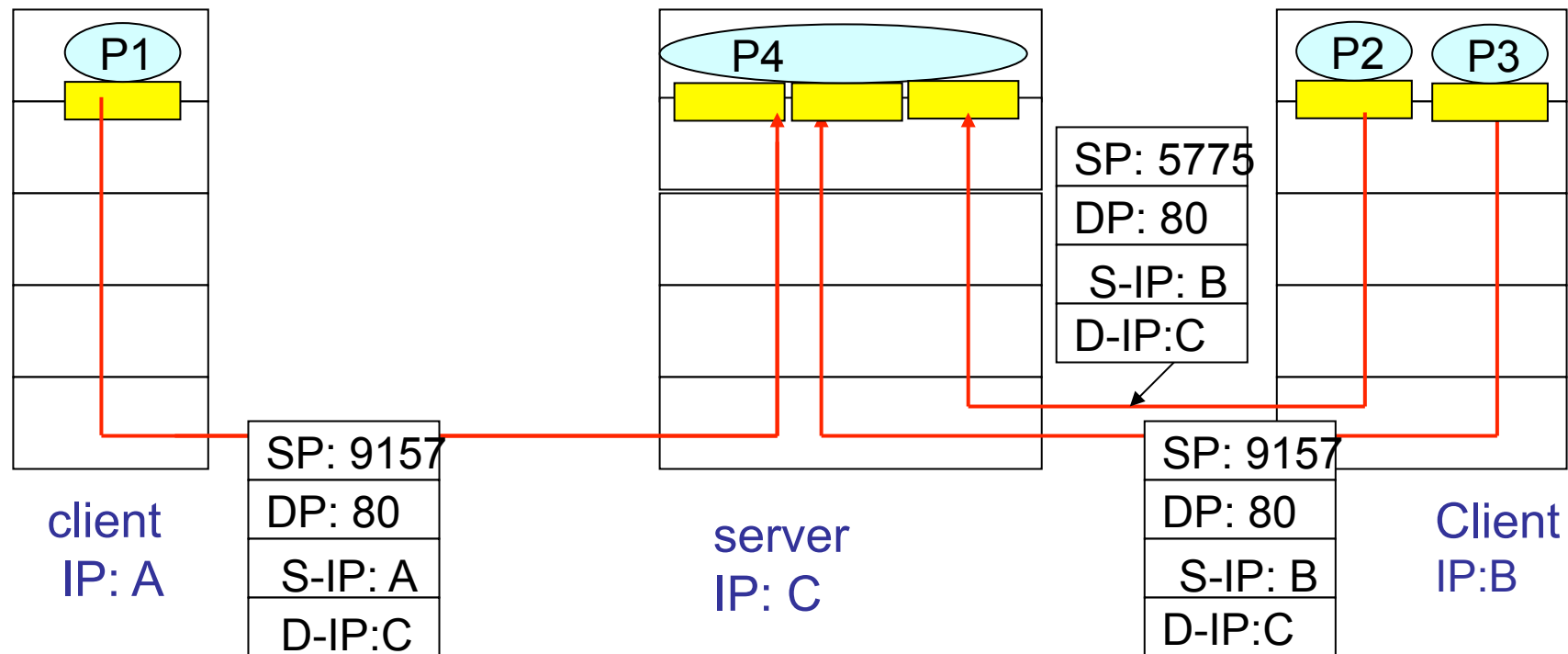
SP megadja a “vissza” címet

Multiplexelés-demultiplexelés, összeköttetés-alapú eset (TCP)

TCP socket: (forrás IP cím, forrás portszám, cél IP cím, cél portszám) – “4-tuple”



Összeköttetés-alapú eset (TCP): Threaded Web Server



TCP kliens-szerver socket kezelés

- Szerver létrehoz „**hallgató**” **módban** lévő socketeket
 - várja a kliensek kapcsolatfelvételét
- Kapcsolatfelvétel után: **dedikált socket** minden kapcsolathoz
 - Socket-socket közötti virtuális áramkörkapcsolás: TCP kapcsolat, duplex byte folyam
- Szerver különböző TCP socketeket hozhat létre ugyanazzal a helyi IP címmel és port számmal
 - mivel a távoli host IP címével, portjával **más socket párt alkot**
 - szerver gyerek processz összerendelése a kliens processzával
- UDP-ben **nincs dedikált socket**
 - Nincs külön gyerek processz minden távoli processzhez, egy processz kommunikál velük

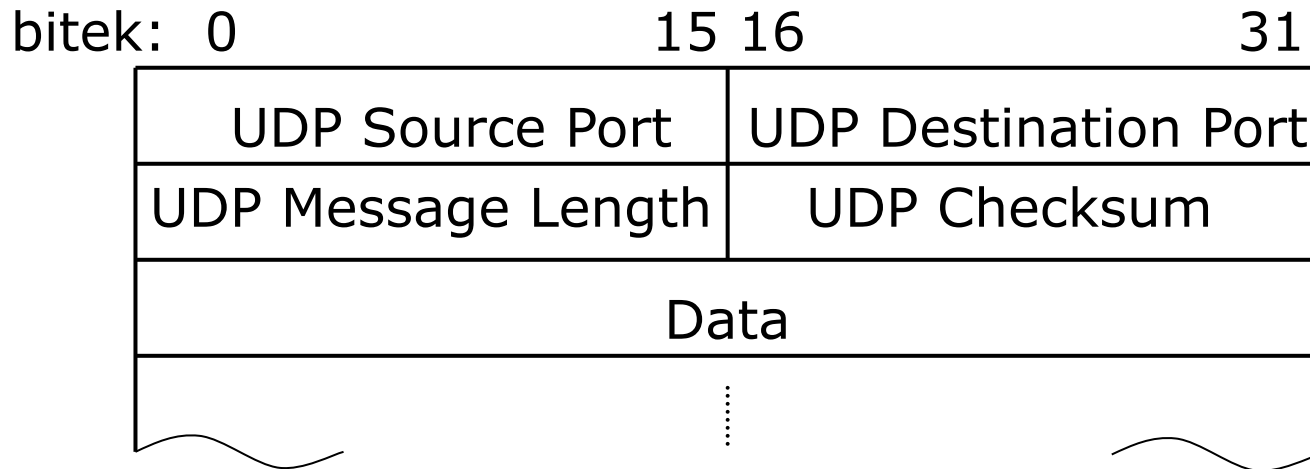
Egyszerű (raw) socket

- **Közvetlenül** az alkalmazásnak továbbítja a csomagot a **fejléccel**
 - Nincs TCP/IP feldolgozás, alkalmazás látja el fejléccel/veszi le a fejléceket
- Windows XP-ben 2001-ben jelent meg: biztonsági aggályok (Unixban régóta)
 - Félelem **TCP reset támadástól**: „lelövi” a TCP kapcsolatot
 - +: gyanús kapcsolatok megszakítása
 - -: harmadik fél beékelődve megszakítja a kapcsolatot
 - +/- ? Peer-to-peer forgalom szűrése (Comcast-NNSquad eset)

UDP (User Datagram Protocol)

- Nincs kapcsolatfelépítés, kézfogás
- Semmi garancia:
 - a csomagok sorrendjére
 - elvesztésének detekciójára
- A megbízható átvitel garantálását az alkalmazási rétegre bízza
- *Unreliable* Datagram Protocol 😊
- Multiplexálást és opcionálisan adatintegritás ellenőrzést nyújt

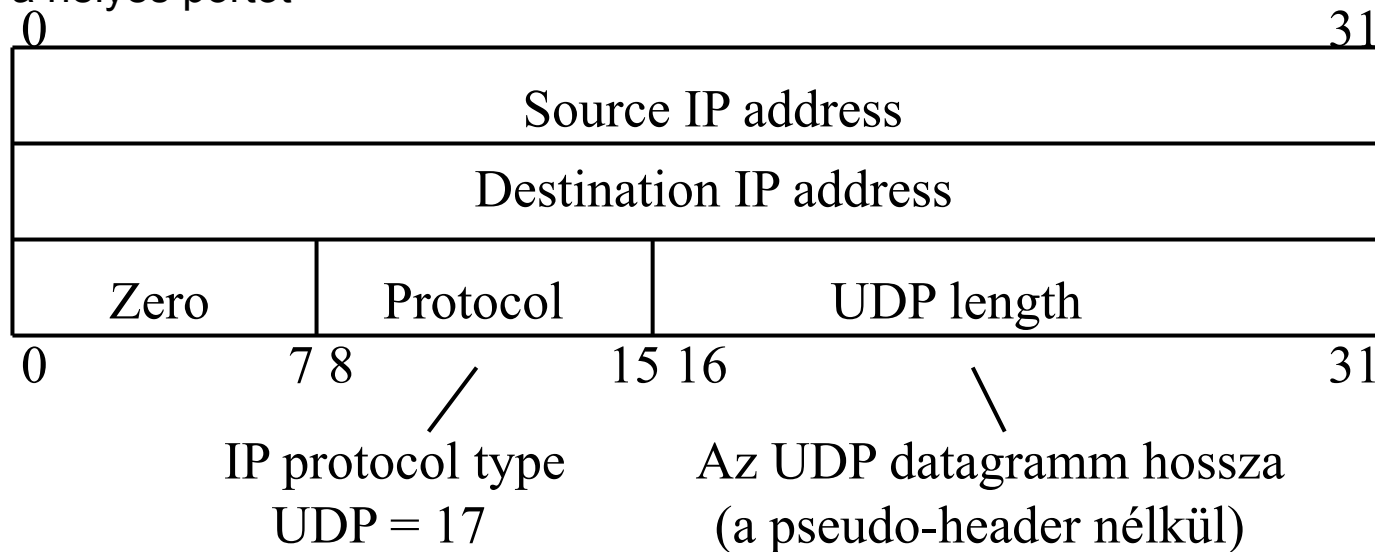
UDP – User Datagram Protocol (2)

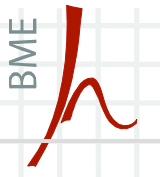


- **Source port** opcionális
- **Length**: oktettben, max (65 535-8-20)
- **Checksum**: opcionális (nincs: 0, IPv6-nál már nem opcionális)
- Az UDP checksum az egyetlen lehetőség annak ellenőrzésére, hogy a datagram helyesen érkezett-e meg
 - **adatra és fejlécre!**
 - az IP-csomag ellenőrzése csak a fejlécet fogja át

UDP – User Datagram Protocol (3)

- Az ellenőrző összeget „pseudo-header” hozzáadásával együtt számolja (IPv4-re)
 - De a pseudo-headert nem viszi át, csak a számoláshoz használja
- Tartalmazza az IP-címeket is
 - annak ellenőrzésére, hogy a datagramm elérte a helyes címzettet, nemcsak a helyes portot





Példa ellenőrzőösszegre

- Túlcsordulást figyelembe kell venni
- Példa: két 16 bites integer összeadása

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
carry bit	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
összeg	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
ellenőrzőösszeg	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Miért használunk UDP-t?

