

# תקשורת מחשבים ואלגוריתמים מבוזרים



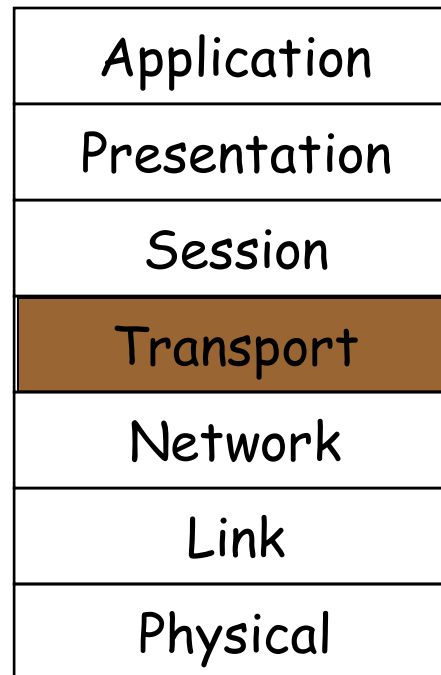
קורס מס' 202-2-1131

מתרגל: ד"ר גיא לשם [leshemg@cs.bgu.ac.il](mailto:leshemg@cs.bgu.ac.il)

הרצאה חמישית – שכבת התעבורה

# חיקוק שכבת תצורה באחד 7 השכבות

---



The 7-layer OSI Model

# מבוא לשכבת ההובלה

## המטרה שלנו:

□ להבין את העקרונות העומדים מאחורי שירותי שכבת ההובלה.

■ ריבוב (multiplexing) / פילוג (demultiplexing).

■ העברת אמינה של מידע.

■ בקרת זרימה.

■ בקרת עומסים.

□ ללמוד על פרוטוקלי שכבת ההובלה באינטרנט

■ UDP: תעבורה חסרת קשר (connectionless).

■ TCP: תעבורה מונחת קשר (connection).

■ TCP: בקרת עומסים (flow control).

# שכבת ההובלה

---

□ שרתי שכבת ההובלה.

□ ריבוב/פילוג. □ עקרונות של בקרת עומסים.

□ תעבורה חסרת קשר: UDP. □ בקרת עומסים TCP.

□ עקרונות תעבורת מידע

אמינה ברשת.

□ תעבורה מונחת קשר: TCP

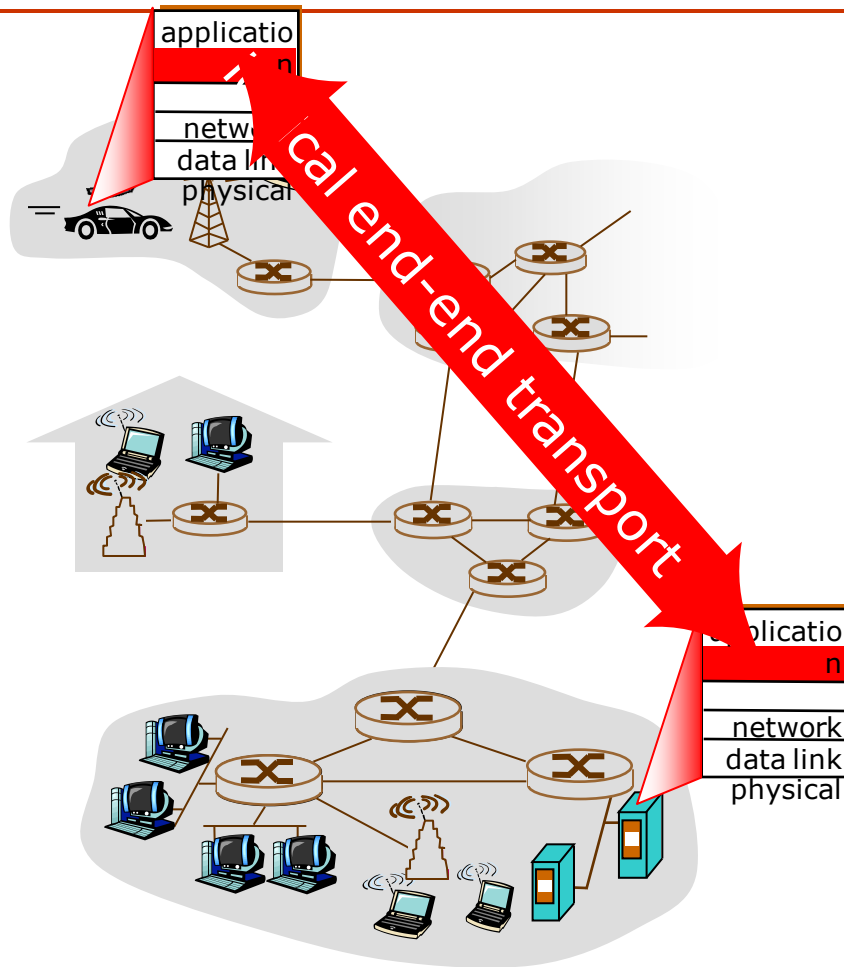
■ מבנה המקטע.

■ תעבורת מידע אמינה.

■ בקרת זרימה.

■ ניהול הקשר

# שירותי שכתת ההוללה ופרוטוקולים



□ שירותי ופרוטוקולי התעבורה מספקים חיבור לוגי בין שני מחשבים ברשת. שני יישומים מחוברים כביכול דרך ערוץ ישיר.

□ פרוטוקולי התעבורה רצים במחשבי הקצה: הצד השולח מחלק את המסרים לסגמנטים ומעביר אותם לרמת הרשת. הצד המקבל מרכיב מחדש את הסגמנטים למסרים ומעביר אותם לרמת היישום.

□ קיימים מספר פרוטוקולי תעבורה, העיקריים באינטרנט הם **TCP** ו- **UDP**

# רמת הרשת לצומת רמת התעבורה

□ **רמת הרשת:** העברת הודעות בין שני מחשבים.

□ **רמת התעבורה:** העברת תקשורת בין יישומים, בין היישום השולח ליישום המקבל המתאים. רמת התעבורה היא מעל רמת הרשת.

□ **אנלוגיה:** קחו לדוגמא שני בתים האחד בתל-אביב והשני בירושלים. בכל בית עשרה ילדים. כל ילד בכל בית שולח מכתב לכל אחד מהילדים בבית השני, כל מכתב במעטפה נפרדת, ולכן מכל בית נשלחים 100 מכתבים לבית השני. בכל בית ישנו ילד שתפקידו לאסוף את המכתבים לשליחה מאחיו, להעביר אותם לסניף הדואר ולחלק את המכתבים המגיעים. שירות הדואר מספק קשר לוגי בין בתים.

□ מסרים = מכתבים במעטפות.

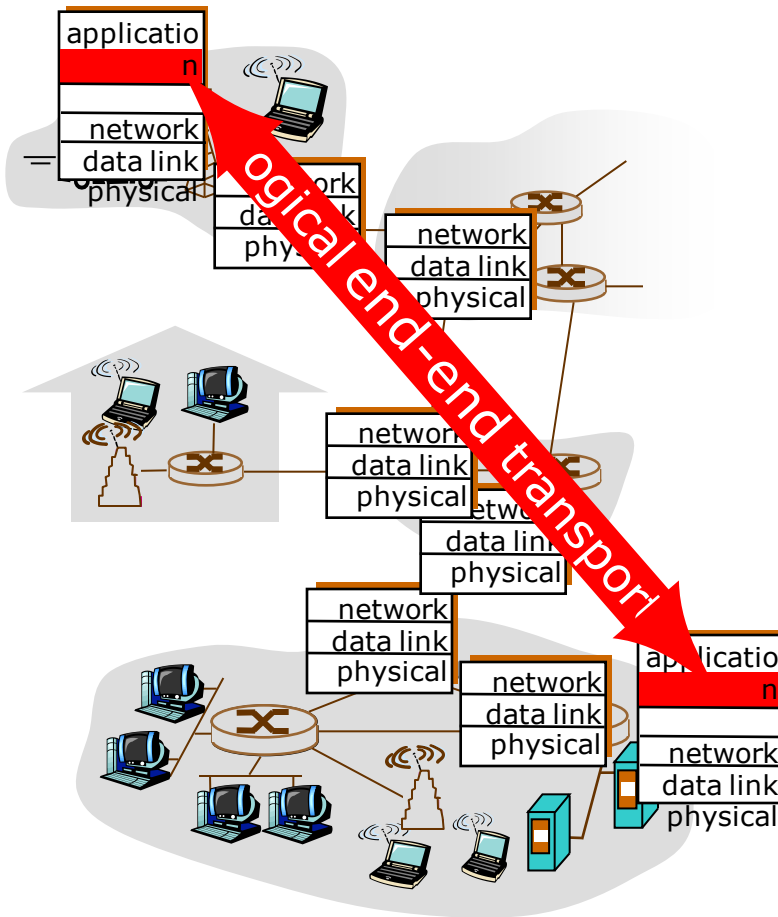
□ תהליכים = ילדים.

□ מחשבי קצה (hosts) = בתים.

□ פרוטוקול התעבורה = הילד שתפקידו לאסוף ולחלק מכתבים.

□ פרוטוקול הרשת = שירות הדואר, כולל סניפי הדואר.

# פרוטוקולי תעבורה באינטרנט



■ **שירות תעבורה אמין TCP** - שירות שבו הודעות מועברות ומתקבלות בסדר הנכון מבלי ללכת לאיבוד. מאפיינים:

- בקרת עומס
- בקרת זרימה
- הקמת קשר במנגנון Handshake.

■ **שירות תעבורה לא אמין UDP**:

- **Best effort** - ייתכן איבוד הודעות, הגעתן שלא על פי הסדר והכפלתן. בפרוטוקול זה יש הרחבה מינימלית של רמת ה-IP.

■ **שירותים שאינם קיימים באינטרנט:**

- הבטחת רוחב פס מינימלי
- הבטחת השהייה מינימלית.
- כדי לתמוך ביישומי זמן אמת, כמו טלפוניה, צריך לתמוך בשירותים האלו.

# שכבת ההובלה

---

- שרותי שכבת ההובלה.
- ריבוב/פילוג.
- עקרונות של בקרת עומסים.
- תעבורה חסרת קשר: UDP.
- בקרת עומסים TCP.
- עקרונות תעבורת מידע  
אמינה ברשת.
- תעבורה מונחת קשר: TCP
  - מבנה המקטע.
  - תעבורת מידע אמינה.
  - בקרת זרימה.
  - ניהול הקשר



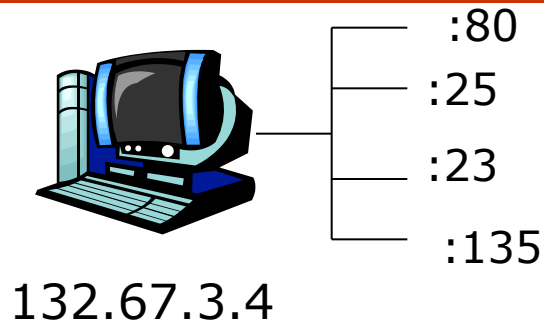
# ריבוק (Multiplexing) / פילוף (de-multiplexing)

- מאפשר תמיכה במספר יישומים באותו מחשב קצה.
- **Multiplexing**: המחשב השולח מוסיף למידע header. (זיהוי socket המקור והיעד לצורך demultiplexing במחשב המקבל).
- **De-multiplexing**: המחשב המקבל מעביר את הסגמנטים המתקבלים אל ה- socket הנכון, לפי המידע ב- header.
- **סיכום**: התפקיד החשוב שמבצעת שכבת ההעברה הוא ריבוב (Multiplex) הנתונים השונים שמגיעים מהיישומים השונים על גבי אפיק אחד, והיכולת להפרידם (DeMultiplex) בהתקן היעד.

□ אנלוגיה בין האינטרנט לרשת טלפוניה:

<u>אינטרנט</u>	<u>טלפון</u>
כתובת IP	מספר טלפון
Socket	שקע ומכשיר טלפון
מספר port	מספר שלוחה

# כתובות אינטרנט



## □ מיעון ברמת הרשת (IP)

### ■ כתובת IP לכל תחנה ברשת

□ ארבעה בתים, סה"כ 32 ביט

□ אמור להיות ייחודי לכל תחנה

□ לדוגמא: 132.67.3.4

## □ מיעון ברמת התחנה (Host)

### ■ Port

□ לכל תחנה 64K פורט-ים

□ חלק מהפורטים שמורים לשימוש יישובים מיוחדים

■ 21 – FTP

■ 80 – HTTP (גלישה)

■ 25 – SMTP (אימייל)

# Port Mechanism

- כל תהליך או אפליקציה המשדרים ב- UDP או ב- TCP מזוהים באמצעות מספר חד-חד ערכי (בכל Host בן 16 ביט), סה"כ - 64k פורטים.
- Ports 1-1023 מוגדרים Well Known Ports ומנוהלים ע"י IANA (רשות האינטרנט להקצאת מספרים The Internet Assigned Numbers Authority)  
ניתן למצואם ב: rfc 1700 -
- FTP: 21, Telnet: 23, SMTP: 25, DNS: 53, HTTP: 80, POP-3: 110...
- Registered Ports נקראים Ports 1024-49151
- Dynamic and/or Private ports נקראים Ports 49152-65535
- מרבית האימפלמנטציות מקצות ל Clients - פורטים 1024 - 5000.
- Socket: הצירוף של IP Address ושל ה- Port - נותן מזהה חד-חד ערכי לאפליקציה ברשת.

**Local IP: 111.22.3.4**  
**Local Port: 2249**



**Remote IP: 123.45.6.78**  
**Remote Port: 3726**

# SOCKET – *משק*

---

□ Connection מזוהה ע"י צמד Sockets

■ {TCP, 129.5.32.11, 1026, 129.5.1.1, 21}

יזוהה קשר בין לקוח FTP בכתובת 129.5.32.11 לבין שרת FTP בכתובת 129.5.1.1.

□ קיימים 3 סוגי Sockets :

■ Stream Socket : קשר מהימן קצה לקצה (TCP).

■ Datagram Socket : קשר Connectionless (UDP).

■ Raw Socket : גישה ישירה לפרוטוקולים ברמות נמוכות יותר.

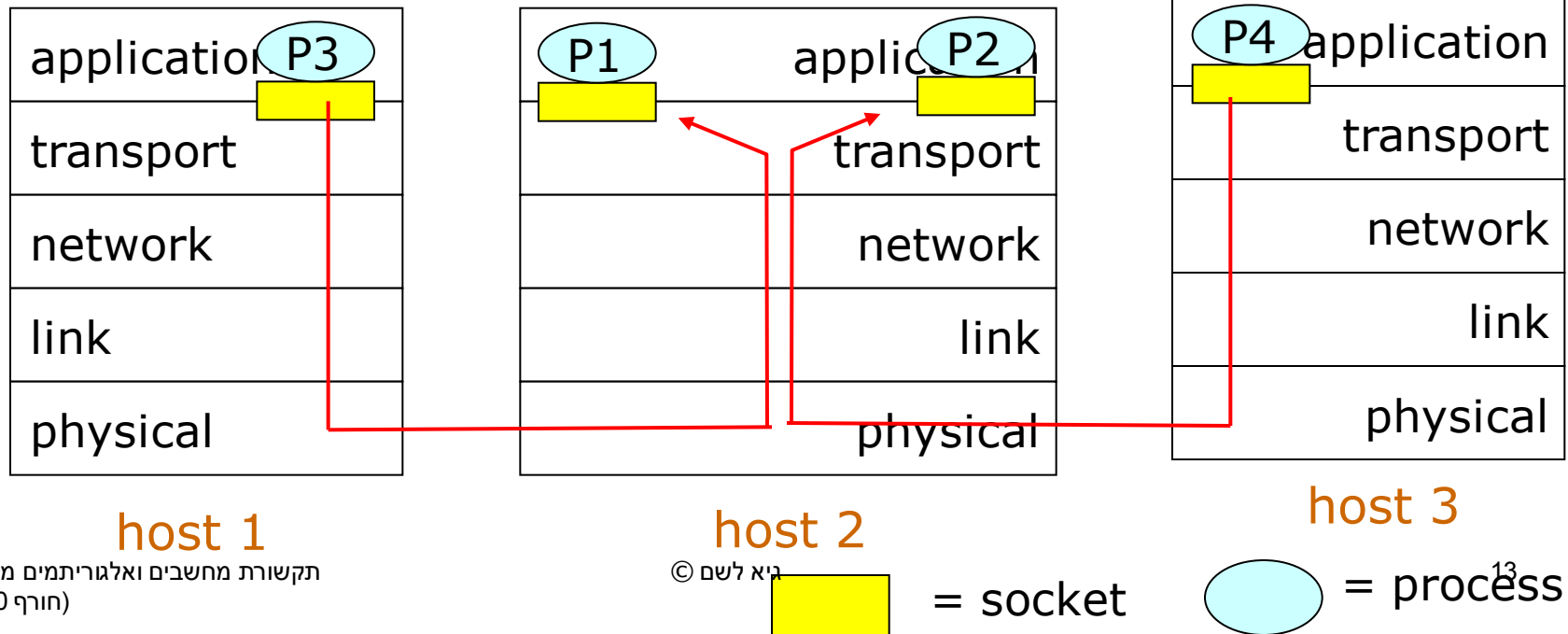
# ריבוק (Multiplexing) / פילוף (de-multiplexing)

## Demultiplexing at rcv host:

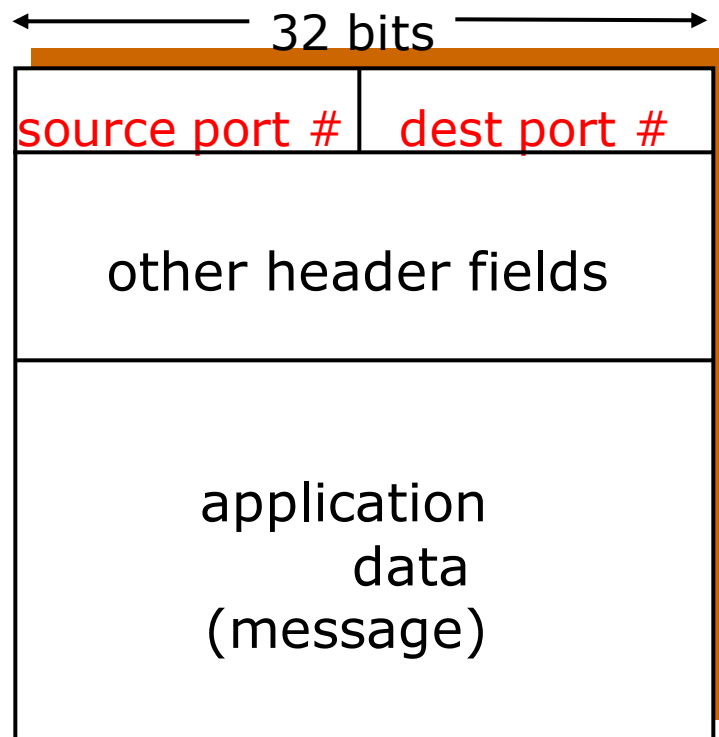
המחשב המקבל מעביר את הסגמנטים המתקבלים אל ה- socket הנכון, לפי המידע ב- header.