- Nincs kapcsolatfelépítés és torlódásvezérlés: kisebb késleltetés
- Egyszerű: nem kell a kapcsolat állapotait tárolni az adónál, vevőnél
 - Pl. szerver több aktív kapcsolatot tud támogatni
- Kis méretű fejléc
- Nincs torlódásvezérlés: gyorsan eléri a maximális adatátviteli rátát

UDP – User Datagram Protocol

Összefoglalás: funkcionalitás és a költsége

- Portkezelés
 - a különböző alkalmazások/folyamatok megkülönböztethetők
- Több alkalmazás egyidejű kezelése
 - port-hozzárendeléssel és multiplexeléssel/demultiplexeléssel
- Hibajelzés az UDP datagram tartalmára és az IP csomag további részeire
- A fentiek költsége:
minimum 8 oktettnyi overhead

UDP alkalmazása

- Média: streaming, valós idejű játékok, VoIP, IPTV
 - habár a streaming szolgáltatások nagy része már TCP-t használ (nem valós időben)
- Gyors és rövid lekérdezések:
 - DNS
 - DHCP
 - RIP
 - NTP
- Tipikusan a hálózati forgalom pár százaléka
 - De növekszik a részaránya és nincs torlódás szabályozás: aggályok!
 - Datagram Congestion Control Protocol (DCCP)

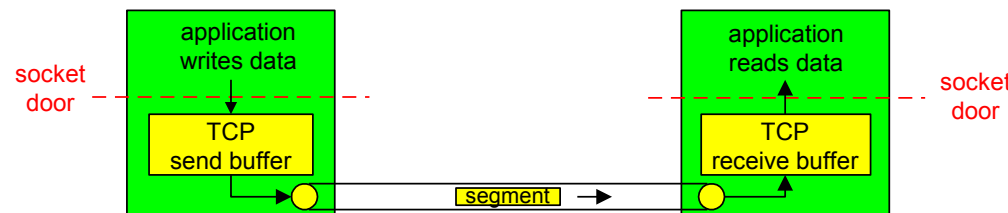
TCP megalkotói

- Vinton Cerf és Robert Kahn, 1974
- Előzetesen egy protokollnak képzelték el: TCP/IP, később szétvált
- PC, Ethernet, LAN, Web, streaming korszak előtt!
- Elég általános, hogy ma is működjön (módosításokkal)
- Megkapták érte az ACM Turing díjat

TCP –Transmission Control Protocol

Fő jellemzői

- **Célkitűzés:** megbízható szállítási szolgáltatás nyújtása az IP nem megbízható datagram-szolgáltatásán
- **Jellemzői:**
 - **Virtuális összeköttetések:** összeköttetés épül fel és marad fenn a kommunikáció tartamára (3 utas kézfogás)
 - **Stream-típusú szolgáltatás:** byte streamek sorrendhelyes átvitele
 - **Pufferelt átvitel:** a streamből a datagramm megtöltéséhez szükséges mennyiséget várja össze (küldő és vevő oldali buffer), max MSS
 - **Duplex kapcsolatok:** két független stream
 - **Vezérlő információk küldése:** az ellenkező irányban folyó streambe ágyazva (piggybacking)



Összeköttetés alapú

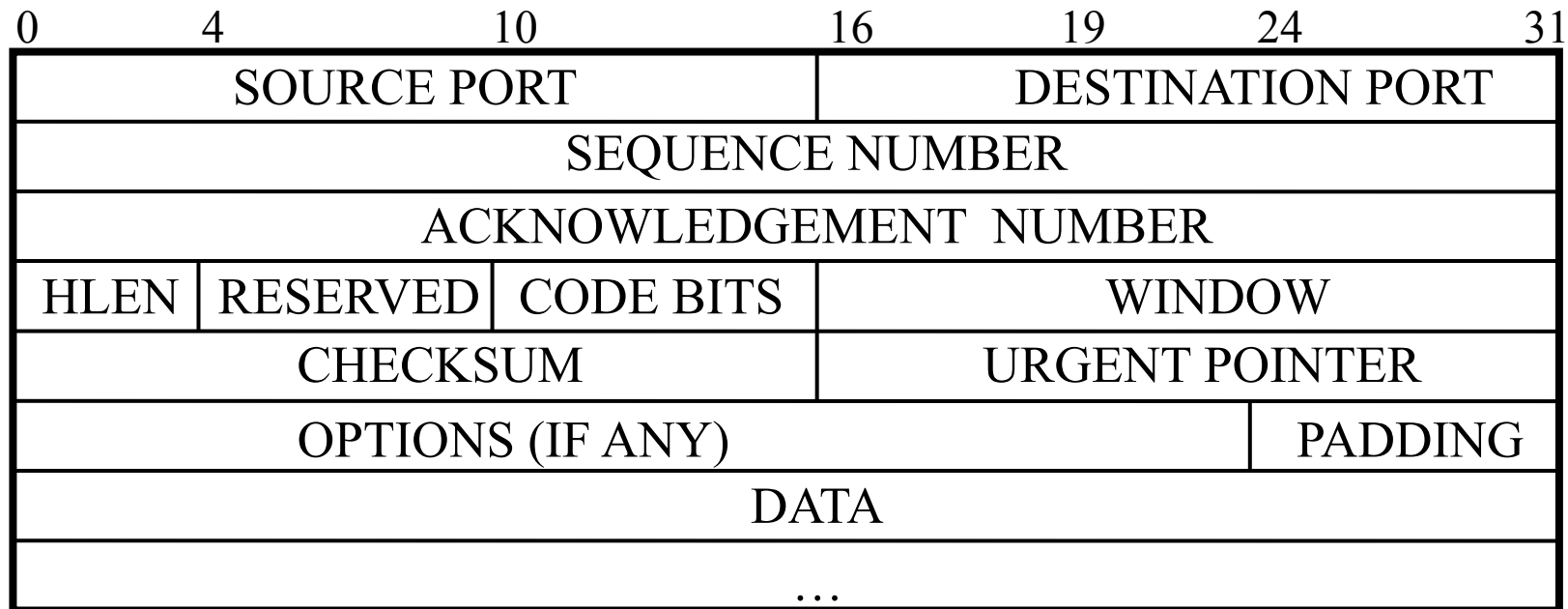
- Előzetes kapcsolatfelépítés (“kézfogás”) a két végpont processz között
- Se nem áramkörkapcsolás (TDM, FDM), se nem virtuális áramkörkapcsolás
 - Miért nem?
- Pont-pont: nem képes multicastre

TCP – Transmission Control Protocol

Miről lesz szó?

- **Adategység:** szegmens; szegmensstruktúra
- **Megbízható átvitel** sorszámozás és pozitív nyugtázás segítségével
- Összeköttetés-alapú kommunikáció: **kapcsolatfelépítés és -lebontás**
- **Forgalomszabályozás** (flow control) ablakmechanizmus segítségével
- **Torlódásvezérlés** (congestion control)

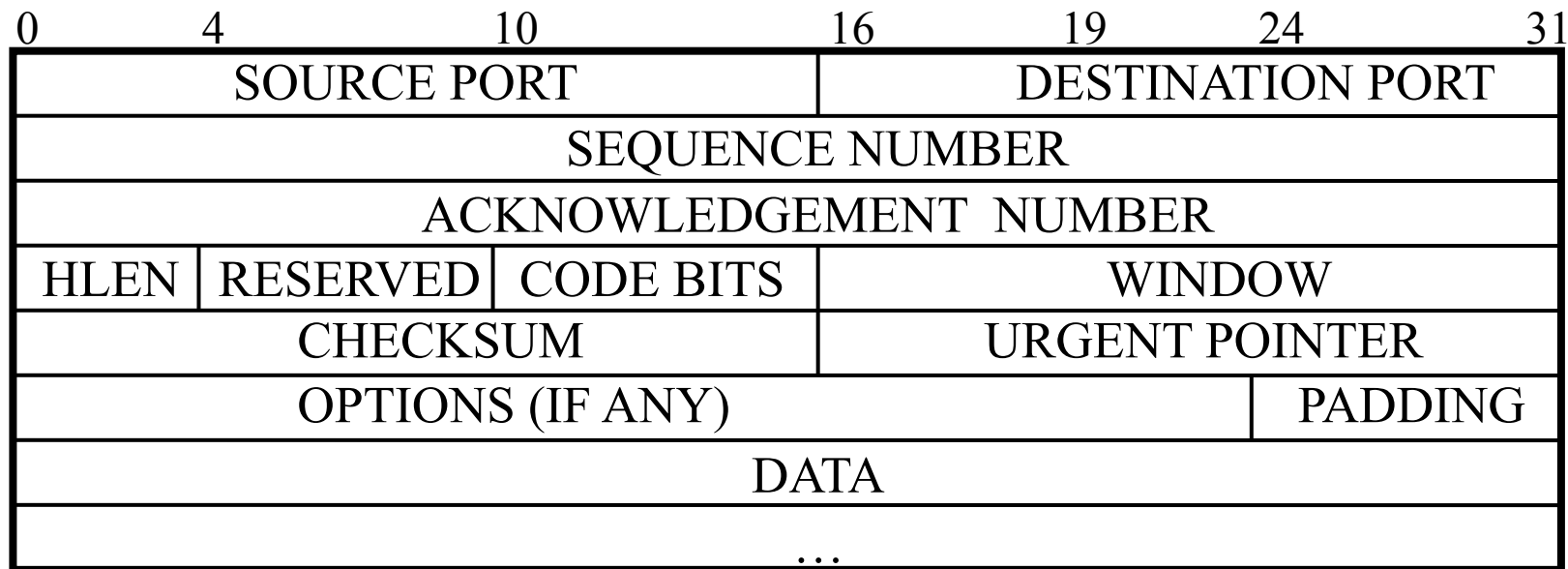
A PDU – Protocol Data Unit (a TCP-ben: „segment”) struktúrája



- **Sequence no.:** a szegmensben levő adat első byte-jának pozíciója a küldő byte stream-jében
- **Ack no.:** annak a byte-nak a sorszáma, amelyet a forrás legközelebb vár (előzőeket lenyugtázva ezzel)

TCP – Transmission Control Protocol

Szegmensformátum (folyt.)



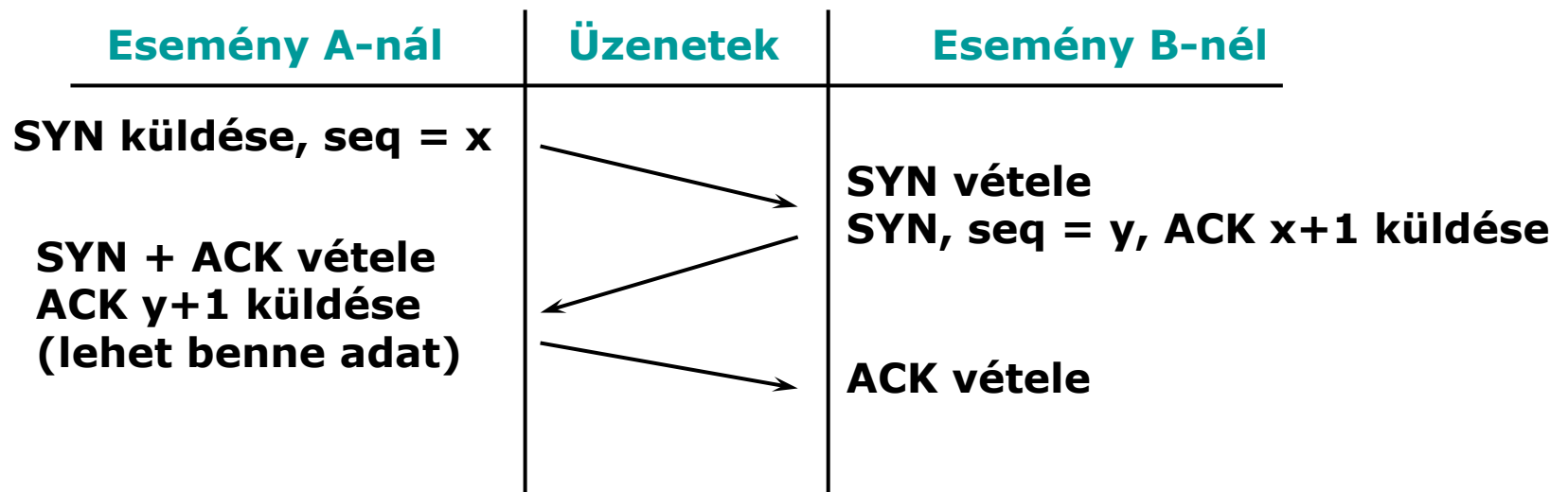
- **HLEN**: fejléc mérete, 4 oktettekben, minimum 20 byte, max 60 byte
- **Code bits** (flags): URG, PSH, ACK, RST, SYN, FIN bitek a kapcsolat kezeléséhez használt jelzőbitek
- **Window**: a küldő ismertté teszi a **vételi pufferének méretét**
- **Checksum**: mint az UDP-ben (pseudoheader)
- **Urgent pointer**: ha URG=1, a szegmens „urgent” részt tartalmaz, ilyenkor a végére mutat (pl. jelszóküldés)

MSS (Maximum Segment Size)

- A legnagyobb adatméret bájtban, amit a TCP hajlandó küldeni egy szegmens adat részében: alkalmazás adatai, fejlécek nélkül
- Össze kell egyeztetni az adatkapcsolati réteg MTU-jával
 - elkerülendő a tördelést
- TCP kapcsolatfelépítésnél kell egyeztetni, **MSS opció a fejlécben**
- TCP adó használhat Path MTU discovery-t is: dinamikus MSS változtatás

Hívásfelépítés a TCP-ben

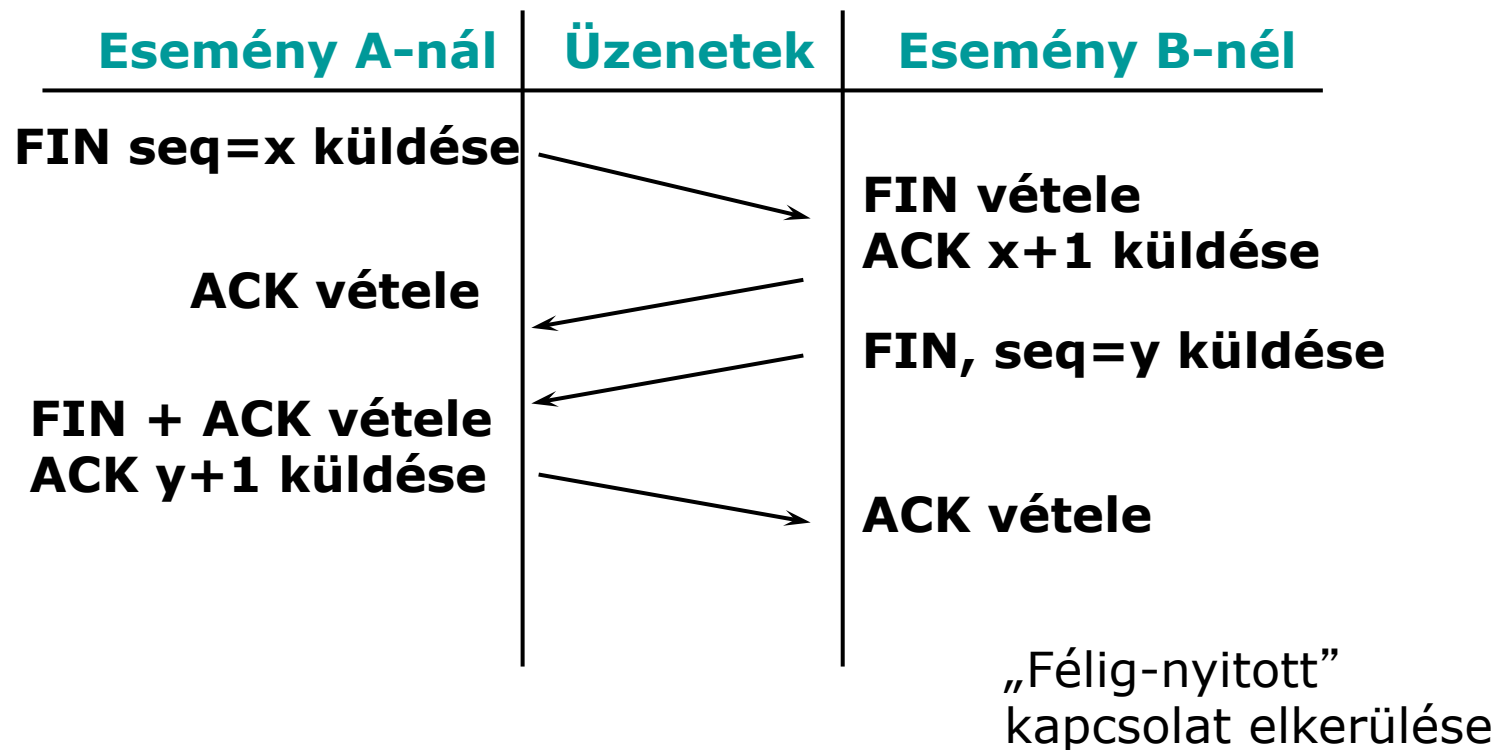
„3-way handshake” eljárás (3 utas kézfogás)