## Multiplexing at send host:

המחשב השולח מוסיף למידע header. (זיהוי socket המקור והיעד לצורך demultiplexing במחשב המקבל).



# De-multiplexing - איך? כואף?



TCP/UDP segment format

- למחשב המקבל מגיעות חבילות IP. לכל חבילה יש כתובת IP מקור ויעד.
- לכל חבילה יש port מקור ויעד.
- הנמען מתעניין בעיקר בשדה Port יעד. לפי כתובת ה IP ו-port היעד,
- רמת התמסורת במחשב יודעת להפנות את החבילה ל- socket המתאים.
- Port המקור משמש לצורך שליחת תשובה.

# De-multiplexing פרוטוקול UDP

□ יצירת socket עם מספרי port מתאימים: המחשב השולח צריך לדעת את מספר ה-port אליו יש לפנות במחשב המקבל כדי לדבר עם אפליקציה מסוימת.

□ Socket ה-UDP מזהה ע"י שני שדות:

■ .dest IP address

■ .dest port number

□ כאשר הנמען מקבל את חבילת ה-UDP:

■ בודק את מס' port היעד.

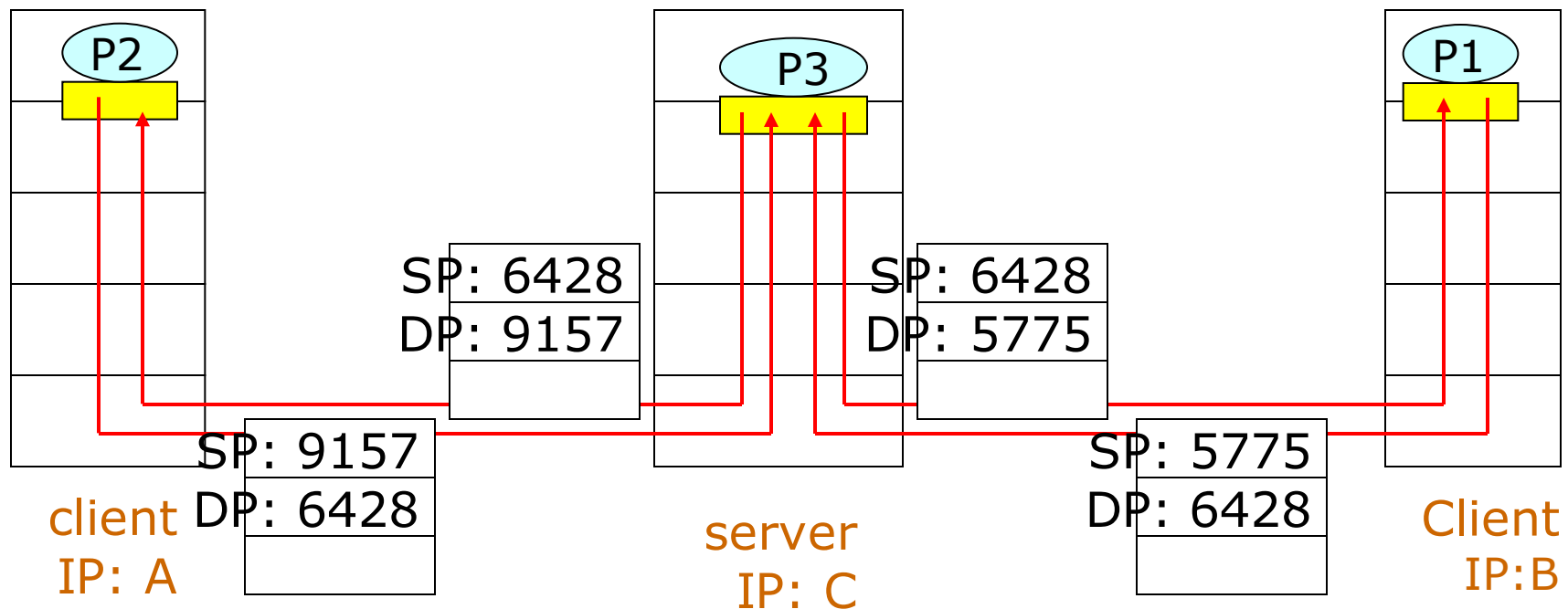
■ מכוון את החבילה ל-socket עם מס' ה-port המתאים.

□ לאותו socket מופנות

חבילות ממחשבי מקור שונים בעלי כתובת IP שונה ומס' Port מקור שונה.

# UDP פרוטוקול De-multiplexing

```
DatagramSocket serverSocket=new DatagramSocket(6428);
```



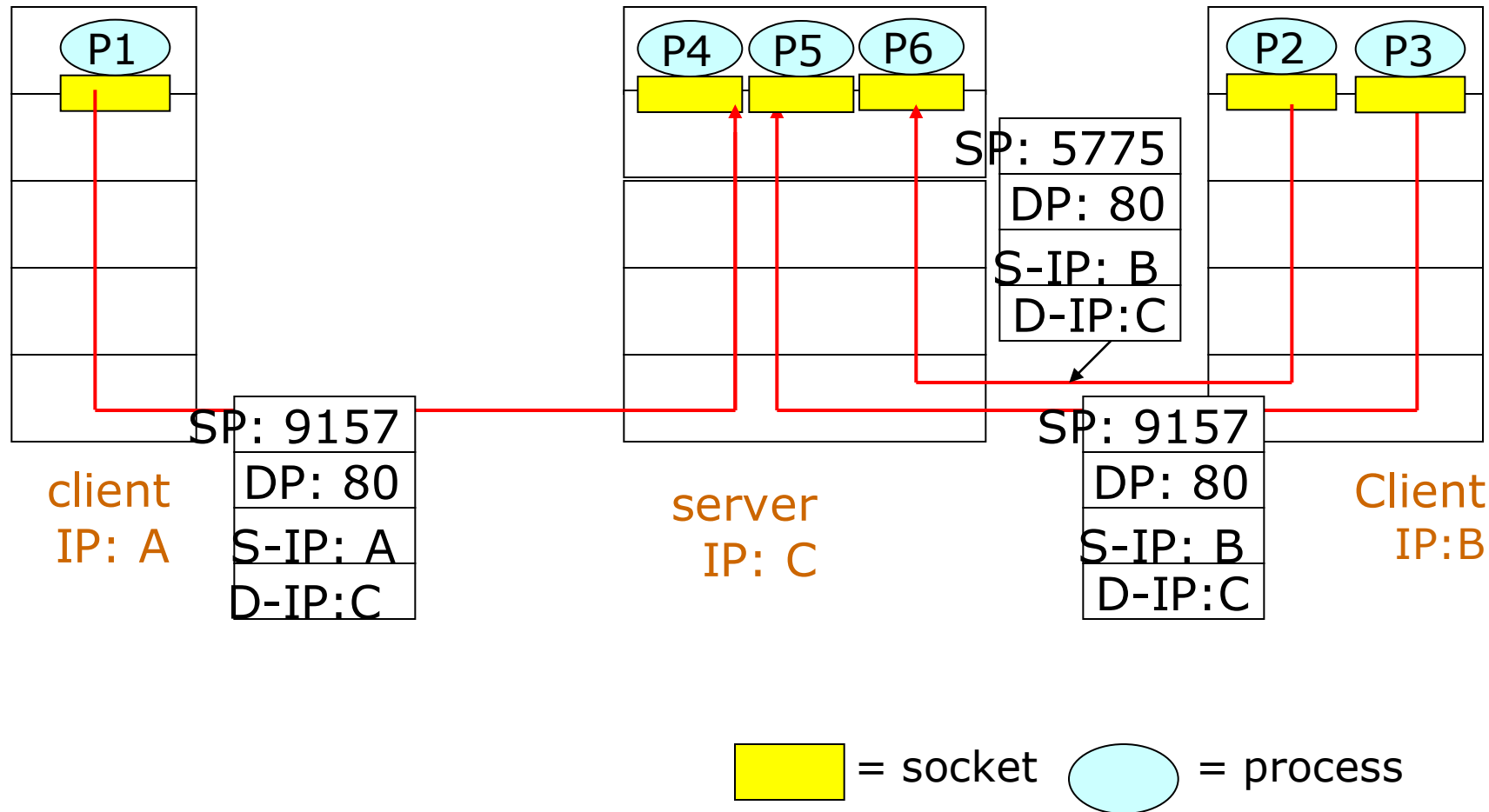
SP provides "return address"   = socket   = process



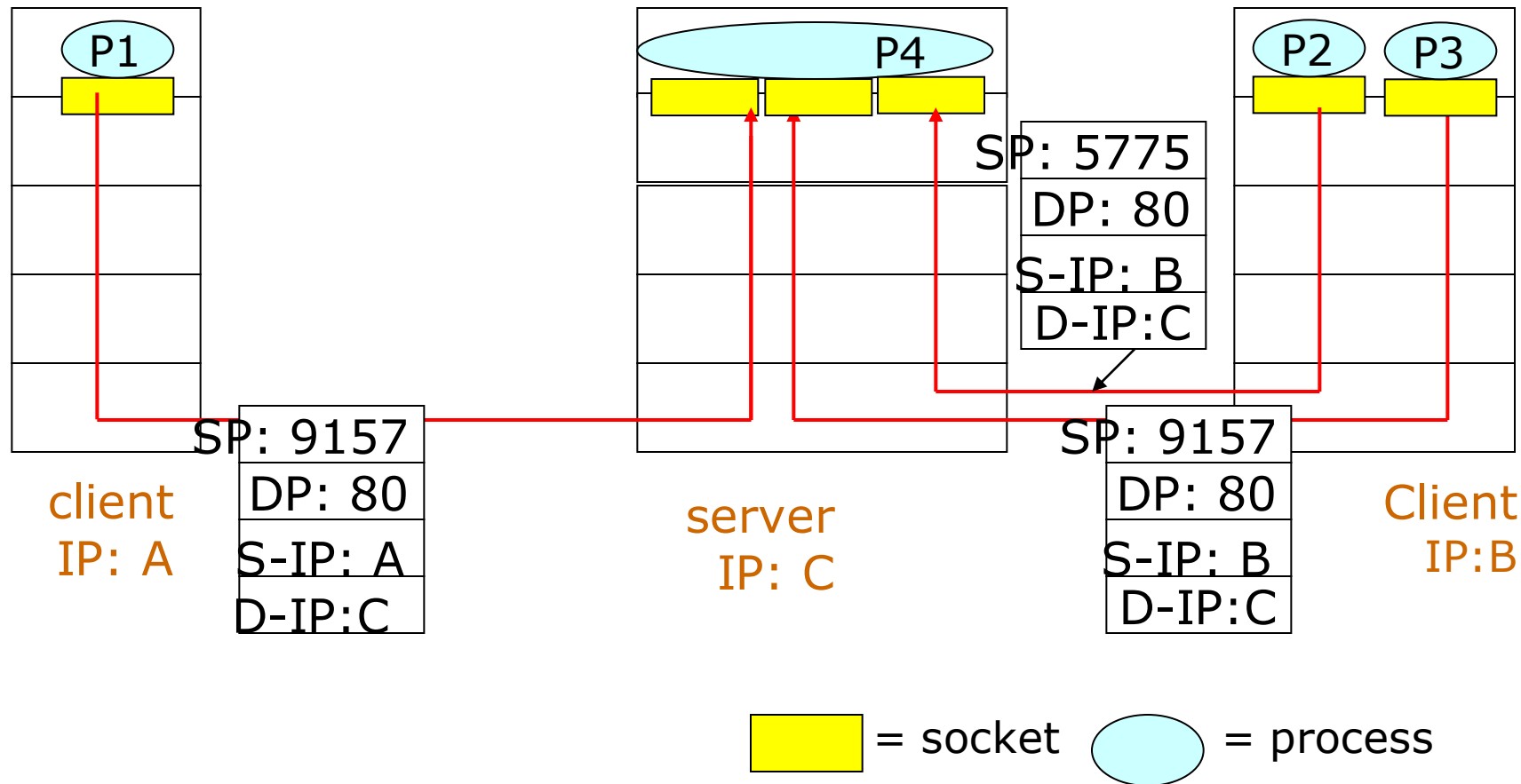
# TCP De-multiplexing פרוטוקול

- ב TCP ה- socket מזהה ע"י 4 שדות:
  - כתובת IP יעד.
  - מס' port יעד.
  - כתובת IP מקור.
  - מס' port מקור.
- המחשב המקבל משתמש בכל ארבעת השדות כדי להפנות את החבילה ל- socket המתאים.
- המחשב המקבל יכול לתמוך בריבוי sockets של TCP. כל אחד מזהה ע"י 4 שדות משלו.
- בשרתי Web יש socket נפרד לכל לקוח.
- בחיבור HTTP שהוא non-persistent (לא נשמרת עקביות הקשר) יש socket נפרד לכל בקשה.

# TCP פירוק De-multiplexing



# De-multiplexing פרוטוקול TCP: שרת WEB



# שכבת ההובלה

---

- שרותי שכבת ההובלה.
- ריבוב/פילוג.
- תעבורה חסרת קשר: UDP.
- עקרונות תעבורת מידע אמינה ברשת.
- תעבורה מונחת קשר: TCP
  - מבנה המקטע.
  - תעבורת מידע אמינה.
  - בקרת זרימה.
  - ניהול הקשר
- עקרונות של בקרת עומסים.
- בקרת עומסים TCP.

# תצורה חסרת קשר (connectionless) UDP

## מדוע צריך את פרוטוקול UDP?

- חסכון התקורה ביצירת החיבור.
- פרוטוקול פשוט – חשוב בשביל מכשירים פרימיטיביים, כמו מכשיר אזהקה.
- בעל header קטן – הודעות קצרות יותר.
- אין בקרת דחיסה ועומס – גורם להעמסת הרשת ואפשרות לניצולת מקסימלית.

□ פרוטוקול תעבורה פשוט.

□ חבילות UDP יכולות ללכת לאיבוד או להגיע של על פי הסדר (**Best Effort**).

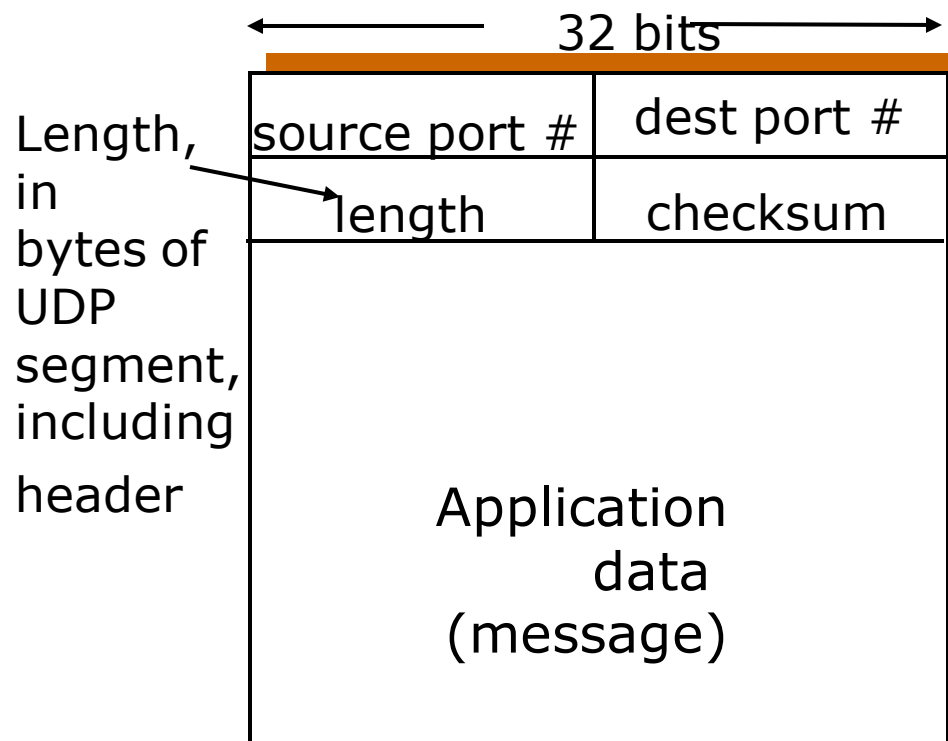
□ *connectionless*:

■ ללא לחיצת יד,

■ כל חבילת UDP מטופלת בנפרד.

■ חסכון ביצירת החיבור.

# UDP פורמט



UDP segment format

משמש ליישומי מולטימדיה, בהם אפשרי לאבד חבילות ובהם חשוב קצב מהיר של העברה, יש עדיפות למהירות על פני אמינות.

משמש גם בשליחת הודעות DNS (**Domain Name System**) : אלו הודעות קטנות ואין חשיבות לסדר בין חבילות שונות. וגם משמש בפרוטוקול SNMP שהוא פרוטוקול לניהול רשת.

# UDP Checksum: איפוי שגיאות

מטרה: מציאת שגיאות פשוטות בלבד ללא מנגנון לתיקון שגיאות.

## שולח:

□ מחלק את הסגמנט

לקטעים של 16 ביט.

□ מחבר אותם ומבצע

השלמה ל-1.

□ את התוצאה שם בשדה

ה-checksum.

## המקבל:

□ מחשב את ה-

checksum של הסגמנט

שקיבל.

□ בודק אם המחושב שווה

לזה שהתקבל:

■ אם הוא שונה התגלתה

טעות.

■ אם זהה לא התגלתה טעות

אבל ייתכן שהייתה טעות.

# דוגמא $f$ -Checksum באינטרנט

שנים לב: □

כאשר מחברים מספרים, אם יש נשא סופי חבר אותו אל התוצאה (נשא מעגלי).