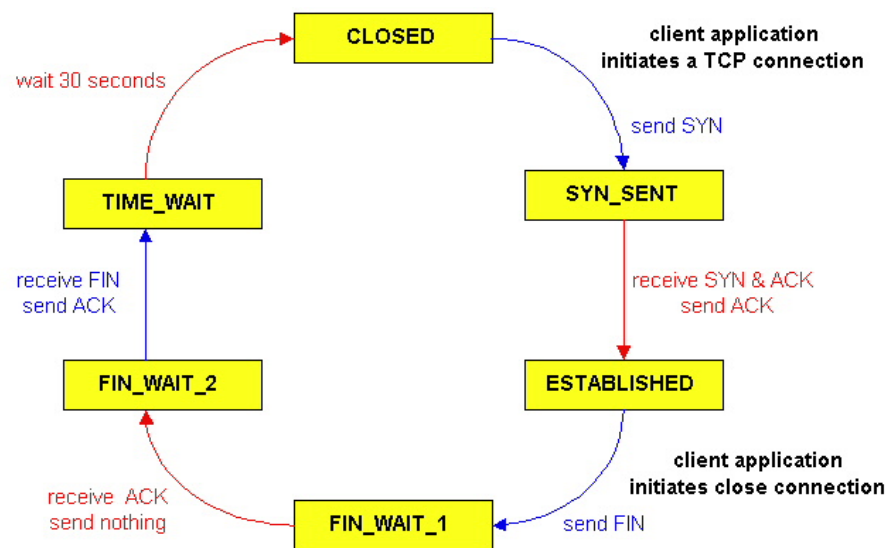
Elején véletlenszerű sorszám: a TCP sorszám predikciós támadás elkerülése

Híváslebontás a TCP-ben

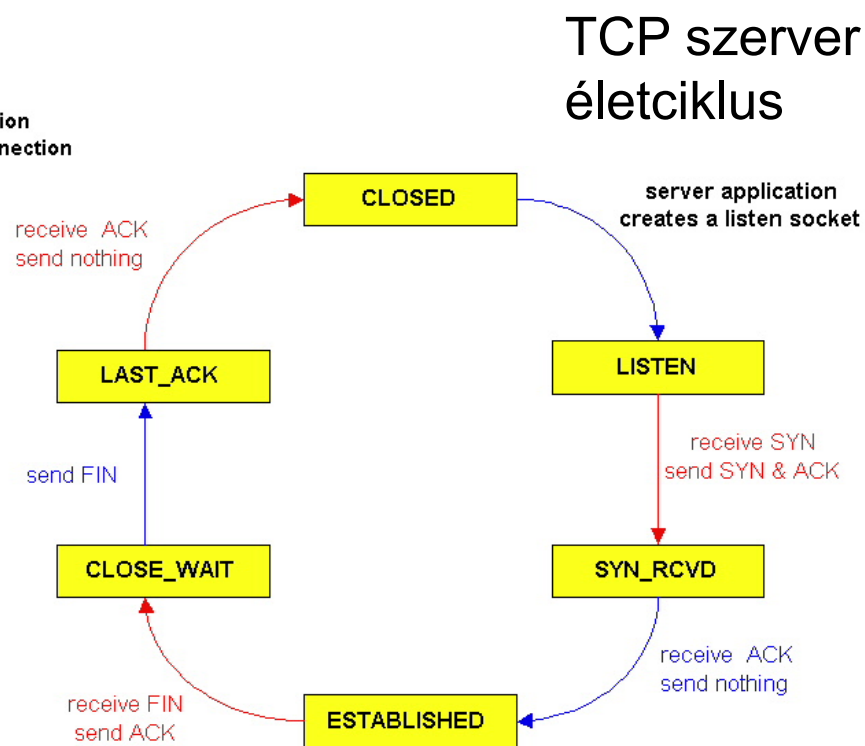
„Modified 3-way handshake” eljárás



TCP kapcsolat menedzsment



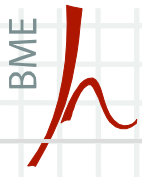
TCP kliens
életrajza



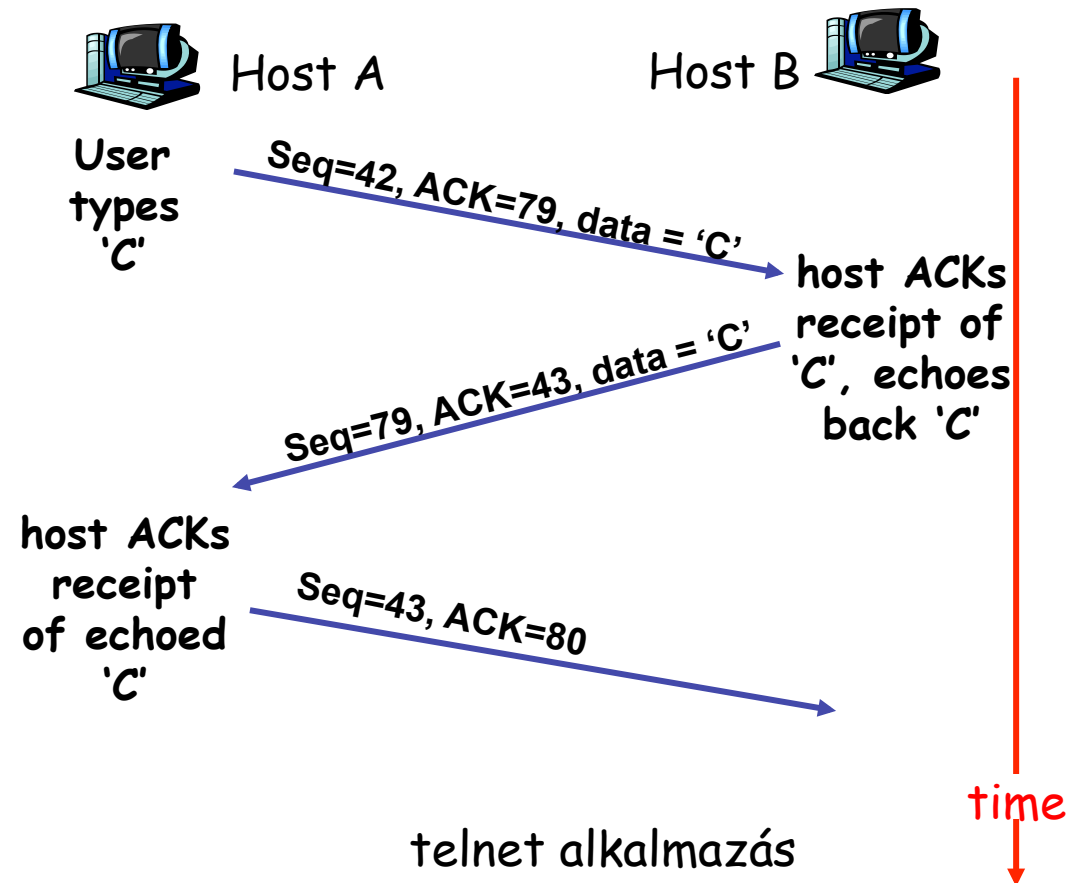
TCP szerver
életrajza

SYN “egyéb” használata

- TCP SYN elárasztásos támadás (SYN flood attack)
 - Nem küld a támadó harmadik szegmenst (ACK) a kapcsolatfelépítésnél
 - (Distributed) Denial of Service: leköti a szerver erőforrásait
 - Védelem: SYN vagy RST cookies
- Port feltérképező programok (pl. nmap)



Sorszámok és nyugtaszámok használata a TCP-ben

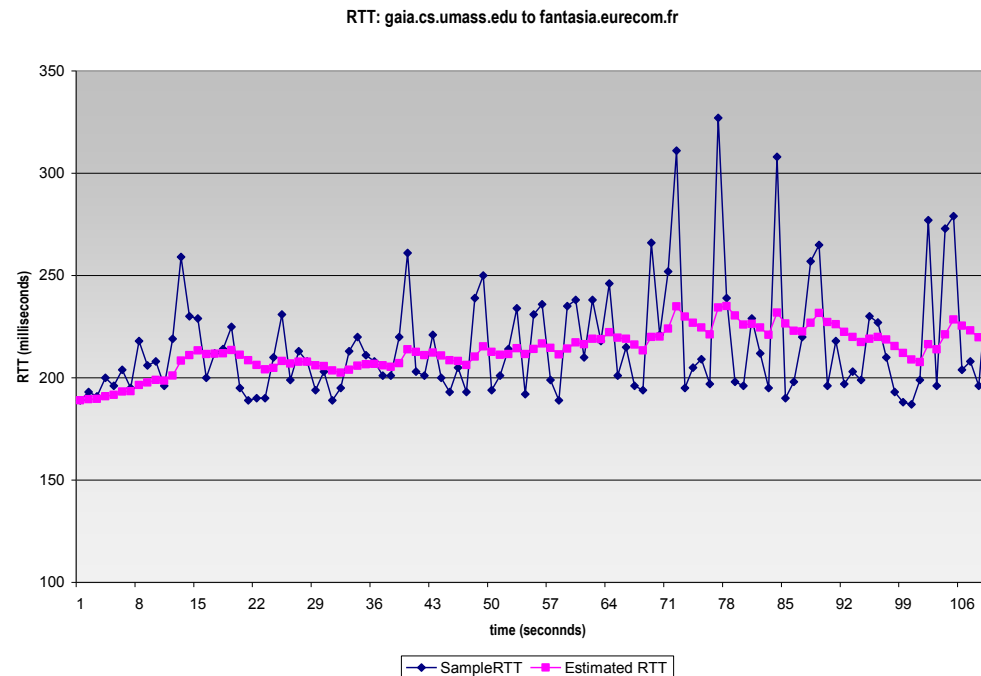


- Várakozás ACK-ra: **time-out**
- Mekkora válasszuk a time-out-ot?
- Probléma a túl kicsivel és a túl nagygal
- Megoldás:
a teljes terjedési időhöz
(RTT - round-trip time) igazítani, **adaptívvá** tenni
- Legyen hosszabb, de mennyivel, mivel RTT dinamikusan változik?
- Szabályok arra, hogy mit tegyünk, ha nem jön ACK a time-out alatt

Time-out és RTT

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

- SampleRTT: pillanatnyi RTT
- Becsült RTT (Estimated): exponenciális mozgó átlaggal
- a régebbi RTT-k súlya exponenciálisan csökken
- jellemzően: $\alpha = 0.125$



Time-out meghatározása

Time-out idő beállítása:

- EstimatedRTT + “biztonsági tartalék”
- Nagy ingadozás EstimatedRTT értékében -> nagy biztonsági tartalék

Becslés SampleRTT szórására EstimatedRTT-től:

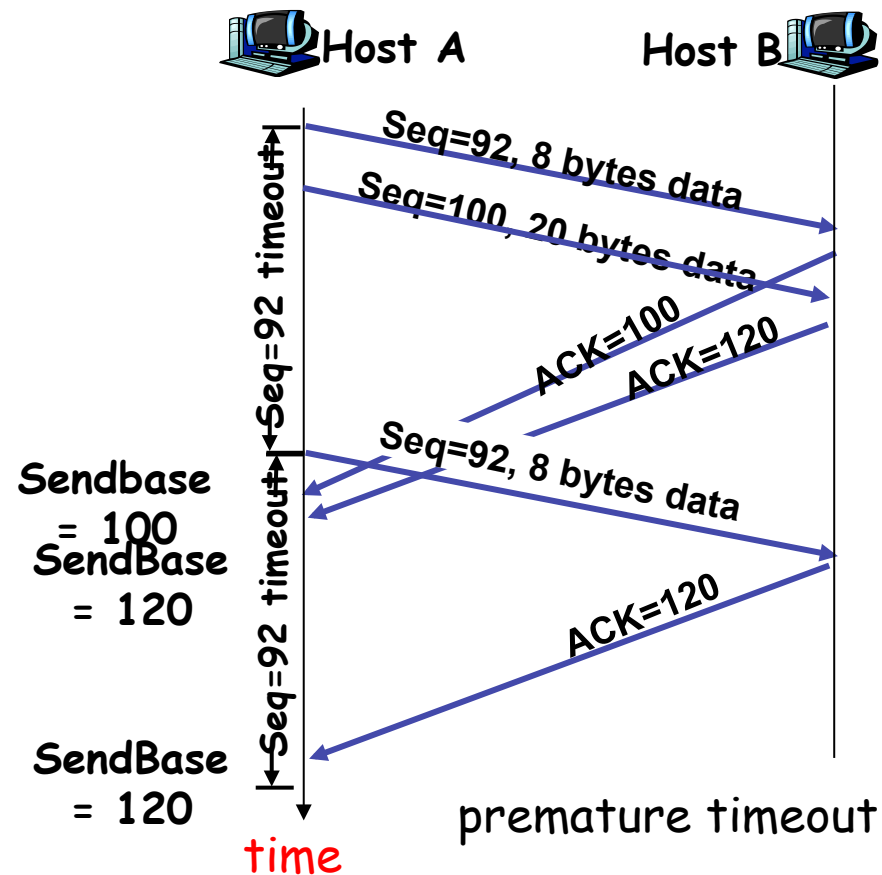
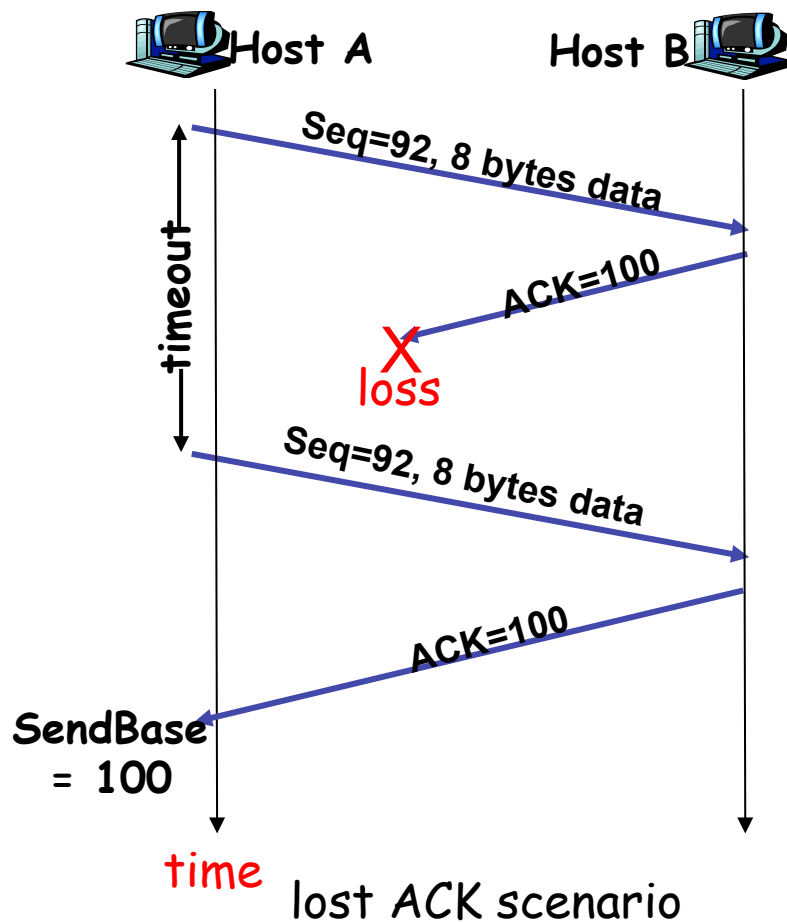
$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(jellemzően $\beta = 0.25$)

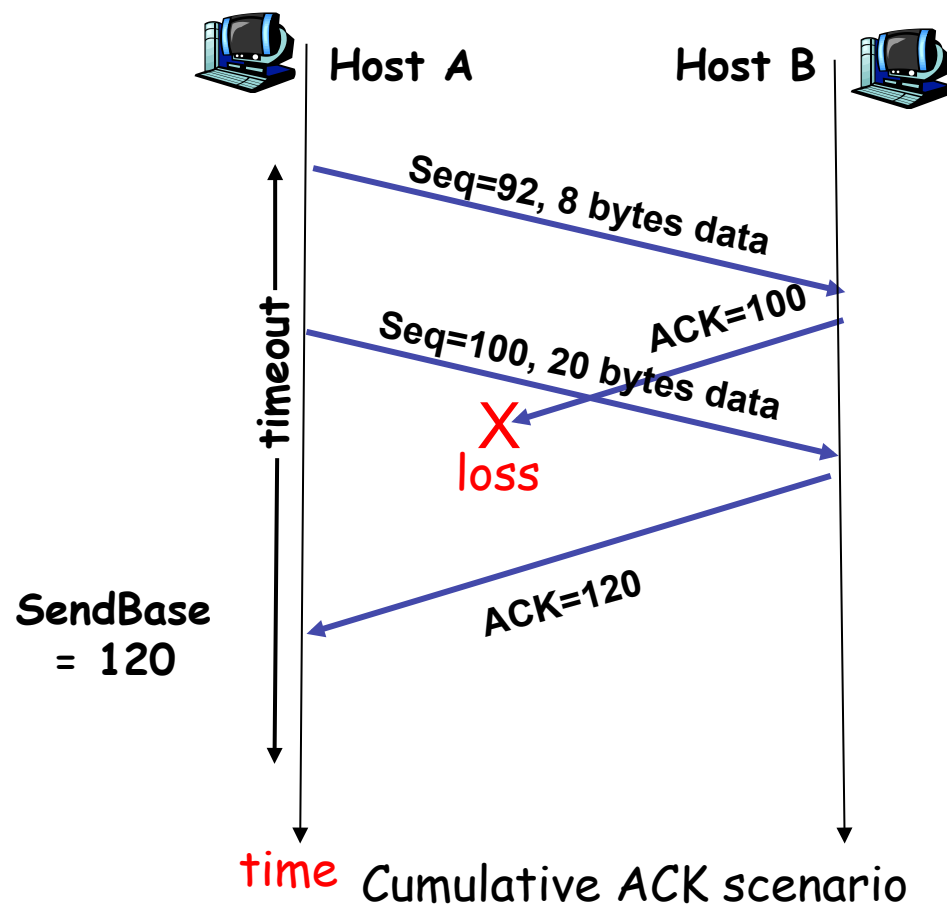
- Time-out értéke:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

Újraküldési esetek a TCP-nél: elveszett nyugta és korai timeout



Újra küldési esetek a TCP-nél: összevont nyugta



Time-out kétszerezés

- Mai TCP implementációkban: amennyiben lejár a timer és újraadás van, kétszereződik a time-out értéke
 - Miért? Hol használtunk hasonlót?
- Minden újraadás után exponenciálisan növekszik
- Nyugta vétele után visszaáll az EstimatedRTT és DevRTT kombináció
- Torlódásvezérlés egyik eszköze

Gond összevont nyugtával

- Pl. 10000 bájt kerül elküldésre 10 szegmensben
 - Ha elveszik az első szegmens, a fogadó nem tudja visszajelezni, hogy 1000-9999 megjött, de 0-999 nem
 - Újra el kell küldeni a teljes 10000 bájtot
- Megoldás: **szelektív nyugtázás (SACK)**
 - SACK-ban meg tudja mondani, hogy 1000-9999 megjött sikeresen
 - Opcionális fejrészmezőben
 - Népszerű, minden TCP implementációban