דוגמא: חיבור של 2 מספרים בעלי 16 ביטים. □

	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
נשא מעגלי	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1



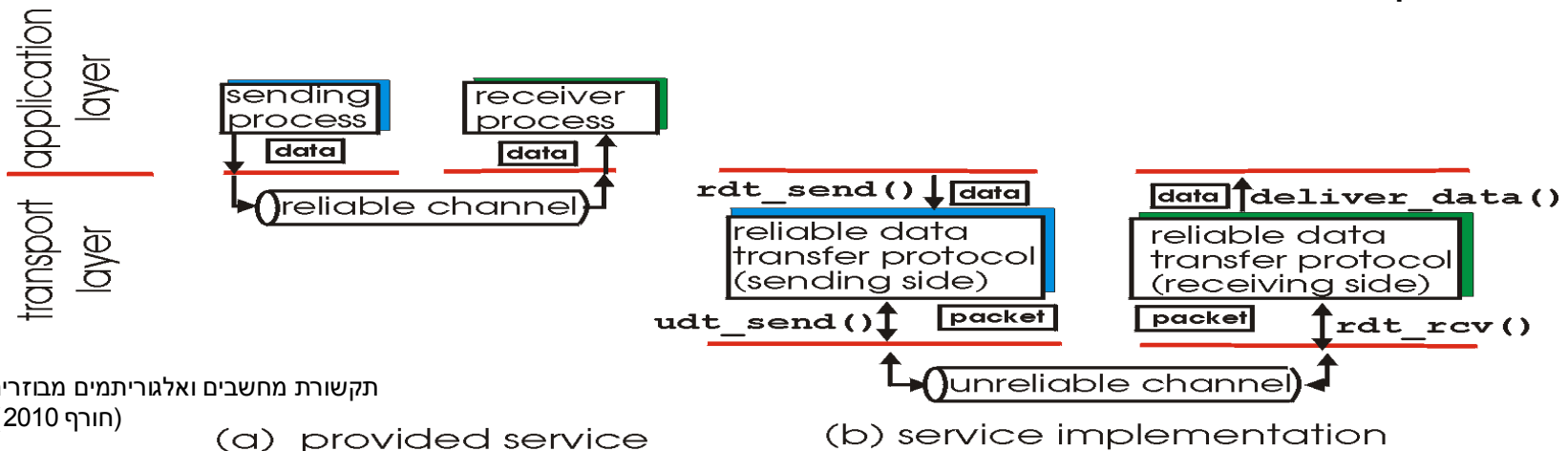
# שכבת ההובלה

---

- שרותי שכבת ההובלה.
- ריבוב/פילוג.
- תעבורה חסרת קשר: UDP.
- עקרונות תעבורת מידע  
אמינה ברשת.
- תעבורה מונחת קשר: TCP
  - מבנה המקטע.
  - תעבורת מידע אמינה.
  - בקרת זרימה.
  - ניהול הקשר
- עקרונות של בקרת עומסים.
- בקרת עומסים TCP.

# א'ק אפצ'ס הפטחת אמינות דרך צרוף רשת שאינם אמינות?

- חשיבות האמינות היא ברמות שונות ולא רק ברמת התעבורה. יש לה חשיבות גם ברמת הערוץ והיישום. שליחת מידע מתבצעת דרך רמה תחתונה שאינה אמינה, דבר שיכול לגרום לשיבוש הודעות.
- ניתן לזהות שגיאה באמצעות קוד ושדות הסתברותיים (כמו Checksum). ישנו סיכוי ששגיאה לא תתגלה, אך הוא זניח.
- ככל שהתווך יותר משובש כך על פרוטוקול התעבורה האמין, rdt (reliable data transfer protocol) להתמודד עם יותר מצבים-דבר שמעלה את מורכבות הפרוטוקול.
- נבצע סקירה של פרוטוקולים, כל אחד מהם מורכב יותר מקודמו, עד שנגיע לפרוטוקול האמין ביותר.

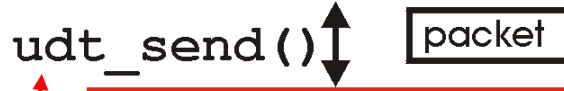
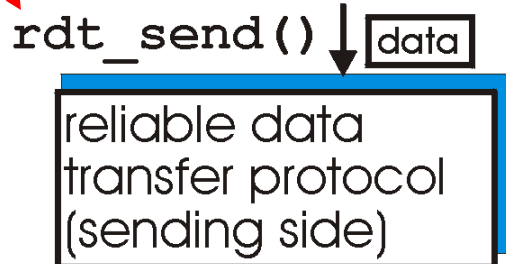


# תצורה חז'א א'נ'ה

**rdt\_send()** : called from above, (e.g., by app.). Passed data to

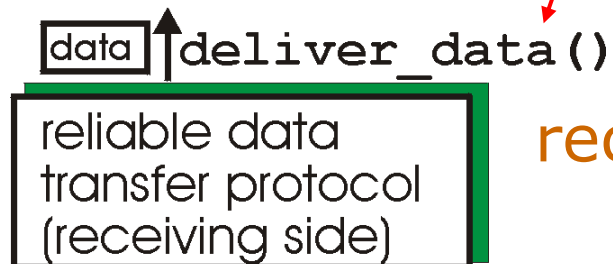
deliver to receiver upper layer

send  
side

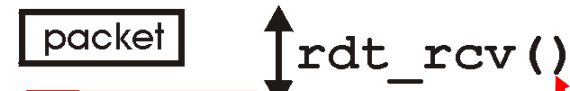


**udt\_send()** : called by rdt, to transfer packet over unreliable channel to receiver

**deliver\_data()** : called by rdt to deliver data to upper



receive  
side



**rdt\_rcv()** : called when packet arrives on rcv-side of channel



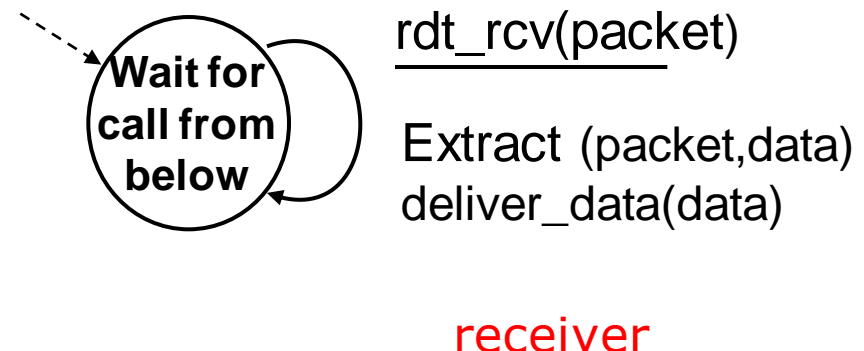
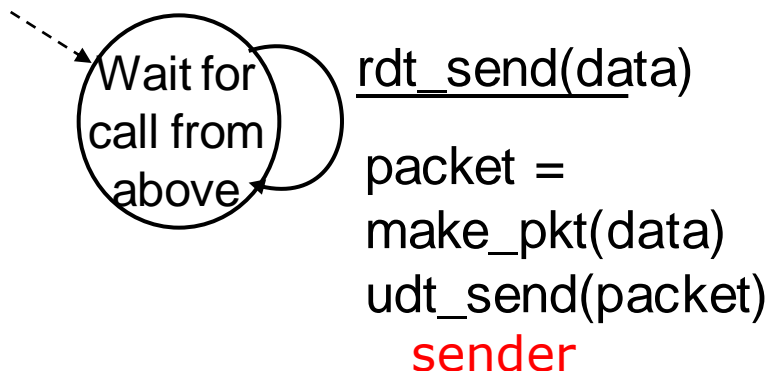
## תצבורת מידע אמנה

- בסדר חשיבות עולה - נדרש לפתח את הצדדים השולח, המקבל של פרוטוקול להעברת נתונים אמינים (rdt).
- נתיחס רק להעברת נתונים חד כווני
  - אך השליטה תהיה על זרימת המידע בשני הכיוונים!
- נשתמש במכונת מצבים סופית להצגת השולח והמקבל.



# Rdt1.0: פרוטוקול פשוט שניתן שהצד השני יקבל את הנתונים.

- לשולח ולמקבל יש רק מצב אחד - המתנה לקבלת הודעה.
- **הצד השולח** מחכה לקבלת הודעה מהרמה העליונה. כאשר מתרחש מאורע של שליחת הודעה מתבצעת פעולת שליחת ההודעה, תוך העברתה לרמה התחתונה.
- **הצד המקבל** מחכה לקבלת הודעה מהרמה התחתונה. כאשר מתרחש מאורע של קבלת הודעה - המקבל מוציא מהחבילה את המידע.
- סיכום:
- הערוץ מוגדר להיות אמין ואין בדיקת corrupt.
- כל שגיאה שכן תתרחש תגיע ישר לאפליקציה.
- אין אפשרות לתקיעה של הפרוטוקול.



## Rdt2.0: ערוץ תקשורת לא אמין

□ באופן עקרוני ערוץ תקשורת יכול "להעיף" סיביות מהחבילות במהלך השידור.

■ מספר הנחות:

□ אין אובדן חבילות (אבל ייתכן שיגיעו משובשות).

□ החבילות מגיעות בסדר הנכון.

□ כל הטעויות מתגלות על ידי מנגנון ה-checksum.

□ שאלה: איך מתאוששים מטעויות?

■ קיים היזון חוזר מהצד המקבל לצד השולח:

□ **ACK** (acknowledgment): אישור על קבלת חבילה תקינה.

□ **NAK** (negative acknowledgment): אישור על קבלת הודעה משובשת.

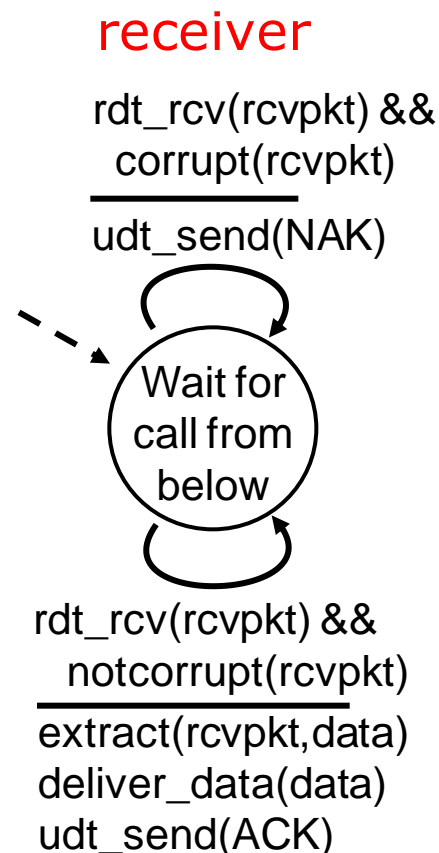
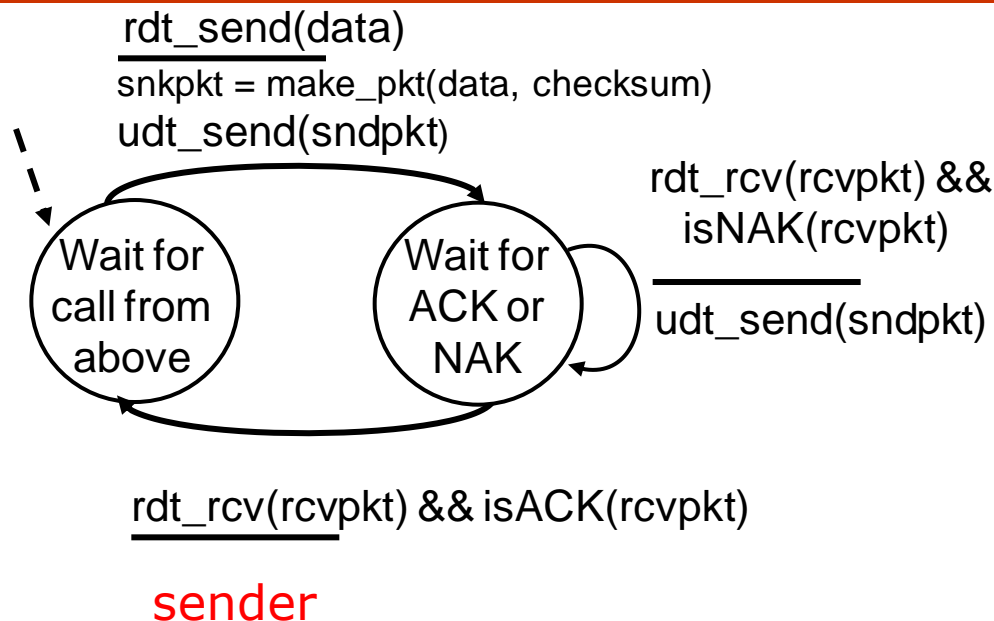
□ חבילה שהגיעה משובשת תשלח מחדש (retransmission).

□ תוספות שלא היו קיימות ב-Rdt1.0:

■ גילוי טעויות.

■ היזון חוזר מהצד המקבל לשולח (ACK, NAK).

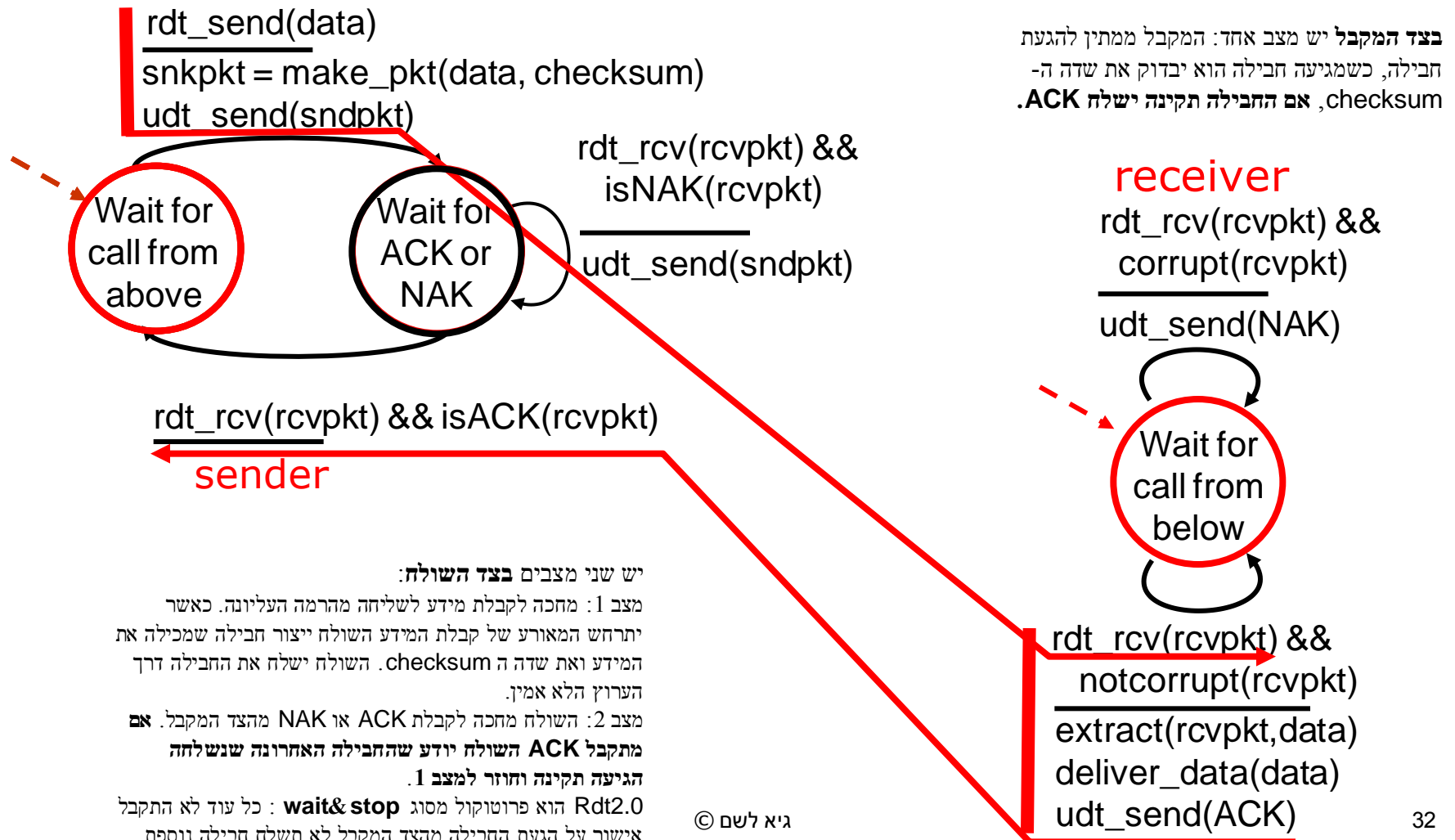
# רדט2.0 דיאגרמת המצבים



## בעיות אפשריות:

- חבילה שהתקבלה משובשת ולא התגלתה תגיע לאפליקציה כחבילה תקינה.
- NAK שהשתבש והגיע כ ACK יגרום לאיבוד חבילה.
- ACK שהשתבש והגיע כ NAK יגרום לשליחה חוזרת מיותרת, וגם להעברת חבילה משוכפלת לאפליקציה

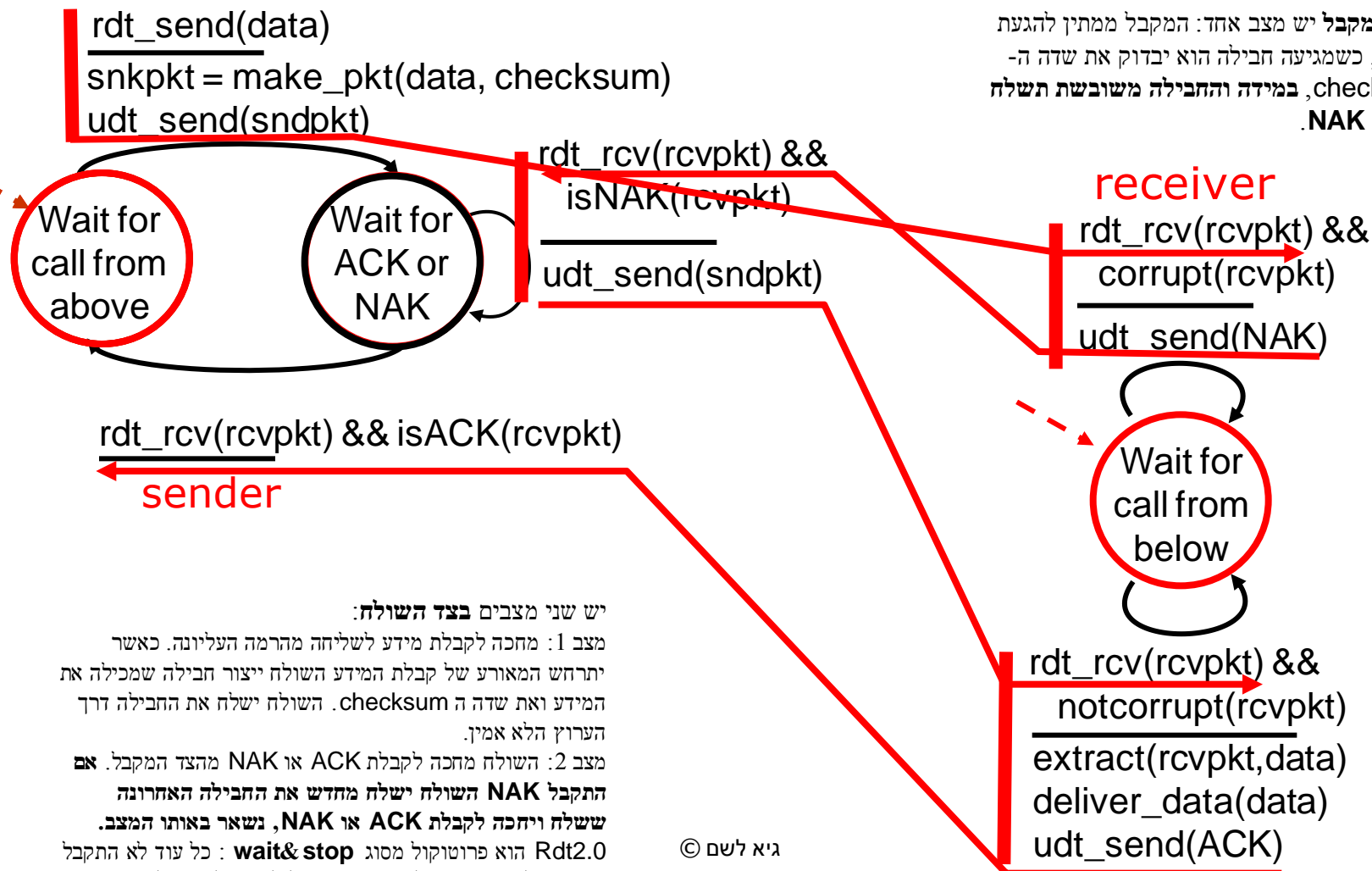
# פאזות rdt2.0 כ'ce kff





# פצולת rdt2.0 עם נאכה

בצד המקבל יש מצב אחד: המקבל ממתין להגעת חבילה, כשמגיעה חבילה הוא יבדוק את שדה ה-checksum, במידה והחבילה משובשת תשלח הודעת NAK.