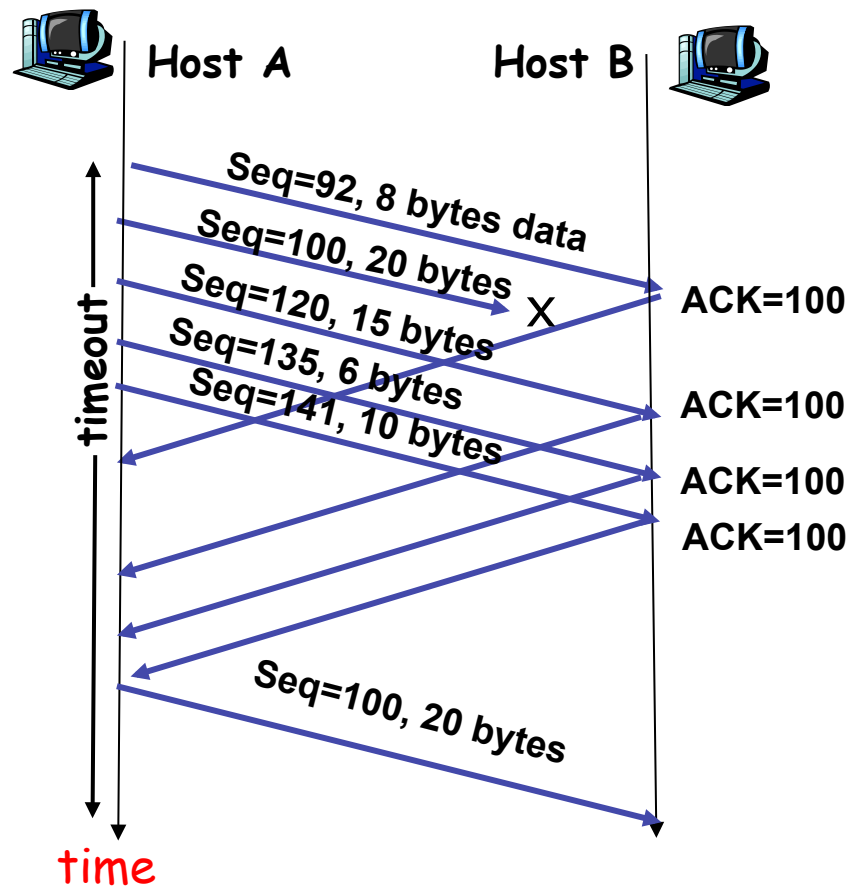
Gond sorrend keveredéssel

- Sorrendkeveredés miatt elveszett csomagnak hiszi a küldő
->újraküldés->forrás visszafogás, timer kétszerezés
- Megoldás: **D-SACK (duplikált nyugta)**
 - Fogadó szól, hogy az újraküldött csomag duplikátum
->visszagyorsul a forrás
 - Opcionális fejrészmezőben
 - Népszerű, minden TCP implementációban

Fast retransmit (1)

- A **time-out idő** gyakran **túl hosszú**:
 - nagy késleltetés előtt az elveszett csomagot az adó újra tudná küldeni.
- De hogy értesülhetne az elveszett csomagról előbb, mintsem hogy letelt volna a timeout?
 - Az elveszett szegmensekre utalhatnak a duplikált ACK-ok
 - Ha a vevő hézagot vesz észre a vett szegmensek sorozatában (elveszett csomag) akkor újból lenyugtázza a megelőző helyesen vett szegmenst.
 - Több egymást követő duplikált ACK érkezhethet.
- **Fast retransmit szabály**: ha az **adó 3 egymást követő ACK-t kap** (plusz az eredeti ACK) ugyanarra a szegmensre, feltételezi, hogy az azt követő szegmens elveszett és
 - **újraküldi azt még mielőtt lejárna a timeout.**

Fast retransmit (2)



Számítási példa (Fast retransmit)

Az A és B végpont közötti kommunikáció során az A végpont által küldött utolsó 4 szegmens TCP PDU-jában a sorszám (sequence number) 245, 296, 372, 458. B válaszként küldött 4 szegmens TCP PDU-jában az ACK-szám 372, 372, 372, 372.

Erre válaszként az A terminal hány bájtos szegmenst fog küldeni fast retransmit esetén, amennyiben nem járt még le a time-out idő?

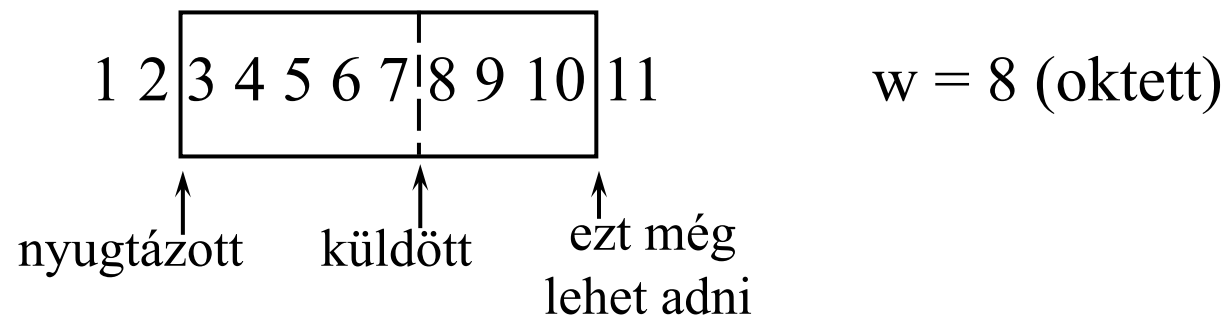
Számítási példa (TCP vételi ablak)

Az „A” és „B” végpont közötti kommunikáció során „A” végpont utolsóként elküldött TCP PDU-jában a sorszám (sequence number) 8350, a hasznos adatrész 1400 byte.

„B” válaszként küldött TCP PDU-jában az ACK-szám 8050.

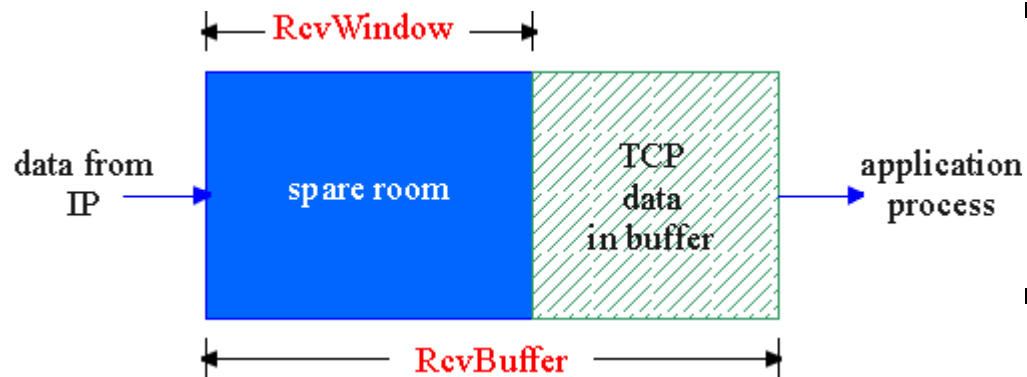
Hány byte-nyi adatot küldhet még „A” a következő nyugta megérkezéséig, ha a B vételi ablakmérete 2000?

Flow control: a „sliding window” módszer elve



- Csúszóablakos („sliding window”) mechanizmus
 - az ablak mérete megadja a „kintlevő”, nyugtázatlan csomagok max. számát. (Pl.: $w=8$)
- A TCP-ben: az ablak-mechanizmus oktetteken működik

TCP Flow control: hogy működik?



- Spare room:
= **RcvWindow**
= $\text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$

- A vevő közli a szabad helyének a méretét (**RcvWindow**) a küldött szegmensben
- Az Adó legfeljebb **RcvWindow** mennyiségű nyugtázatlan adatot küld
- Nem szabad összekeverni a torlódási ablakkal (**CongWin**)

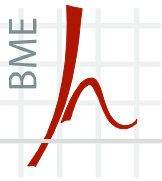
Csúszóablakos problémák

- I. Ha a vevő **nullás csúszóablak** méretet hirdet: adó leáll a küldéssel
 - Ha **elveszik a vevő csomagja vagy nincs küldendő csomagja, nincs infó az új csúszóablak méretről**: adó vár hiába
 - Megoldás: adó egy **timert** indít, lejártá után felkéri a vevőt, hogy küldjön ACK-ot az új ablakméretről
- II. **Buta ablak jelenség**
 - Ha a vevő oldalon kicsi (akár 1 byte) szabadul fel, lehet 1 byte az ablak
 - A küldő 1 byte-ot küld, a vevő megint 1 byte-tal nyit
 - Erőforráspocsékolás: kisebb az adat mint a fejléc!
 - Megoldás:
 - A vevő nem nyitja az ablakot, csak akkor, **ha MSS nagyságrendűt nyithat**
 - A küldő nem küld, hacsak nem
 - MSS-nyit küldhet
 - Mindent küldhet, amit az alkalmazás kért