יש שני מצבים בצד השולח:

מצב 1: מחכה לקבלת מידע לשליחה מהרמה העליונה. כאשר יתרחש המאורע של קבלת המידע השולח ייצור חבילה שמכילה את המידע ואת שדה ה-checksum. השולח ישלח את החבילה דרך הערוץ הלא אמיין.

מצב 2: השולח מחכה לקבלת ACK או NAK מהצד המקבל. אם התקבל NAK השולח ישלח מחדש את החבילה האחרונה ששלח ויחכה לקבלת ACK או NAK, נשאר באותו המצב.

Rdt2.0 הוא פרוטוקול מסוג **wait&stop**: כל עוד לא התקבל אישור על הגעת החבילה מהצד המקבל לא תשלח חבילה נוספת.

# *rdt2.0 יש ליקוי חמור!*

על פניו נראה כי הפרוטוקול מבטיח אמינות אבל יש בו פגם חמור

מה קורה אם ACK/NAK

פגומים ?

□ השולח לא יודע מה קרה אצל המקבל.

□ השולח אינו יכול פשוט לשלוח מחדש את החבילה כי אז יכול להיווצר מצב של שכפול

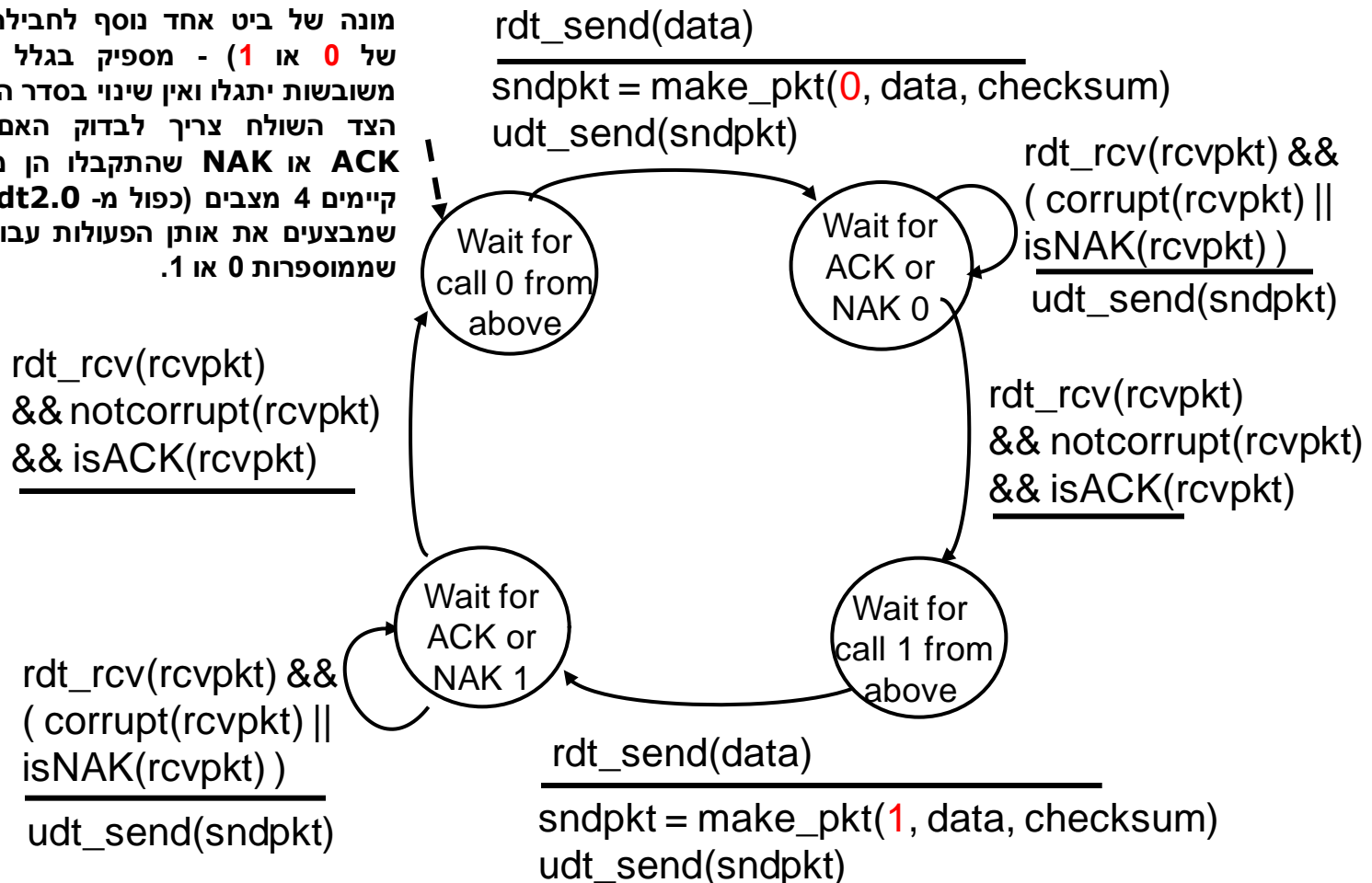
טיפול בשיכפול:

□ אם יהיה שיבוש הודעת NAK אז חבילה שהגיעה משובשת לא תשלח שוב כנדרש ויהיה איבוד הודעה.

□ אם יהיה שיבוש הודעת ACK אז תשלח אותה ההודעה פעמיים.

# rdt2.1: הצד השולח, טיפול ACK/NAKs - מסופים

מונה של ביט אחד נוסף לחבילה (מספור של 0 או 1) - מספיק בגלל שחבילות משובשות יתגלו ואין שינוי בסדר החבילות. הצד השולח צריך לבדוק האם הודעות ACK או NAK שהתקבלו הן משובשות. קיימים 4 מצבים (כפול מ-Rdt2.0), מפני שמבצעים את אותן הפעולות עבור חבילות שממוספרות 0 או 1.



# rdt2.1: הצד המקבל, טיפול ACK/NAKs - מסופים

צריך לבדוק האם התבילה שהתקבלה משוכפלת, על פי המספר הסיידורי של ההודעה האחרונה שהתקבלה (0 או 1).

אם ההודעה הגיעה משוכפלת הפרוטוקול ידע לא להעביר אותה לרמת היישום. המקבל לא יודע אם הודעת ה- ACK או NAK האחרונה ששלח התקבלו בצורה תקינה בצד השולח.

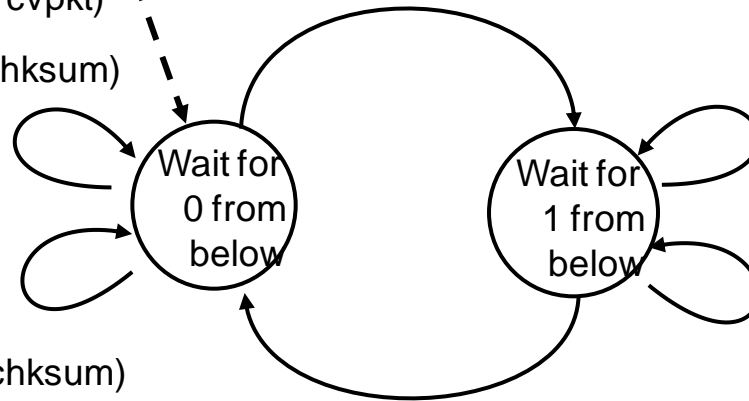
rdt\_rcv(rcvpkt) && (corrupt(rcvpkt)  
sndpkt = make\_pkt(NAK, chksum)  
udt\_send(sndpkt)

rdt\_rcv(rcvpkt) &&  
not corrupt(rcvpkt) &&  
has\_seq1(rcvpkt)

sndpkt = make\_pkt(ACK, chksum)  
udt\_send(sndpkt)

rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)  
&& has\_seq0(rcvpkt)

extract(rcvpkt,data)  
deliver\_data(data)  
sndpkt = make\_pkt(ACK, chksum)  
udt\_send(sndpkt)



rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)  
&& has\_seq1(rcvpkt)

extract(rcvpkt,data)  
deliver\_data(data)  
sndpkt = make\_pkt(ACK, chksum)  
udt\_send(sndpkt)

rdt\_rcv(rcvpkt) && (corrupt(rcvpkt)  
sndpkt = make\_pkt(NAK, chksum)  
udt\_send(sndpkt)

rdt\_rcv(rcvpkt) &&  
not corrupt(rcvpkt) &&  
has\_seq0(rcvpkt)

sndpkt = make\_pkt(ACK, chksum)  
udt\_send(sndpkt)

## צק/ר 2.1 RDT:

- חבילה שבה ה- data התקבל משובש, ולא התגלתה, תגיע לאפליקציה כחבילה תקינה. (האפליקציה תקבל data שגוי).
- מספר סידורי שהשתבש:
  - אם הייתה מתגלה ע"י מנגנון השגיאות, החבילה תטופל כ- corrupted, ויישלח NAK.
  - אם החבילה לא מתגלה, החבילה לא תטופל כי מצפים למספר אחר. לפי הפרוטוקול אין שליחת NAK, ואז הפרוטוקול ייתקע.
- NAK שהשתבש והגיע כ- ACK יגרום לתקיעת הפרוטוקול (רק הצד השולח יעבור למספר הבא).
- ACK שהשתבש והגיע כ- NAK יגרום לתקיעת הפרוטוקול (רק הצד המקבל יעבור למספר הבא).

## rdt2.1: דיון

### השולח:

- מספר סידורי (מיספור) נוסף לחבילה.
- 2 אפשרויות מספור 0 או 1 יספיקו.
- חייבים לבדוק אם התקבל ACK / NAK פגום.
- פעמיים מספר המצבים:
  - המצב חייב "לזכור" אם לחבילה הנכחית יש 0 או 1.

### המקבל:

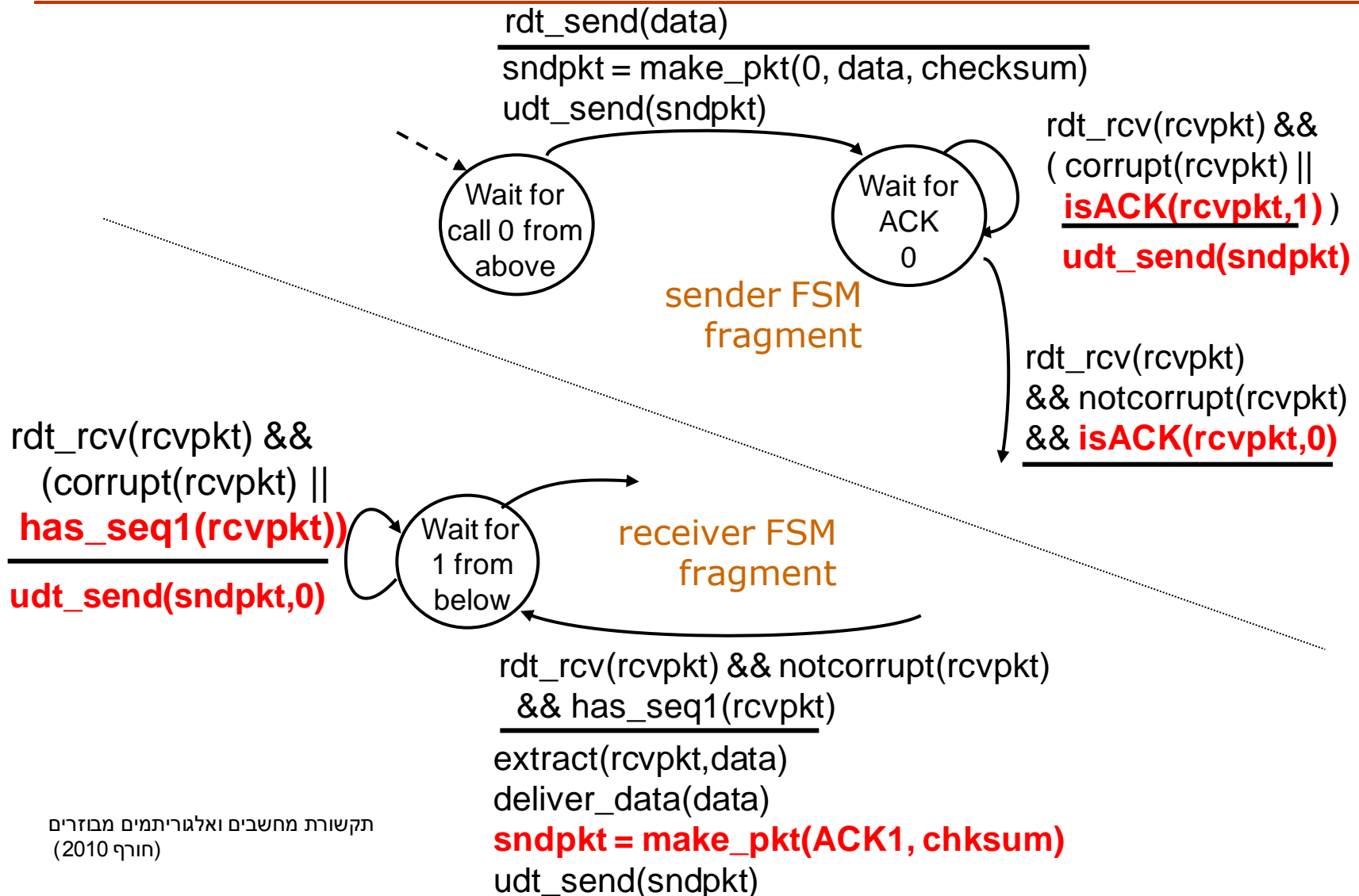
- חייב לבדוק האם החבילה שהתקבלה משוכפלת.
  - המצב מצביע האם המספר הסדרתי שהתקבל הוא 0 או 1.
- שים לב: המקבל אינו יכול לדעת האם ה-ACK/NAK האחרון שהתקבל בשולח הוא תקין.

## NAK *kff* :rdt2.2

---

- הפרוטוקול הוא כמו rdt2.1 אך ללא שימוש בהודעות NAK.
- במקום NAK המקבל שולח ACK להודעה האחרונה שהתקבלה בצורה תקינה. המקבל חייב לצרף את מספרה הסידורי של החבילה שעליה הוא נותן ACK.
- שיפור לעומת rdt2.1: מונע שכפולי חבילות ו-ACK.
- אם יש שכפול של הודעת ACK אז השולח ישלח מחדש את החבילה הנוכחית ולא יהיה איבוד מידע.
- אין טיפול באובדן חבילות, ייתכן מצב בו המקבל והשולח יחכו זמן אינסופי.

## rdt2.2: sender, receiver fragments





## רצור RDT 2.2

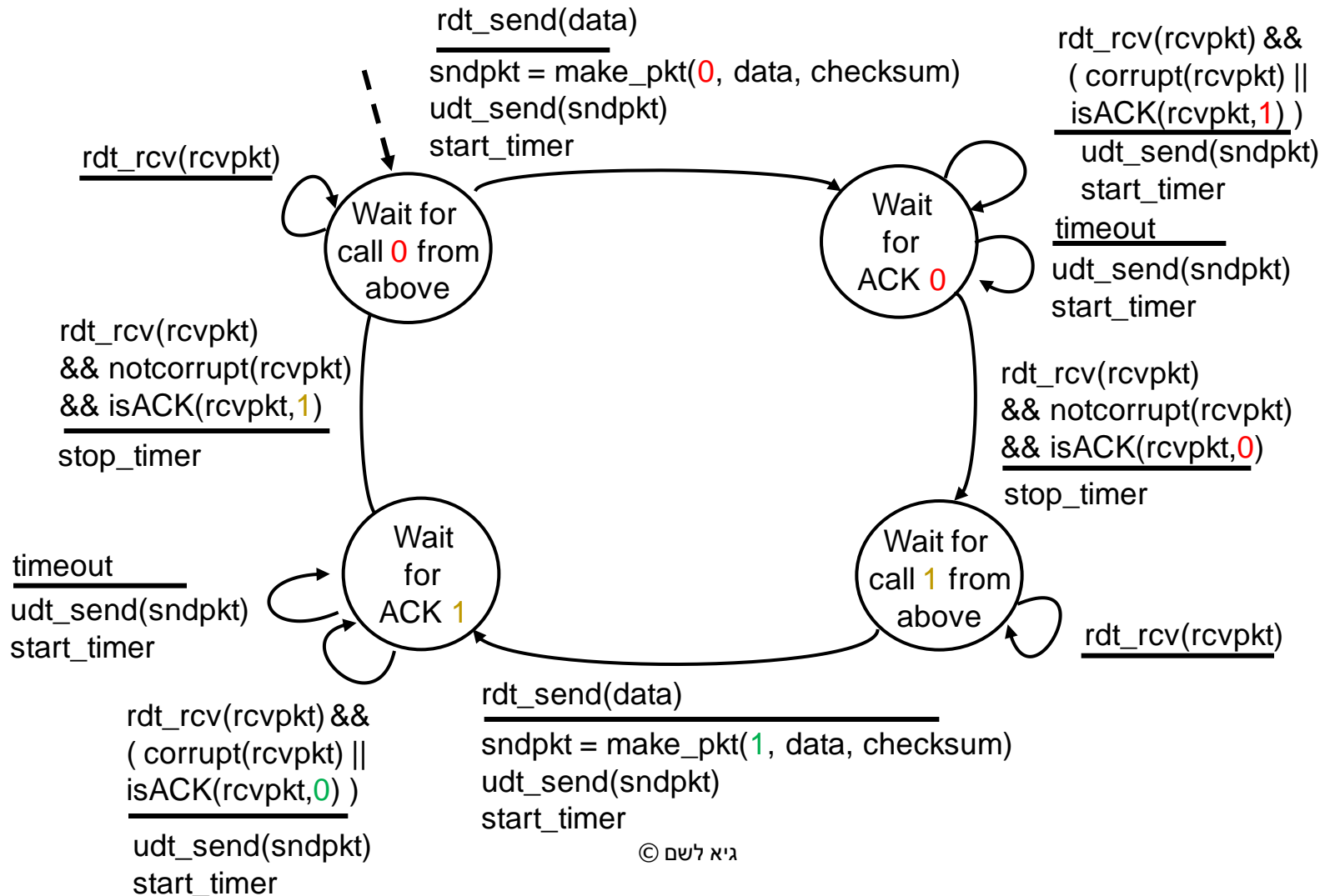
- חבילה שבה ה-data התקבל משובש, ולא התגלתה, תגיע לאפליקציה כחבילה תקינה (האפליקציה תקבל data שגוי).
- מספר סידורי שהשתבש :
  - אם הייתה מתגלה ע"י מנגנון השגיאות, החבילה תטופל כ-corrupted, ויישלח ACK על חבילה קודמת.
  - אם החבילה לא מתגלה, החבילה לא תטופל כי מצפים למספר אחר. לפי הפרוטוקול אין שליחת ACK על חבילה קודמת, ואז הפרוטוקול ייתקע.
- **ACK על חבילה קודמת שהשתבש ל ACK על חבילה נוכחית**  
יגרום לתקיעת הפרוטוקול (רק הצד השולח יעבור למספר הבא)
- **ACK על חבילה נוכחית שהשתבש ל ACK על חבילה קודמת**  
יגרום לתקיעת הפרוטוקול (רק הצד המקבל יעבור למספר הבא)

## rdt3.0: ערוץ עם שגיאות ואובדן

- פרוטוקול שניתן להריץ מעל ערוצים אמיתיים. מתאים לערוצים פיזיים כמו רשת מקומית.
- הפרוטוקול מטפל ב: שגיאות, שכפולים, אובדן, אין טיפול בשינוי סדר של הודעות.
- הנחות:
  - הערוץ יכול לאבד חבילות (מידע או ACK).
  - קיימים המנגנונים מהפרוטוקולים הקודמים: Checksum, מס' סידורי של חבילה, ACK, שליחה מחודשת. אבל הם אינם מספיקים.
- דרך ההתמודדות עם אובדן חבילות:
  - **Timer**: השולח מחכה לACK פרק זמן "סביר" אשר קטן מהעיכוב המקסימלי (Round Trip Time) RTT. במידה ולא התקבל ACK בזמן הזה תתבצע שליחה מחודשת. במידה והיה רק עיכוב ולא אובדן, יתבצע שכפול שבו הצד המקבל יודע לטפל (בעזרת מס' סידורי).
- **בעיות**:
  - מניח שאין שינוי בסדר ההודעות (FIFO), למרות שרשתות datagram לא מבטיחות FIFO. לדוגמא- שתי חבילות שנשלחות זו אחר זו, לשנייה מסלול קצר משל הראשונה ועל כן היא מגיעה לפניה. במיוחד יש שינוי סדר כאשר משתמשים בפרוטוקול ב-IP.
  - אין **pipelining**: לכל היותר שולחים חבילה אחת ומחכים לקבל עליה ACK. איטיות.

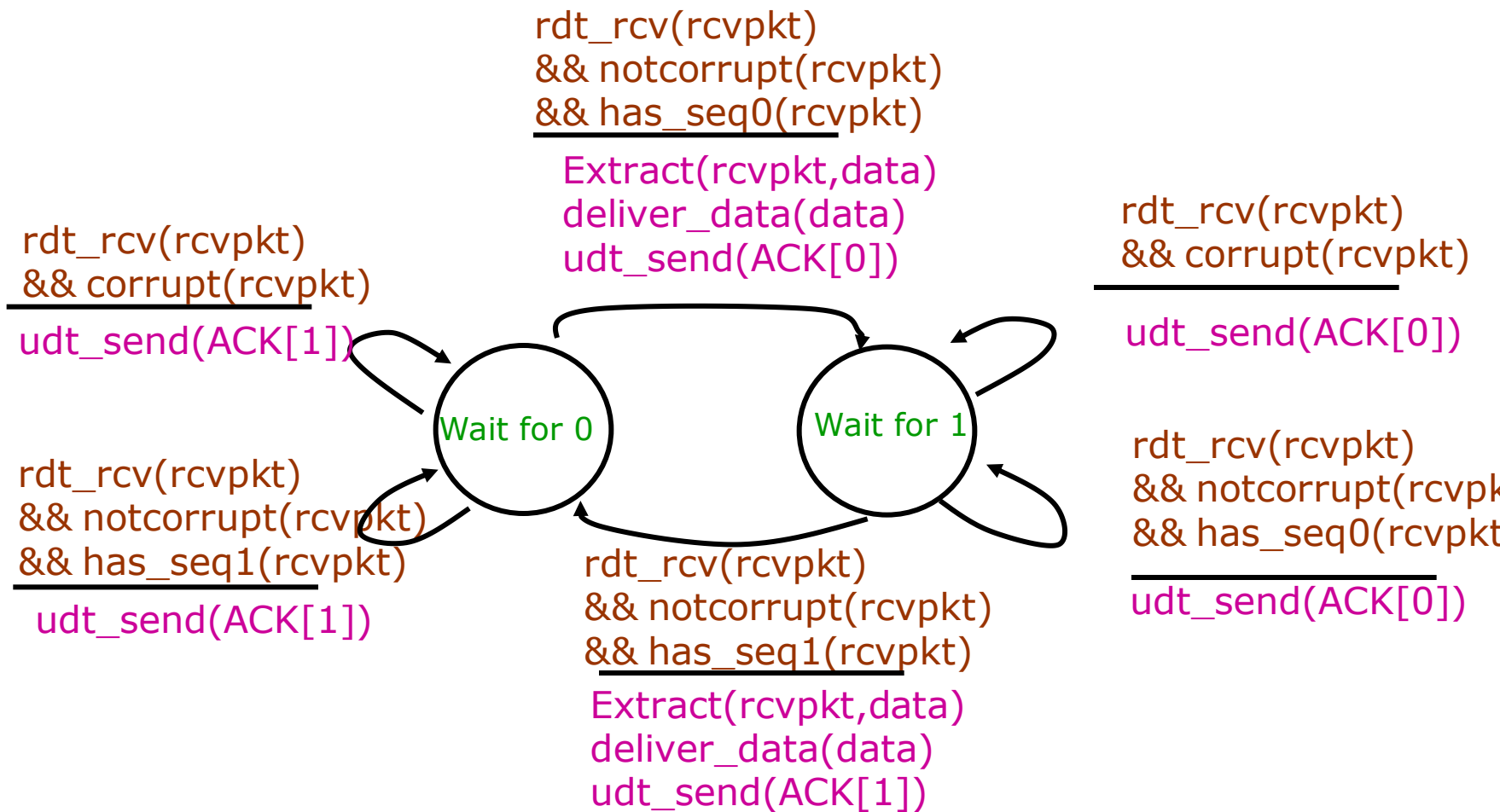
# rdt3.0: פרוטוקול הביט האחתה

הצד השולח:



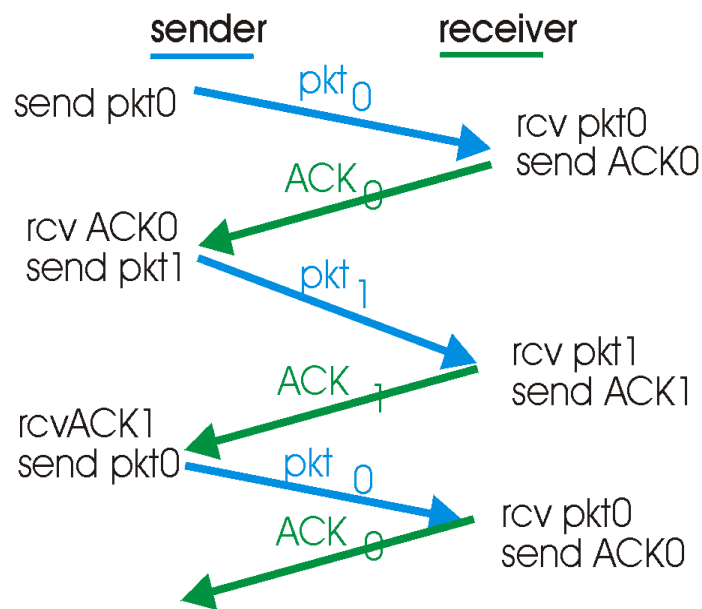
# rdt3.0: פרוטוקול הביט האתחל

הצד המקבל:

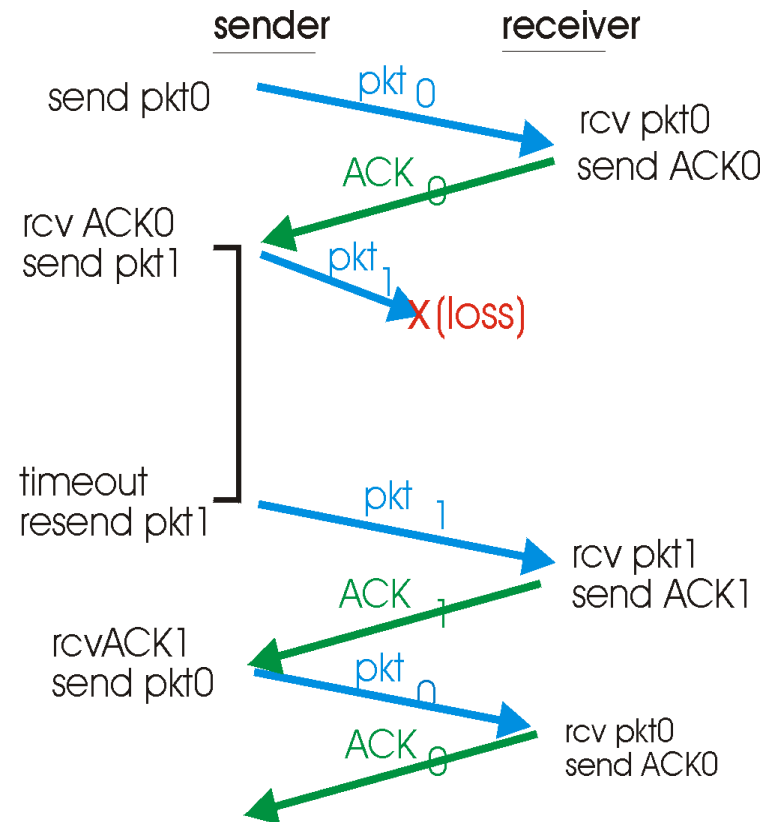


# rdt3.0 פרוטוקול

בתרשימים הבאים מתוארים מצבים של פעילות רגילה, אובדן חבילה, אובדן ACK, ו-timeout שהתרחש לפני שהגיע ה-ACK וגרם להכפלה ממנה הייתה התעלמות:

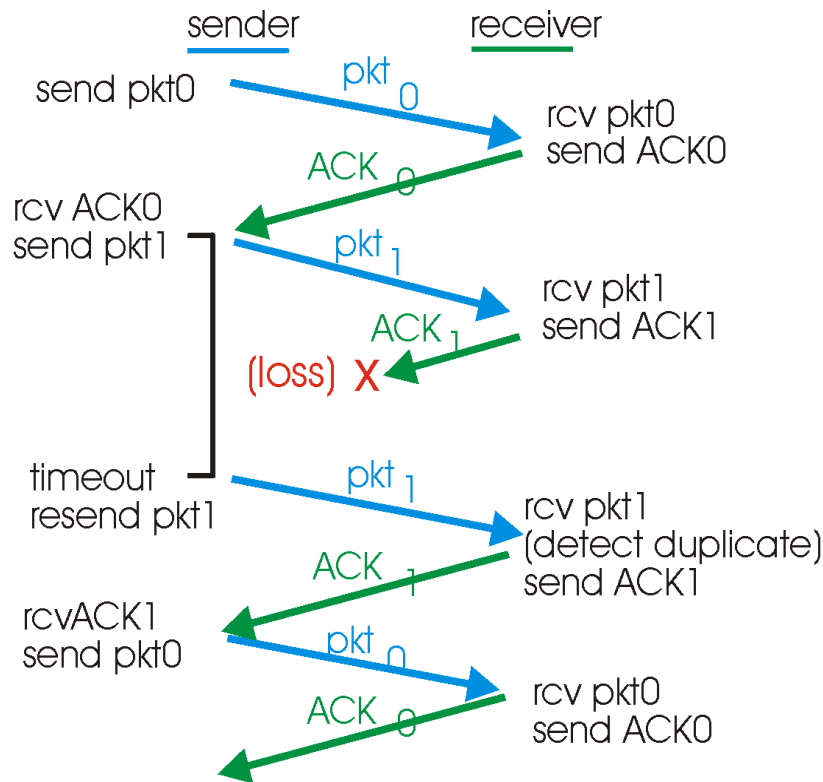


(a) operation with no loss

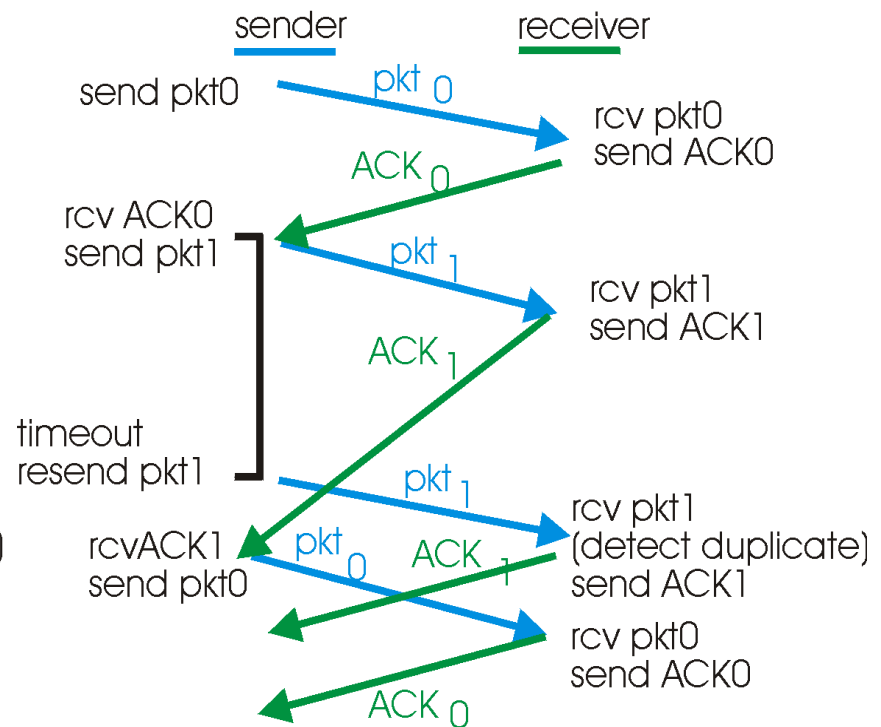


(b) lost packet

# רדט3.0



(c) lost ACK



(d) premature timeout

## ביצועי rdt3.0

- ביצועי הפרוטוקול יכולים להיות גרועים בגלל ש-RTT יכול להיות גדול עקב תורים בנתבים.
- זמן השידור (transmission delay) תלוי באורך החבילה המשודרת ובקצב השידור:
- דוגמא לחישוב נצילות:

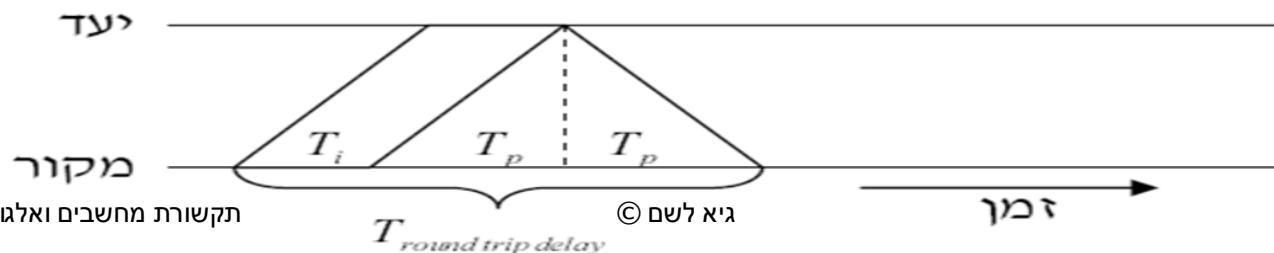
$R=1$  Gbps link,  $T_p=15$  ms (prop. delay),  $L=8000$  bit packet

$$T_i = \frac{\text{אורך - חבילה}}{\text{קצב - שידור}} = \frac{L}{R} = \frac{8000 \text{ bits}}{1 * 10^9 \text{ bps}} = 8 \text{ microseconds}$$

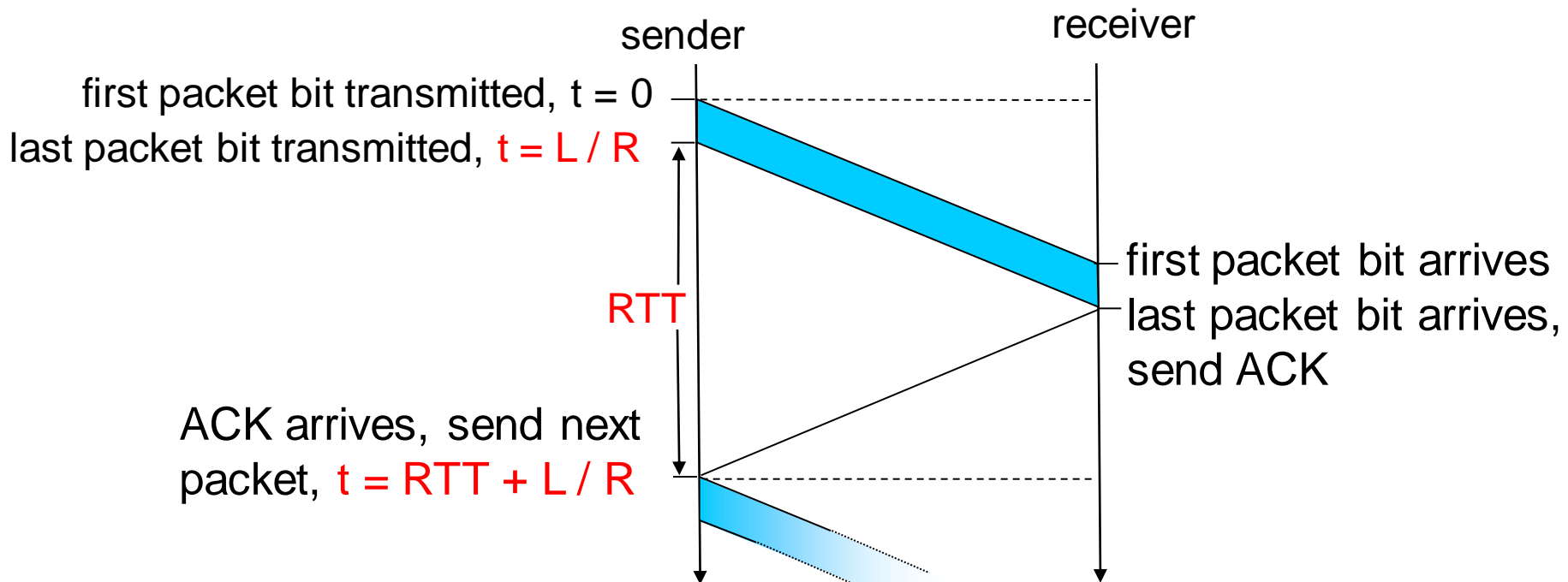
- ניצולת הערוץ: זמן שידור מסגרת אחת ביחס לזמן הכולל למסגרת להישלח ולחזור

$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

דיאגרמת זמנים:



# rdt3.0 ניסיון הער



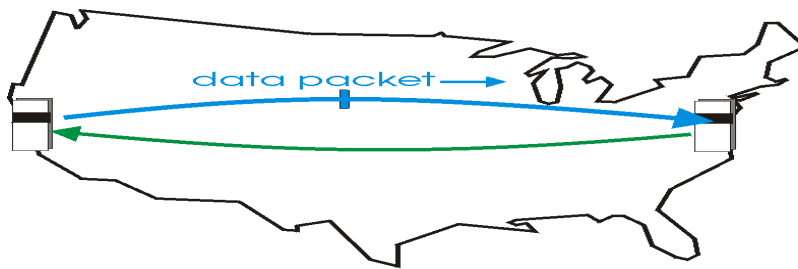
$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

החישוב מדגיש שמערוץ רחב קיבלנו ניצולת מאוד נמוכה.