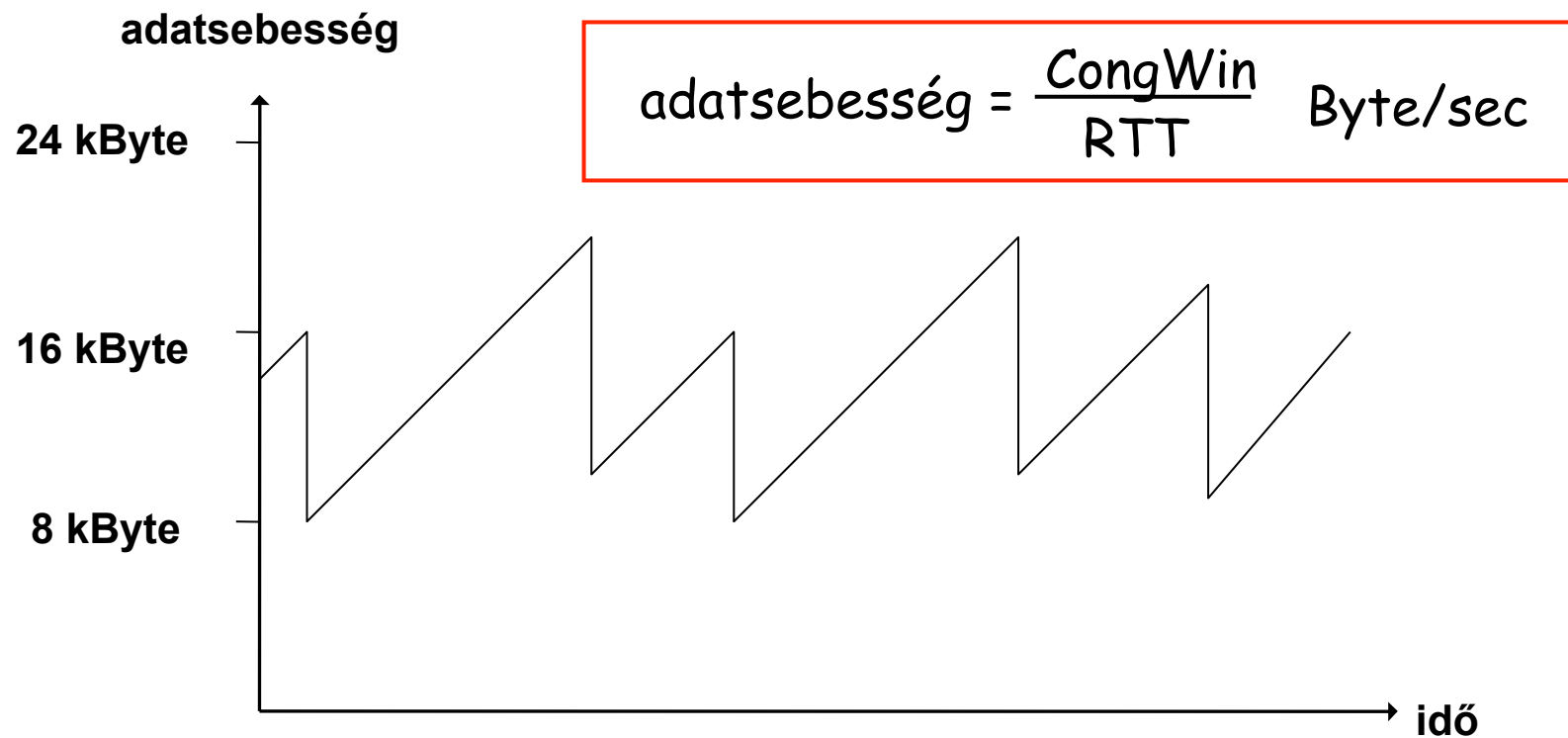
- A torlódásvezérlésről általában:
 - Azok a módszerek, amelyekkel linkek, csomópontok időszakos túlterheltségét megkíséreljük megszüntetni.
- Két fő módszer:
 - „Hálózat által segített” (**network assisted**) torlódásvezérlés
 - A hálózati elemek szolgáltatnak információt a túlterhelésről az adónak
 - Pl. TCP/IP ECN, ATM: külön jelzéscsomagok ehhez
 - Végpontok közötti (**end-to-end**) torlódásvezérlés
 - nincs visszacsatolás a hálózathoz, a végpont következtet arra, hogy torlódás léphetett fel
- A TCP az utóbbit csinálja!

Torlódásvezérlés a TCP-ben

- Módszer:
 - **növeljük az adatsebességet**, ha úgy érzékeljük, van elég átbocsátóképesség
 - amíg **torlódásra utaló jeleket** nem tapasztalunk, ha igen, csökkentjük.
- Hogyan:
 - congestion window, **CongWin**
 - növeljük a CongWin-t minden RTT alatt MSS-sel, amíg veszteséget nem érzékelünk
 - csökkentjük a CongWin-t felére minden vesztéskor
 - = **Additive increase – multiplicative decrease (AIMD)** módszer
- Hogyan érzékeljük a torlódást (vesztést)?
 - timeout letelt, nem jött nyugta
 - többszörös nyugta érkezett ugyanarra a szegmensre



Az adatsebesség alakulása AIMD esetén

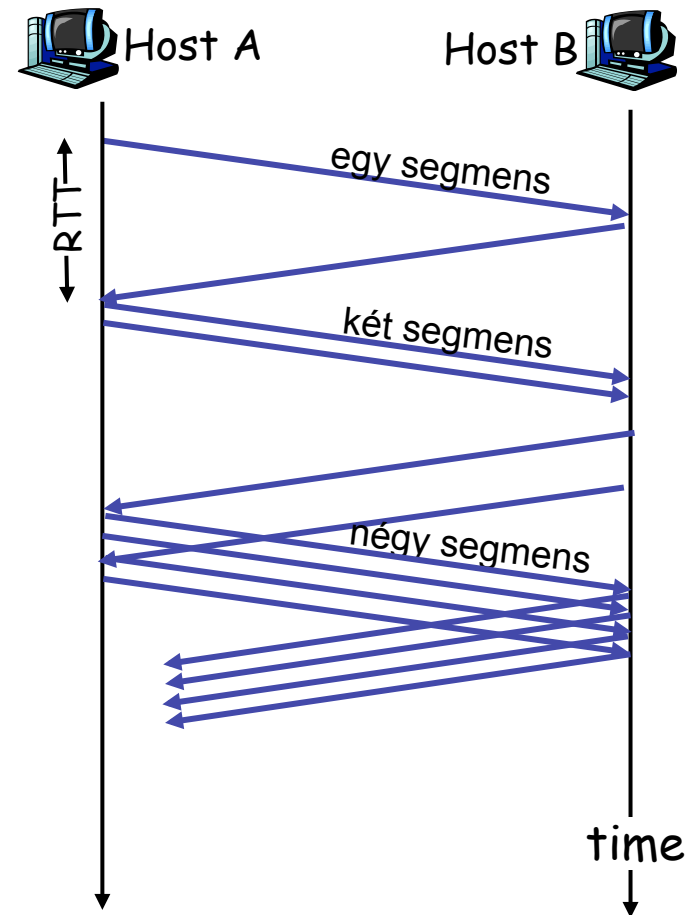


$$\text{LastByteSent} - \text{LastByteAcked} \leq \min \{ \text{CongWin}, \text{RcvWin} \}$$

- Az AIMD kiegészítései:
- **„Slow Start”**
 - az összeköttetés kezdetén a sebesség gyors (exponenciális) növelése az első veszteségig (vagy korlátig)
 - utána AIMD
 - Gyors növelés: minden ACK után kétszerezzük
- Eltérő viselkedés timeout és többszörös (3-szoros) nyugták esetén

TCP Slow Start

- Az elején: **CongWin** = 1 MSS
 - Példa: MSS = 500 byte & RTT = 200 msec
 - kezdeti sebesség = 20 Kb/s
- Mivel a rendelkezésre álló átb. képesség $\gg$ CongWin/RTT lehet,
 - célszerű gyorsan elérni, ezért
 - induláskor exponenciálisan növelni az első veszteség



- 3 duplikált ACK után:
 - **CongWin** felére csökken
 - Utána lineáris növekedés
 - **torlódás elkerülési fázis**
- De timer lejártá esetén:
 - **CongWin** 1 MSS lesz;
 - Utána exponenciálisan nő,
 - Egy korlátig, majd onnan lineáris (**torlódás elkerülési fázis**)

Filozófia:

❑ timer lejártá rosszabb torlódási helyzetet jelez, mint a 3 duplikált ACK

Összefoglaló: TCP torlódás vezérlés

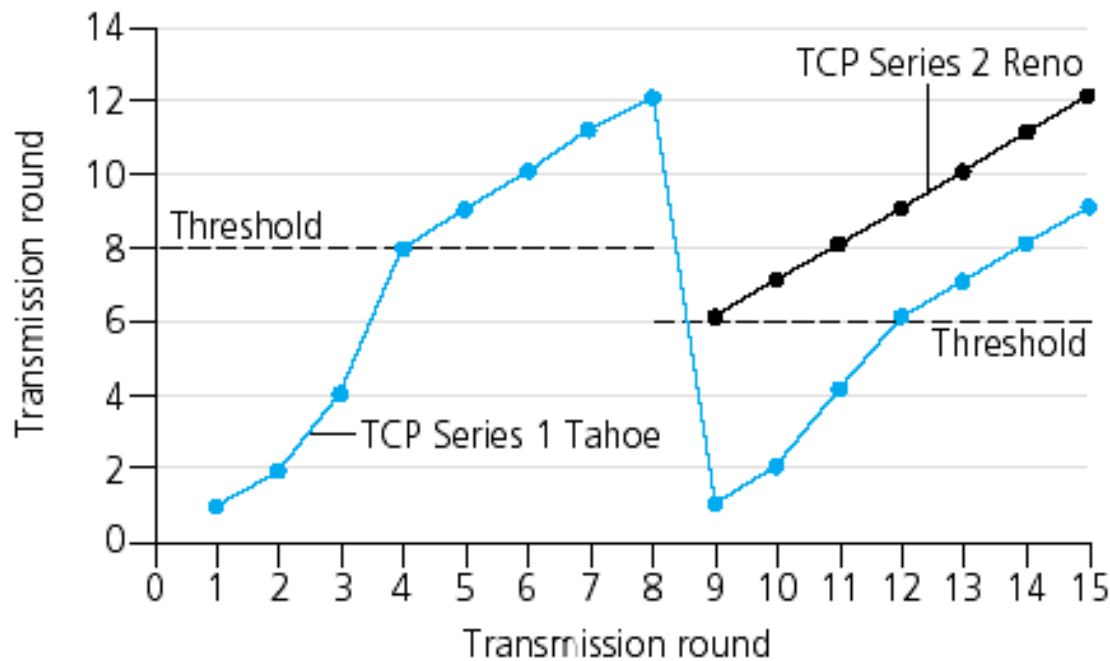
- Ha a **CongWin** egy korlát alatt van, adó **slow-start fázisban**, exponenciális ablak növelés
- Ha **CongWin** a korlát felett, az adó **torlódás elkerülési fázisban**, lineáris ablak növelés
- Ha **3 duplikált ACK**, a korlát $\text{CongWin}/2$ lesz és a **CongWin** pedig a korlát
- Ha **timer lejárt**, a korlát $\text{CongWin}/2$ lesz, míg a **CongWin** 1 MSS lesz

TCP sender congestion control *

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	$\text{CongWin} = \text{CongWin} + \text{MSS}$, If ($\text{CongWin} > \text{Threshold}$) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	$\text{CongWin} = \text{CongWin} + \text{MSS} * (\text{MSS} / \text{CongWin})$	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	$\text{Threshold} = \text{CongWin} / 2$, $\text{CongWin} = \text{Threshold}$, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	$\text{Threshold} = \text{CongWin} / 2$, $\text{CongWin} = 1 \text{ MSS}$, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP implementációk

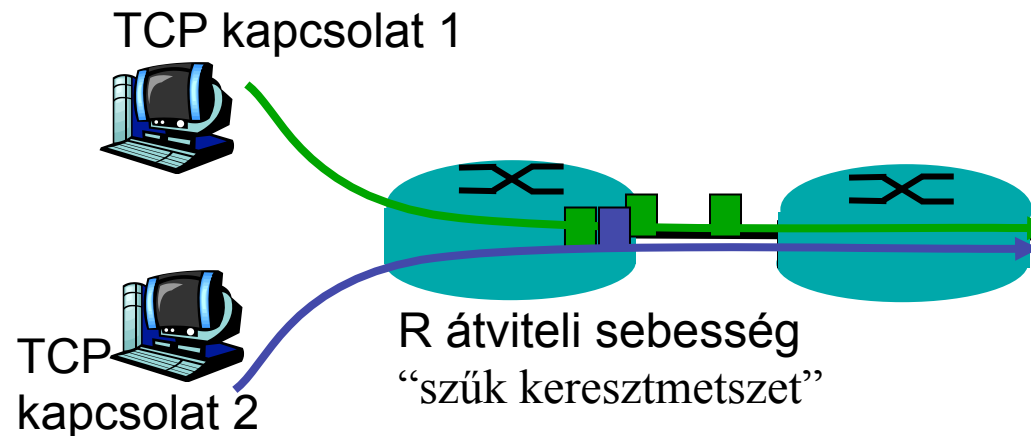
- Korlát 8MSS
- 3 duplikált ACK a 9.átvitelnél:
 - **Tahoe**: nincs fast recovery, így 1MSS-ról indul újra (mint ha timer járt volna le), exp. a korlátig, onnan lineáris
 - **Reno**: CongWin/2-ről lineáris (amennyiben timeout, olyan mint Tahoe)



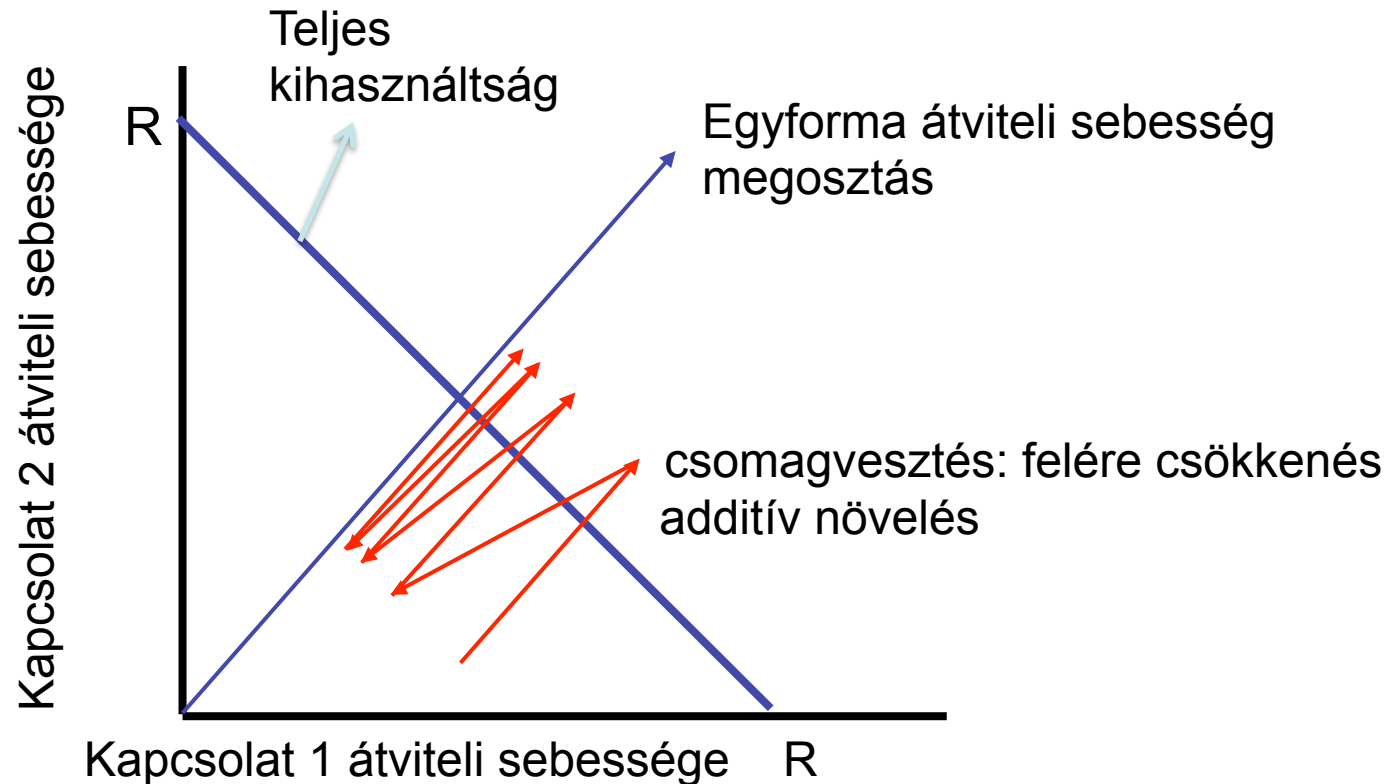
TCP átbocsátóképessége

- TCP átlagos átbocsátó képessége az ablak méret és RTT függvényében?
 - Nem foglalkozunk slow-start-al, ami csak az elején van és nagyon rövid ideig tart
- W az ablakméret a csomagvesztés pillanatában
- Csomagvesztés előtt az átbocsátás: **W/RTT**
- Vesztes után az ablak $W/2$ lesz, átbocsátás: **$W/2RTT$** .
- Átlagos átbocsátás: **$0.75 W/RTT$**

- Idealizált eset:
 - Mindkét kapcsolatnál egyforma MSS és RTT
 - Nincs más TCP vagy UDP kapcsolat
 - AIMD eset (nincs slow-start)



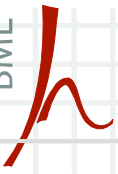
Két versenyző kapcsolat



- Kisebb RTT-vel rendelkező kapcsolatok gyorsabban növelik a CongWin-t → nagyobb átviteli sebesség!
- UDP nem fair, kiszorítja a TCP-t
- Párhuzamos TCP kapcsolatok
 - Böngészők több párhuzamos TCP kapcsolatot építenek fel a webszerverhez
 - Minél több kapcsolat, annál nagyobb átviteli sebesség!
- Példa: R átv. seb., kliens-szerver, 9 kapcsolattal
 - Ha az új alkalmazás 1 TCP kapcsolatot kér, sebessége: $R/10$
 - Ha az új alkalmazás 11 TCP kapcsolatot kér: több mint $R/2$!

TCP és UDP: összefoglalás

- o Mindkettő szállítási rétegbeli protokoll
- o Mindkettő portokat kezel
 - o multiplexelés/demultiplexelés
 - o ezáltal interfész nyújtása az alkalmazási folyamatok felé
- o Az UDP összeköttetés-mentes, best effort szolgáltatás
 - o nem garantál célba juttatást, csak hibajelzést nyújt
 - o gyorsan célba juttat
- o A TCP összeköttetés-alapú, megbízható transzport szolgáltatás
 - o sorrendhelyes, hibamentes szállítást nyújt
 - o ára: késleltetés



Néhány gyakori alkalmazás és a használt transzportprotokollok

<i>Alkalmazás</i>	<i>Alkalmazási rétegbeli protokoll</i>	<i>Használt transzport-protokoll</i>
<i>E-mail</i>	SMTP	TCP
<i>Távoli elérés</i>	Telnet	TCP
<i>Web-elérés</i>	HTTP	TCP
<i>File-átvitel</i>	FTP	TCP
<i>Routing</i>	RIP	UDP
<i>Hálózatmenedzsment</i>	SNMP	UDP, TCP
<i>VoIP, média-streaming</i>	Többnyire nem szabványos	UDP, TCP

Kérdések?

KÖSZÖNÖM A FIGYELMET!

Dr. Simon Vilmos

docens

BME Hálózati Rendszerek és Szolgáltatások Tanszék

svilmos@hit.bme.hu