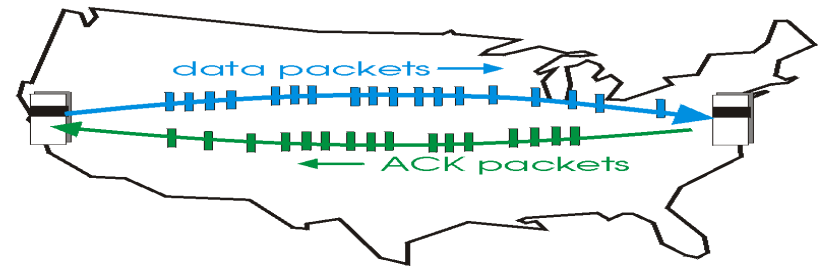


# פרוטוקול Pipeline

- בפרוטוקולים מסוג stop and wait נוצרה בעיה של ניצולת נמוכה, ניתן לשלוח הודעה אחת בכל פעם בלבד. כמו כן RTT יכול להיות גדול בגלל תור שנוצר בנתבים.
- פרוטוקולים מסוג **pipeline** מאפשרים שליחת הרבה הודעות במקביל מבלי שקבלנו עליהן **ACK**.
- הדבר מחייב הגדלת טווח המספור במונים של החבילות. כמו כן צריך חוצצים בשולח כדי לשמור הודעות שנשלחו ועדיין לא התקבל עליהן ACK למקרה שנצטרך לבצע שליחה מחודשת של ההודעה. לפעמים צריך גם חוצצים במקביל.
- מתמודד גם עם שנויי סדר בחבילות ישנות.
- שימוש ב- pipelining גורם להגדלה של הניצולת, אבל תמיד תהיה מגבלה מסויימת שנובעת מגודל החוצצים שיש לנו. מספור המונה הוא סופי.



(a) a stop-and-wait protocol in operation

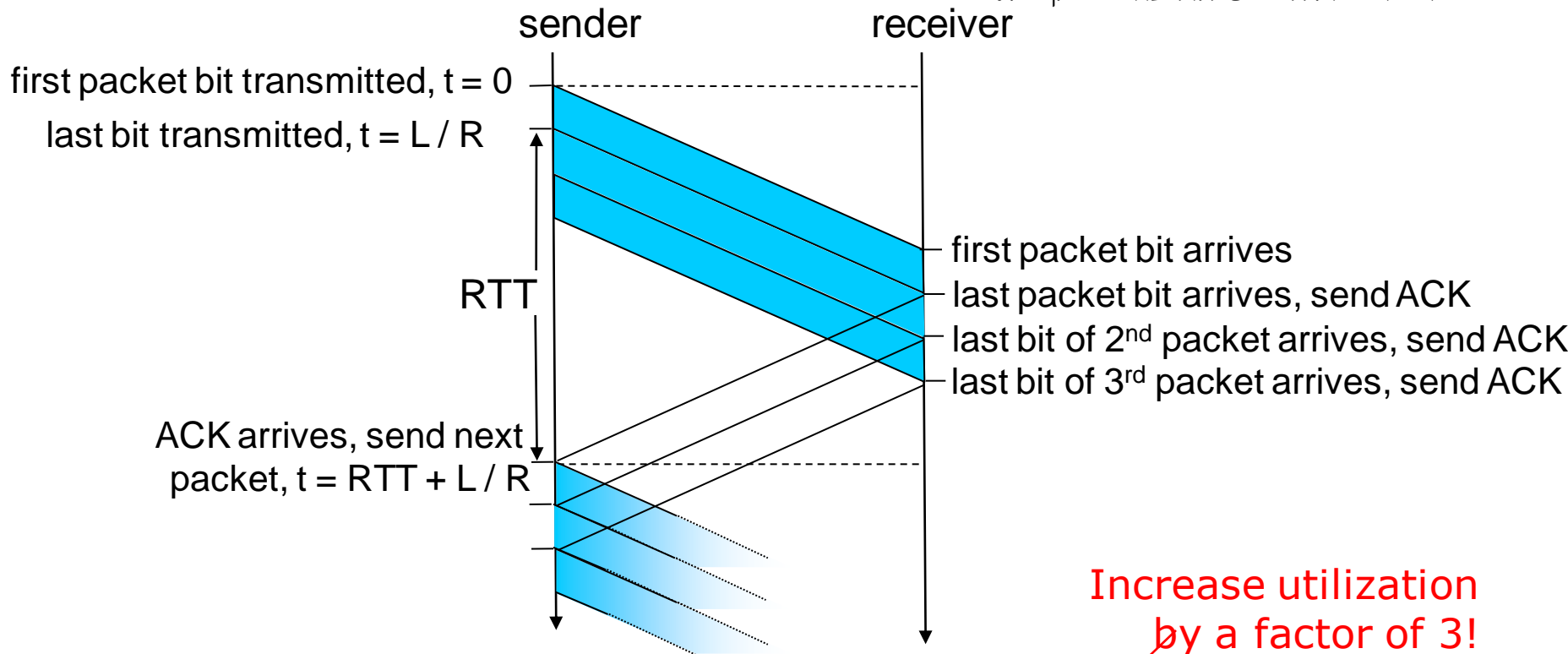


(b) a pipelined protocol in operation

יש שתי אסטרטגיות של pipeline: **Go-Back-N** ו- **Selective Repeat**.

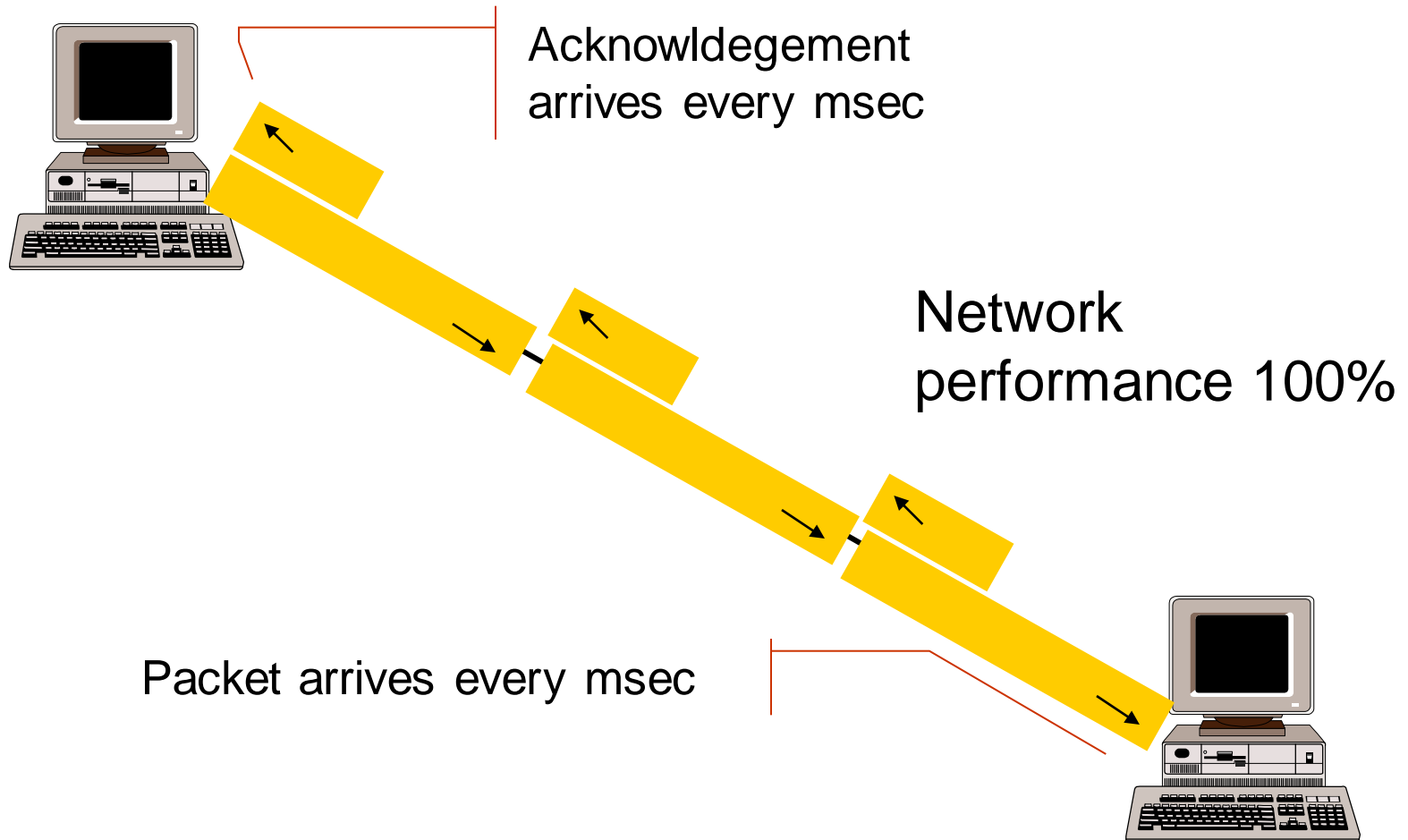
# Pipelining: האדפאת הניצולת

נשתמש בנתונים של החישוב הקודם כדי להראות כיצד pipeline מגדיל את הניצולת, במקרה זה פי 3 כאשר שולחים 3 הודעות במקביל:



$$U_{\text{sender}} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

# (המשק) Pipelining



## Sliding Window Protocols

---

□ חבילות ו-ACK-ים ממסופרות.

□ חלון שליחה (אצל המשדר)

■ מתחיל מהחבילה הראשונה שנשלחה ועדיין לא אושרה.

■ המשדר יכול לשלוח עד  $W_S$  חבילות "קדימה".

■ תפקיד חלון השליחה: לאפשר "עבודה במקביל"

□ חלון קבלה (אצל המקלט)

■ מתחיל מהחבילה הבאה שלה מצפה המקלט.

■ המקלט יכול לבאפר עד  $W_R$  חבילות "קדימה" – כאלה

שהגיעו שלא לפי הסדר המצופה. חבילות מחוץ לחלון –  
נזרקות.

■ תפקיד חלון הקבלה: לארגן את החבילות ולמסור אותן לשכבה

שמעל - לפי הסדר הנכון.

# פרוטוקול הלנת חלון

□ נראה:

■ **Go-Back-N** :  $W_R=1, W_S=N$

■ **Selective Repeat** :  $W_R= W_S=N$

□ פרוטוקולים "בטבע":

■ HDLC (שכבה שניה) מיישם את Go-Back-N.

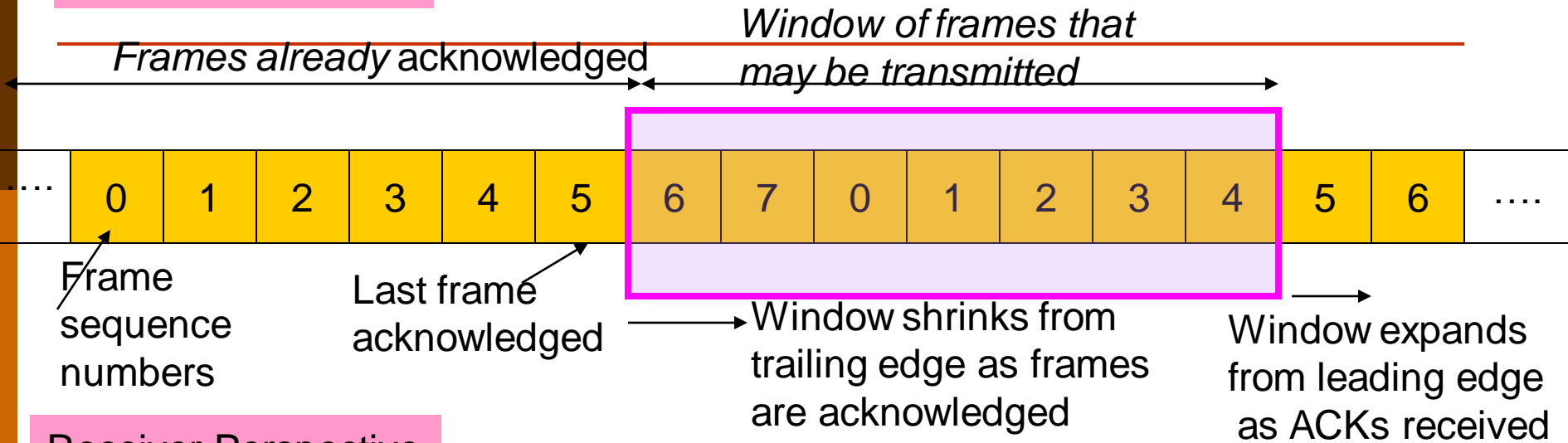
■ TCP (שכבה רביעית) מיישם את SR.

□ **High-level Data Link Control**: פרוטוקול תלוי סיביות (Bit Oriented) המגדיר שיטה לעטיפת נתונים (Data encapsulation) ברמת שכבת קישור הנתונים ובמודל ה-TCP/IP.

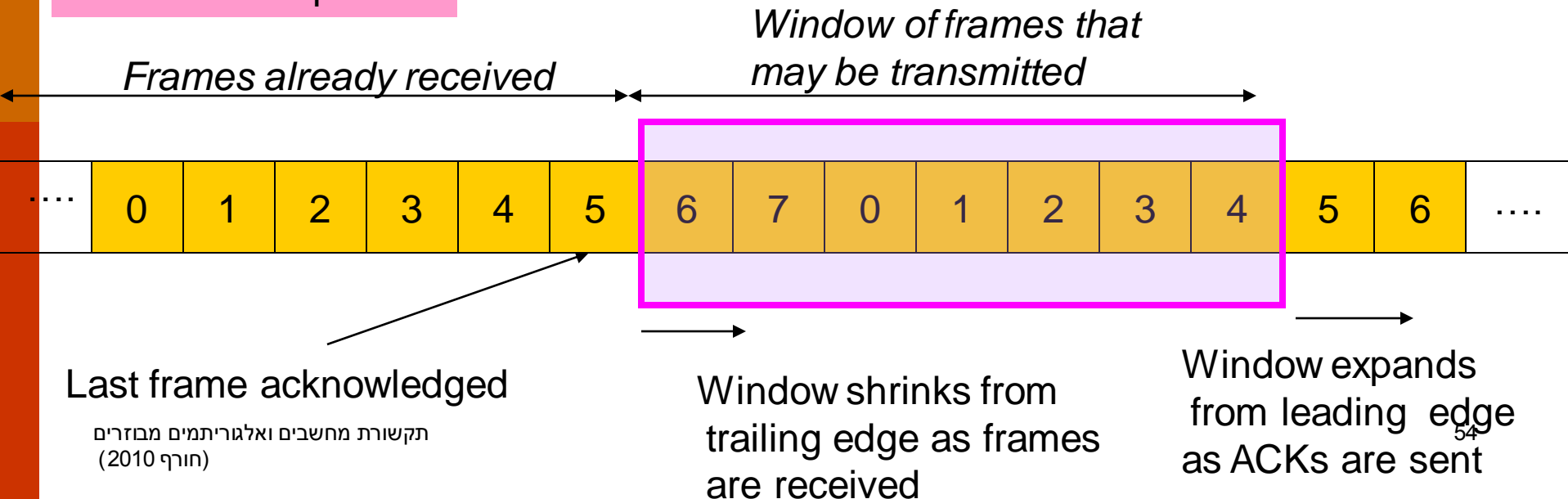
□ **TCP**: נלמד בהמשך.

# הצגת חלון - Sliding Window

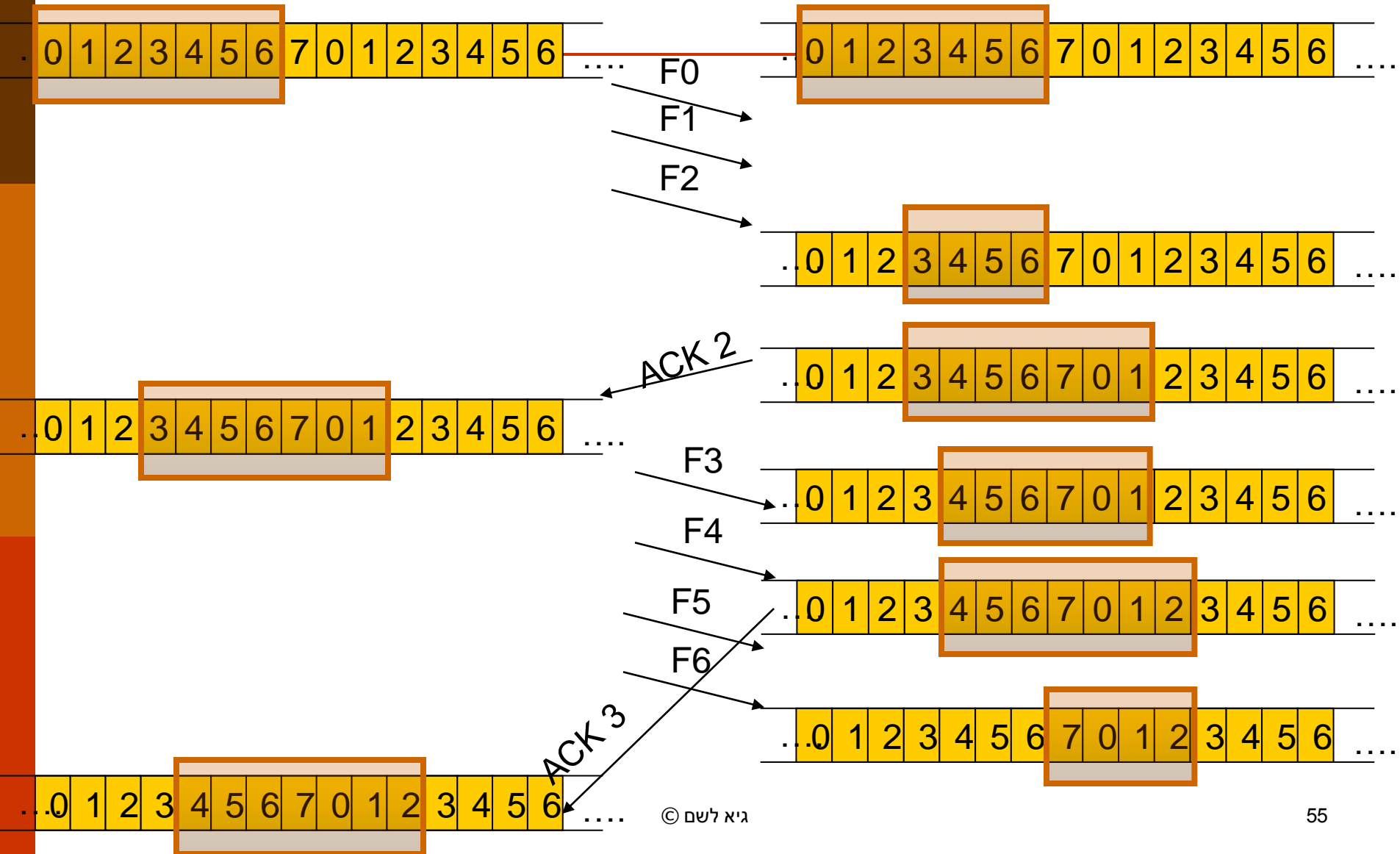
## Sender Perspective



## Receiver Perspective



# פונקציה של חלון



# פרוטוקול Go-Back-N

---

## □ תחנה משדרת

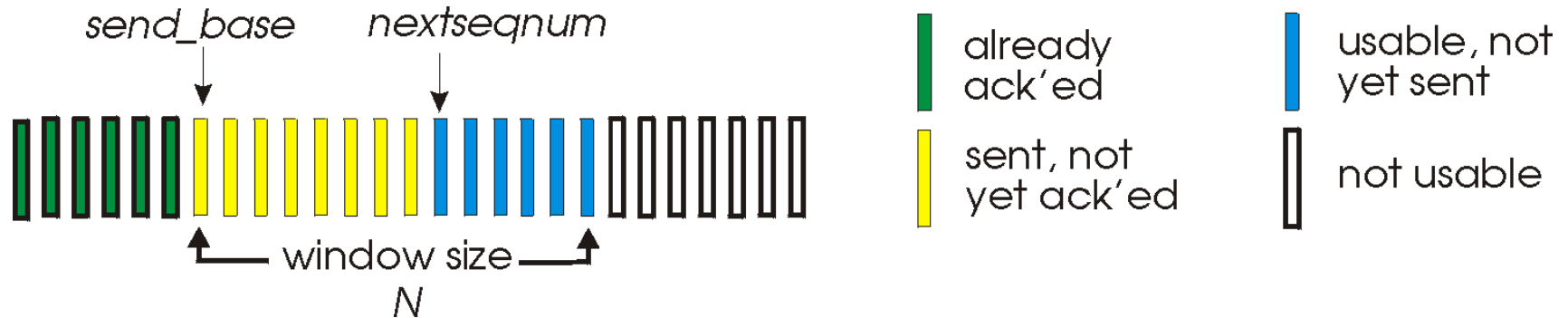
- משדרת חבילות בזו אחר זו לפי חלון שגודלו  $N$ .
- כאשר עובר  $T_{out}$  חוזרת לחבילה האחרונה שעדיין לא אושרה וממשיכה לשדר חבילות החל מחבילה זו.

## □ תחנה קולטת

- מקבלת ומאשרת חבילות אך ורק לפי סדר (האישורים "מצטברים": אישור על חבילה  $n$  מאשר את כל החבילות מהראשונה ועד  $n$ , כולל)

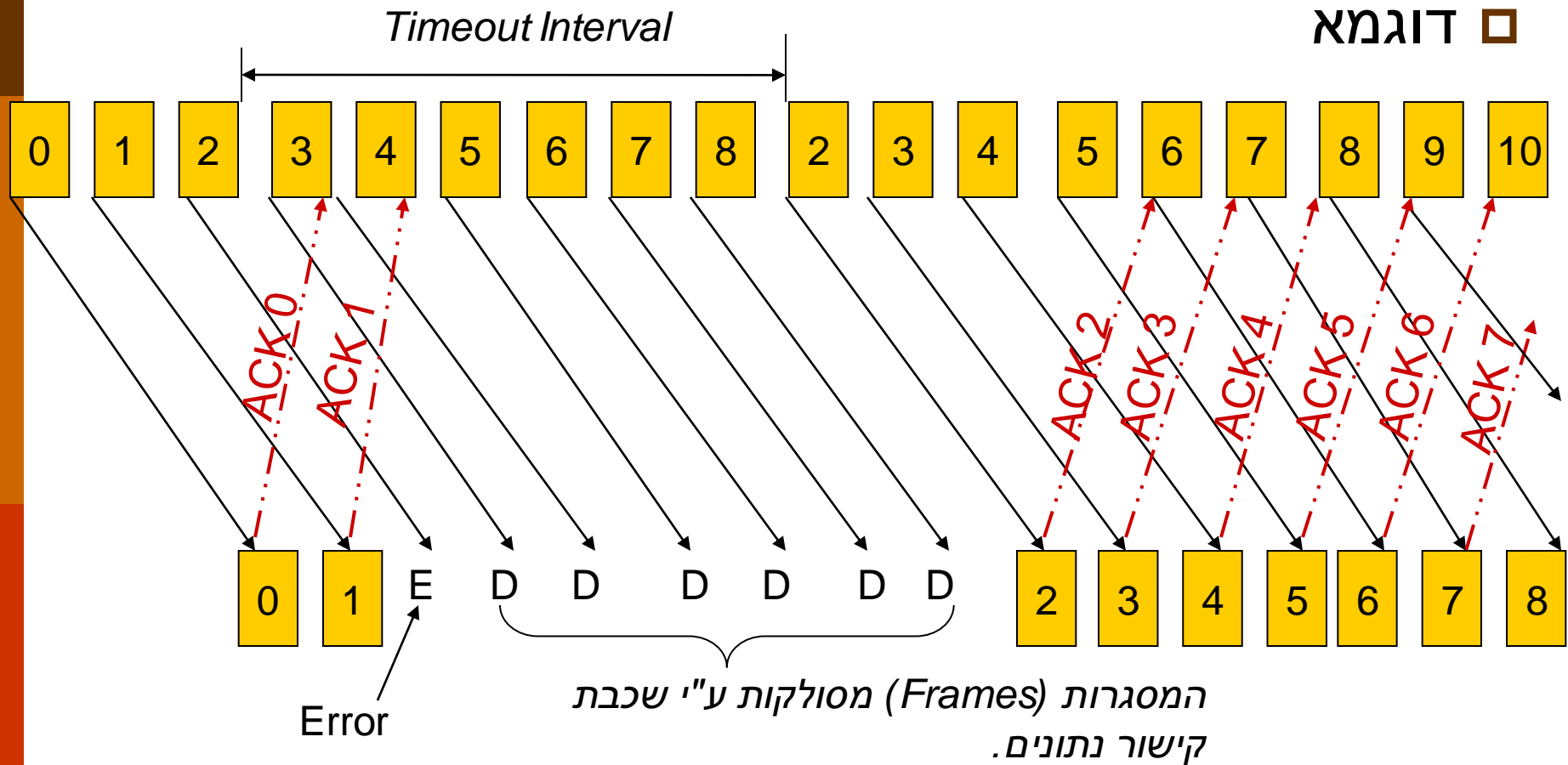


# פרוטוקול Go-Back-N



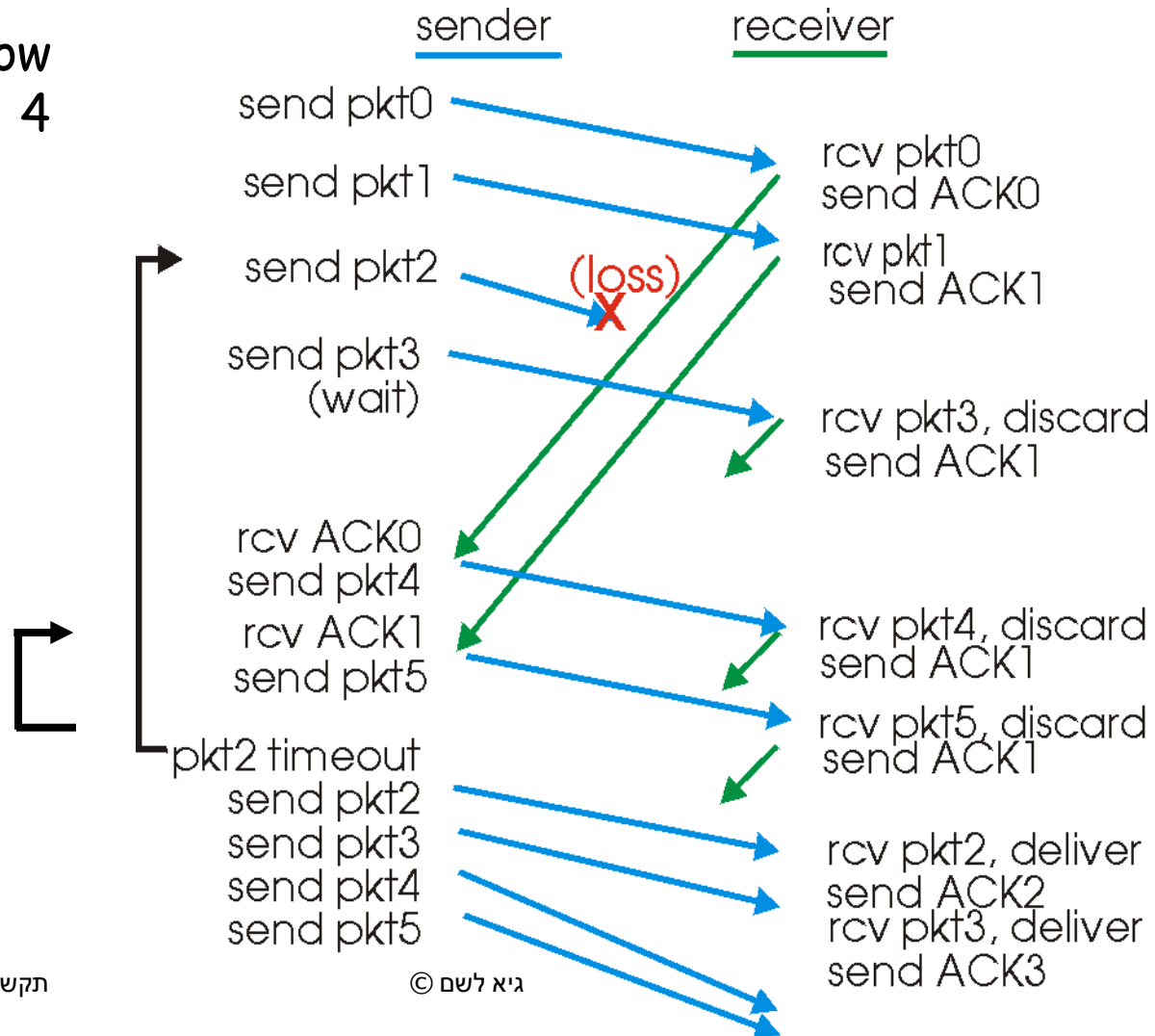
- בפרוטוקול GBN לשולח מותר לשלוח יותר מחבילה אחת בלי לחכות לאישור. קיימת הגבלה למספר הודעות מקסימלי N ב-pipeline. הטווח של מספרים סידוריים אפשריים לחבילות שנשלחו אבל עדיין לא התקבל עליהן אישור מוגדר כ"חלון" בגודל N. תוך כדי פעולת הפרוטוקול החלון זז קדימה.
- לכל חבילה יש timer, וכאשר יש אירוע timeout לחבילה מסוימת יש שידור מחדש של אותה החבילה. **למקבל אין חוצץ** ולכן כל החבילות שנשלחו אחרי חבילה זו ישלחו מחדש גם הן. נוצרת בעיה - ככל שהערוץ פחות אמין יש יותר שליחות מחדשות של הודעות.
- העובדה שלמקבל אין חוצץ היא יתרון עבור רכיבים קטנים.

# פרוטוקול Go-Back-N

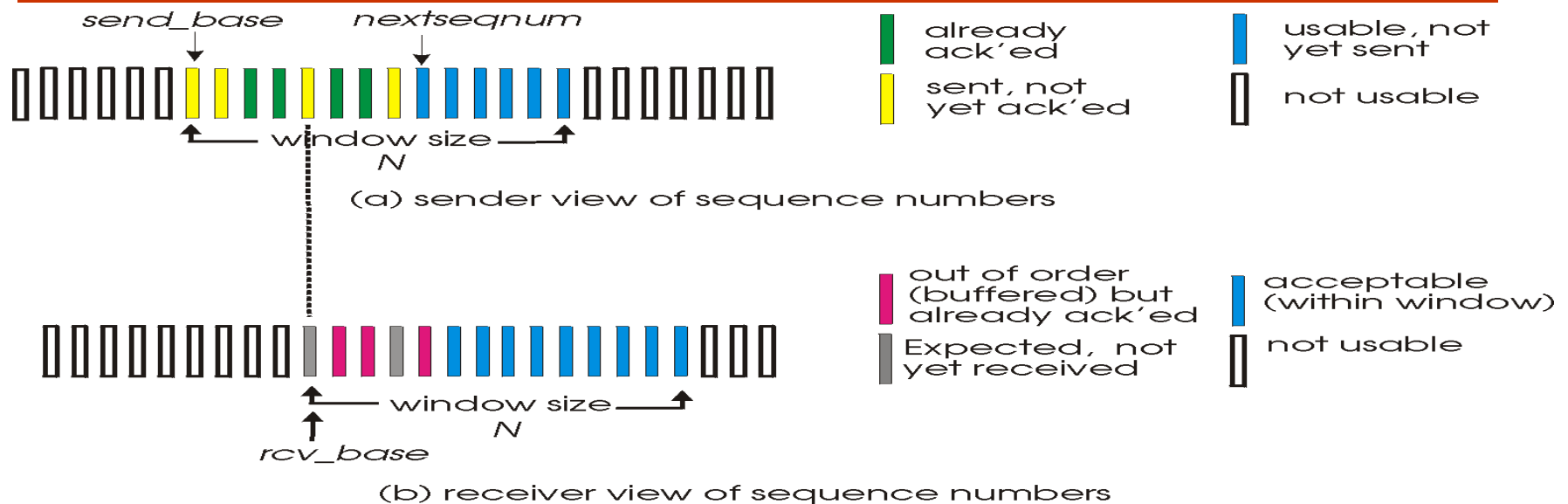


# פרוטוקול Go-Back-N בעבודה

window  
size = 4



# פרוטוקול Selective Repeat



- הפרוטוקול מונע שליחה מחדש מיותרת בגלל שהשולח שולח מחדש רק את החבילות שאבדו או הגיעו למקבל משובשות. לשם כך המקבל צריך לשלוח ACK לכל חבילה שהגיעה תקינה בנפרד.
- נדרש כאן חלון בגודל  $N$  כדי להגביל את מספר החבילות הבלתי מאושרות ב-pipeline. שלא כמו בפרוטוקול GBN השולח כבר קיבל ACK למספר הודעות שנמצאות בתוך החלון. הצד המקבל שולח ACK לחבילה תקינה שהתקבלה גם אם לא התקבלה בסדר הנכון.
- חבילות שמגיעות בסדר לא נכון נשמרות בחוצץ אצל המקבל עד שהחבילות החסרות יתקבלו, ואז החבילות בחוצץ יועברו בסדר הנכון לרמה העליונה.

## פרוטוקול Selective Repeat (SR)

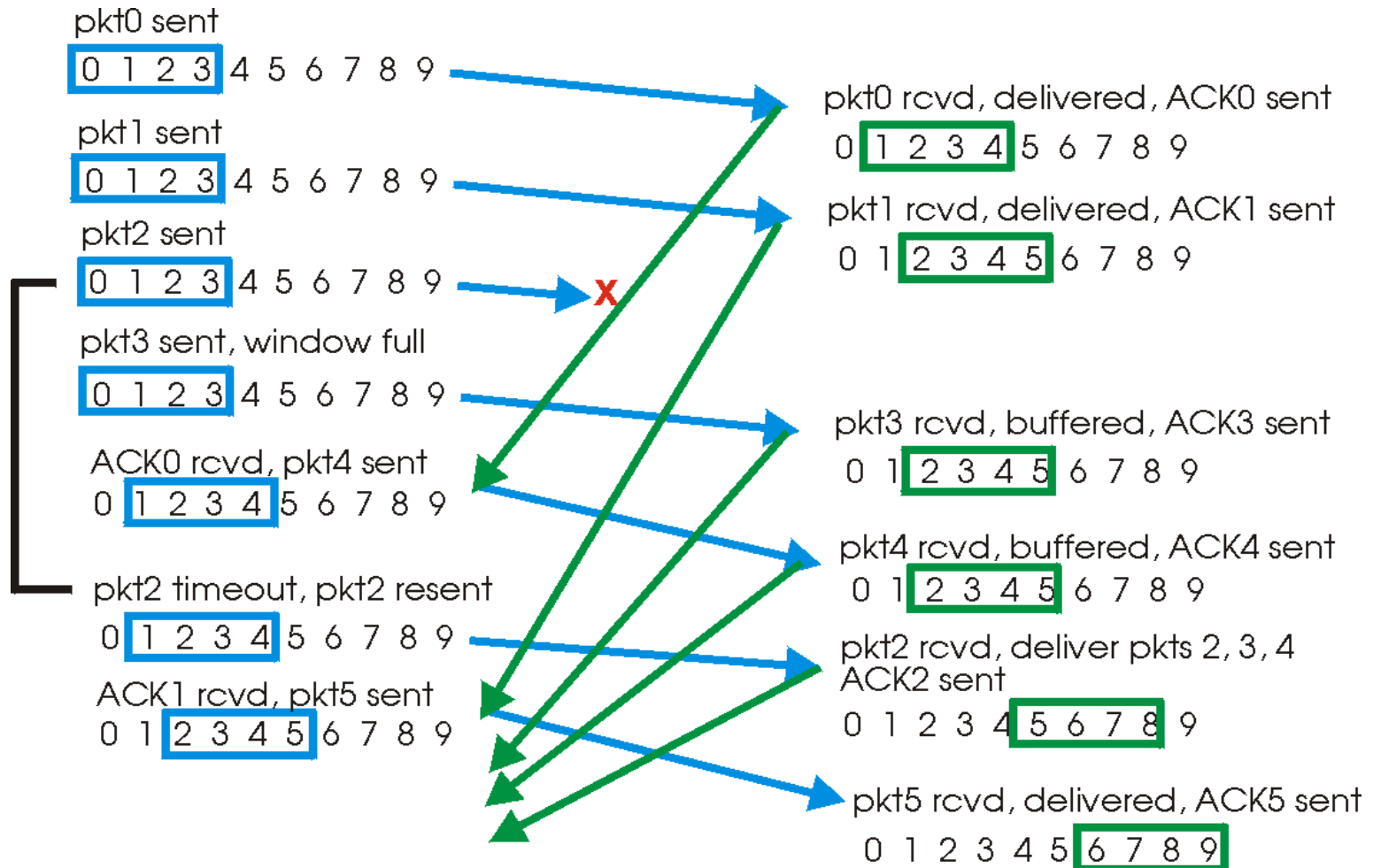
### □ תחנה משדרת:

- משדרת חבילות ברצף זו אחרי זו כל עוד הן נמצאות בגבולות החלון.
- אם לא התקבל  $ACK(i)$  אחרי זמן  $timeout(i)$  משדרת שוב את חבילה  $i$
- $ACK(i)$  שהתקבל מאשר את חבילה  $i$ . אם זו היתה החבילה בעלת ה-SN הנמוך ביותר שעדיין לא אושר – היא מפונה מהחוצץ, והחלון מתקדם.

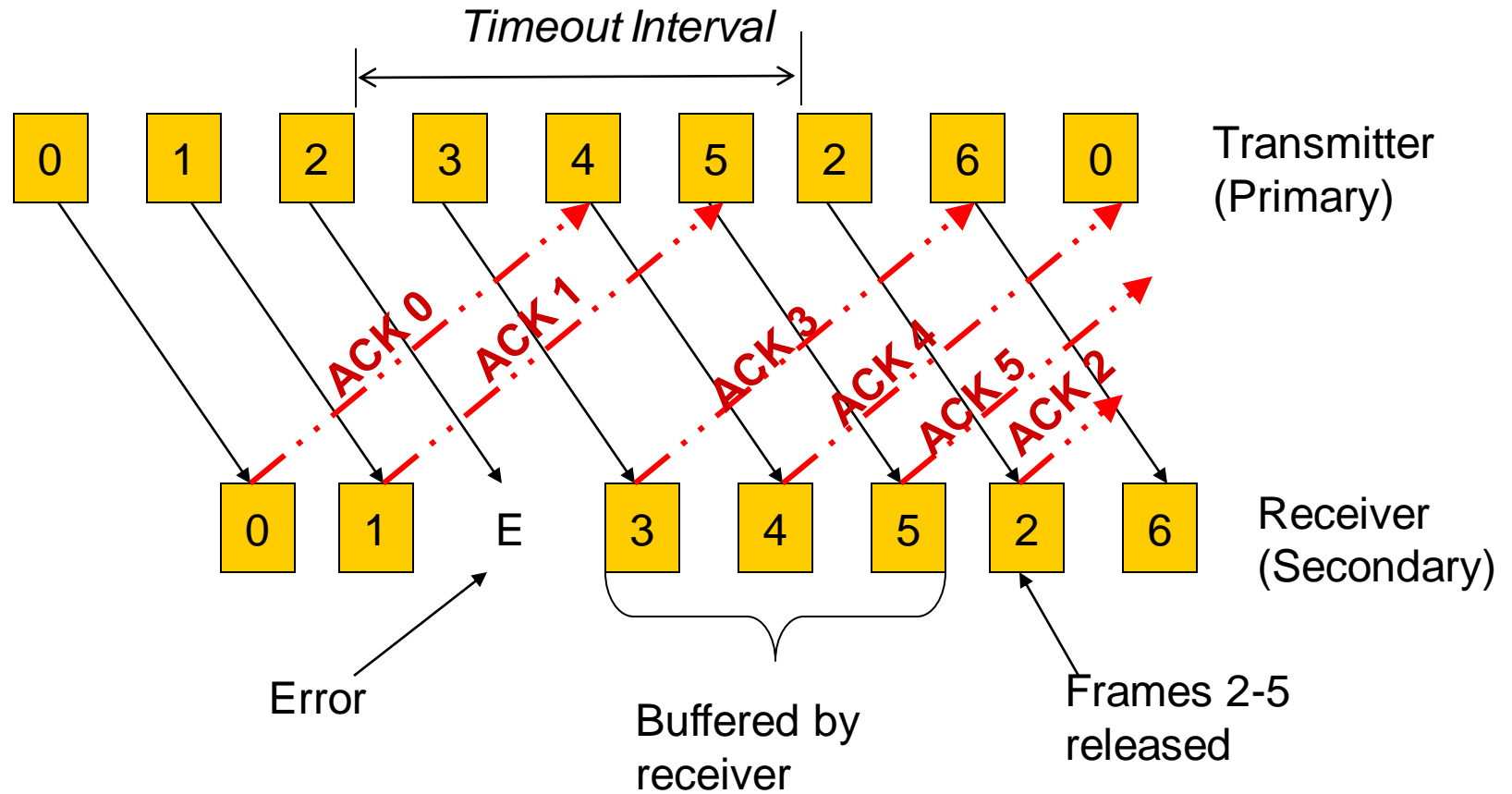
### □ תחנה קולטת:

- קולטת את חבילה  $i$  ושולחת עליה  $ACK(i)$ .
- אם החבילה הגיעה לפי סדר – מעבירה אותה לשכבה מעל, ומקדמת את החלון בהתאם.

# פרוטוקול Selective Repeat הספירה



# Selective Repeat פרוטוקול



# שכבת ההובלה

---

- שרותי שכבת ההובלה.
- ריבוב/פילוג.
- עקרונות של בקרת עומסים.
- תעבורה חסרת קשר: UDP.
- בקרת עומסים TCP.
- עקרונות תעבורת מידע  
אמינה ברשת.
- תעבורה מונחת קשר: TCP
  - מבוא
  - מבנה המקטע.
  - תעבורת מידע אמינה.
  - בקרת זרימה.
  - ניהול הקשר



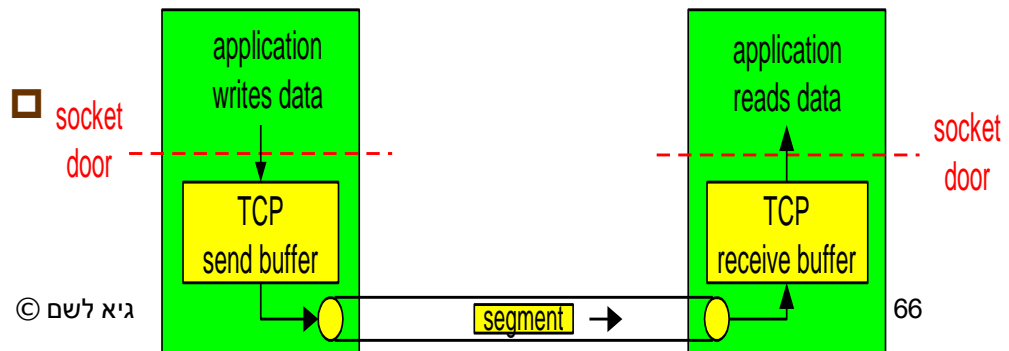
# תכונות יסוד – TCP

- פרוטוקול הממש מעגל וירטואלי (Virtual Circuit).
- מאפשר תקשורת אמינה מקצה לקצה.
- מעביר רצף (לא חסום) של ביתים בשני הכוונים
- משתמש בפרוטוקול IP לקישוריות בלבד ומנהל את שאר העניינים בעצמו.
- רואה את כל הרשת כערוץ לא אמין המסוגל להפר סדר המשלוח.
- פרוטוקול ממשפחת GBN (המיישם גם את SR) כאשר המספור מתיחס לבתים.
- בקרת הזרימה בעזרת חלון.
- הסבר על פעולת פרוטוקול TCP:

- הקמת הקשר בין תחנת מקור לתחנת יעד זהו רק שלב אחד בדו שיח שבין שתי התחנות, הצעד הבא הוא ניהול מתמשך של תקשורת בין תחנת היעד לתחנת המקור, כדי לשדר את כל המידע הדרוש, כדי לוודא את הגעתו התקינה וכדי לשלוח מחדש חבילות מידע שהגיעו אל היעד פגומות. בכיוון ההפוך, צריך הפרוטוקול למזג חבילות מידע למידע השלם, תוך הרכבת החבילות בסדר הנכון ולאחר שוידא שהחבילות תקינות.
- פרוטוקול TCP מקבל מידע למשלוח משכבת היישום, מחלק את המידע למנות שוות גודל ומעביר אותן אל שכבת הרשת להמשך טיפול, כמו כן מטפל הפרוטוקול בהרכבה מחדש של מנות המגיעות אל התחנה, בונה את המידע השלם ומעבירו אל שכבת היישום. אם לא הגיעו כל חבילות המידע הדרושות אל תחנת היעד או שחלקן הגיע במצב לא תקין, תוך תקשורת בין פרוטוקולי TCP במחשב המקור ובמחשב היעד, יעודכנו הצדדים על חבילות שיש לשלוח מחדש ויטפלו בכך עד להשלמת שליחה וקבלת המידע כולו, או עד סיומו של פרק הזמן שהוקצב לתהליך, אז תוצג הודעת שגיאה מתאימה.

# תקשורת מכוונת קשר (Connection Oriented) TCP

- קשר נקודה לנקודה point to point (point) - שולח אחד ומקבל אחד. אין אפשרות ל-multicasting: העברת מידע משולח אחד להרבה מקבלים בפעולת שליחה אחת אינה אפשרית.
- הפרוטוקול רץ רק במחשבי הקצה ולא בנתבים בדרך.
- אמינות: הגעת הודעות בסדר הנכון.
- תומך ב-session דו-כיווני, שני הצדדים שולחים ומקבלים מידע
- (full duplex data) - קיים MSS (maximum segment size) שהוא גודל מקסימלי של סגמנט שניתן להעביר.
- מכוונת קשר (connection oriented): קיימת לחיצת יד (handshake) שמאתחלת את מצבי השולח והמקבל לפני העברת מידע ביניהם.
- בקרת זרימה (flow control): השולח לא ישלח יותר מידע ממה שהנמען יכול לקבל.
- קיימים חוצצים בשני הצדדים.



# שרותי ה-TCP

מכיוון שסגמנטים של TCP נשלחים כחבילות IP ומכיוון שחבילות IP יכולות להגיע לוא דוקא באותו הסדר שבו נשלחו, גם סגמנטים של TCP יכולים להגיע לא באותו הסדר שבו נשלחו. TCP המקבל חבילות מסדר אותן מחדש לפי הסדר שבו נשלחו, אם יש צורך, ומעביר את המידע מסודר לאפליקציה.

מאחר וחבילות IP יכולות להשתכפל, TCP יודע לזרוק חבילות משוכפלות.

TCP נותן בקרת זרימה. כל צד בקשר TCP מכיל Buffer סופי. TCP שמקבל חבילות מהצד השני יאפשר לצד השני לשלוח מידע לפי יכולות ה-Buffer שלו, כלומר לא יאפשר לצד השני לשלוח חבילות ללא קבלת אישור מעבר לגודל ה-buffer. דבר זה מונע ממחשב מהיר לסתום את ה-Buffer של מחשב איטי.

## TCP נותן אמינות ע"י התכונות הבאות:

- המידע של האפליקציה מחולק לחלקים בעלי גודל המוגדר ע"י ה-TCP כ-"גודל הטוב ביותר לשליחה". חלק המידע המועבר ע"י ה-TCP לרמת ה-IP נקרא סגמנט.
- כאשר TCP שולח סגמנט הוא מחזיק Timer שמחכה לקבלת אישור על קבלת הסגמנט מהצד השני. אם האישור אינו מגיע תוך זמן ה-timeout הסגמנט משודר שוב.
- כאשר TCP מקבל מידע מהצד השני הוא שולח אישור על קבלת המידע. האישור אינו נשלח מידית, בד"כ הוא מעוקב חלקיק שניה.
- TCP מחזיק checksum של המידע ושל הרישא של חבילת ה-TCP. מטרת ה-checksum למצוא האם ארעה שגיאה בחבילה בין השידור לקליטה. במידה ו-TCP מוצא checksum שגוי הוא יזרוק את החבילה ולא יודיע לצד השני על קבלתה.

# שכבת ההובלה

---

- שרותי שכבת ההובלה.
- ריבוב/פילוג.
- תעבורה חסרת קשר: UDP.
- עקרונות תעבורת מידע  
אמינה ברשת.
- תעבורה מונחת קשר: TCP
  - מבוא
  - מבנה המקטע (סגמנט).
  - תעבורת מידע אמינה.
  - בקרת זרימה.
  - ניהול הקשר
- עקרונות של בקרת עומסים.
- בקרת עומסים TCP.

# TCP חבילת

Source Port			Destination Port		
Sequence Number					
Acknowledgment Number					
Data Offset	Reserved	Flag	Window		
Checksum			Urgent Pointer		
Option + Padding					
Data					

מזהה את הנקודה בה שכבת היישום בתחנת המקור ובתחנת היעד מקבלת שרות מפרוטוקול TCP ( SrcPort , DstPort – 16 ביט כל אחד).	Destination and Source Port
מספר סידורי המזהה את השידור עבור מנות השייכות להודעה.	Sequence Number
מספר סידורי המציין את מספר החבילה שתחנת המקור מצפה לקבל (32 בייט).	Acknowledgment Number
מציין את מספרן של המילים בנות 32 ביט בכותרת מסגרת ה-TCP.	Data Offset
שמור לשימושים עתידיים.	Reserved
נושא מידע בקרה מגוון.	Flag
מגדיר את גודל החוצץ המיועד לקבלת נתונים בתחנת המקור.	Window
מציין האם החבילה נפגעה בתהליך המשלוח.	Checksum
מצביע אל שדה המידע בחבילה.	Urgent Pointer
מגדיר מגוון אפשרויות TCP.	Option
מכיל את המידע משכבת היישום.	Data

# מדוע דרוש אילווי/תיקון שגיאות אם כרמה 4

## ואם כ- 2

□ כיוון שבשכבת קישור נתונים שרות משלוח אמין לא תמיד קיים.

הסבר:

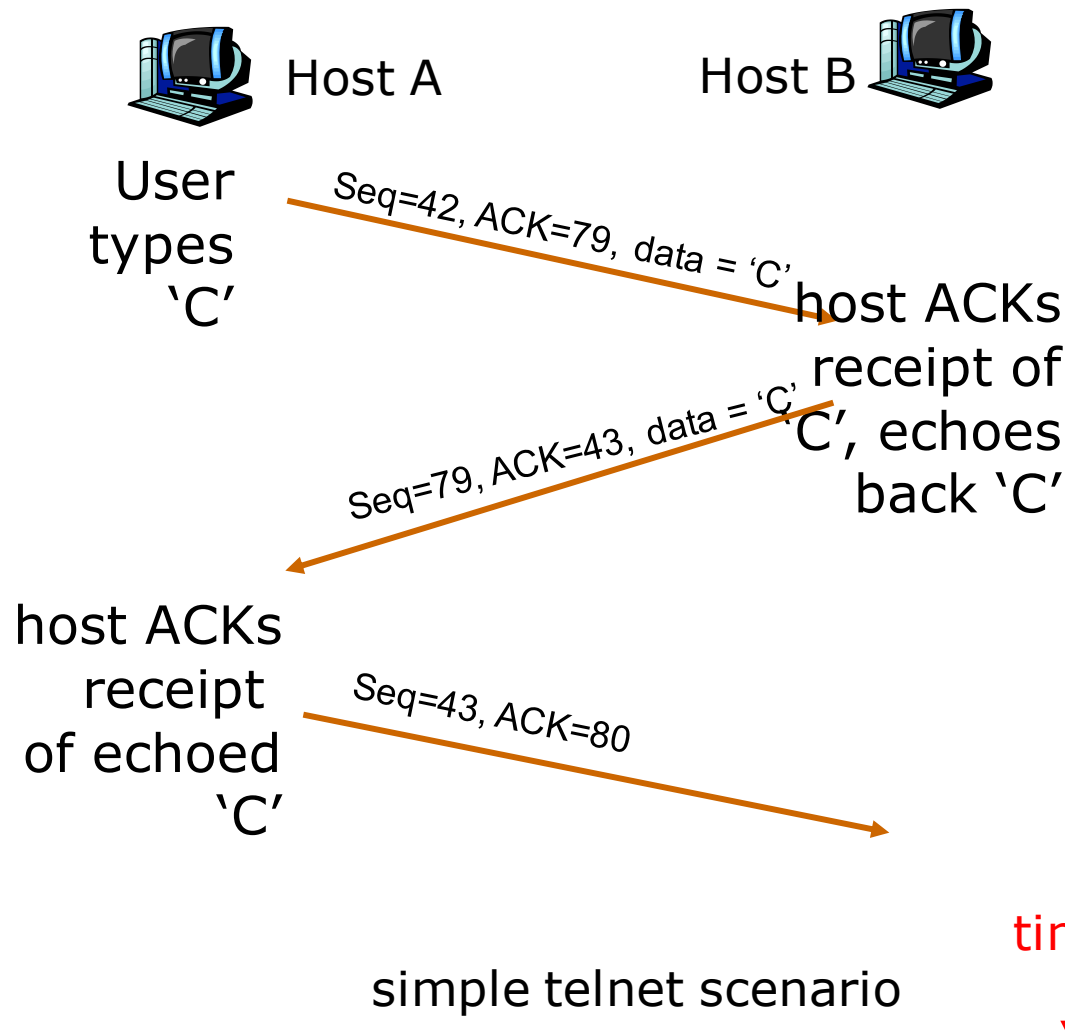
□ משלוח אמין: אם שכבת קישור הנתונים מספקת שירות משלוח אמין, אזי זה מבטיח כי חבילת המידע תנועה בערוץ ללא שגיאות.

□ נזכור שפרוטוקולי שכבת התעבורה (כגון TCP) יכולים גם הם לספק שרות של משלוח אמין. בדומה לשרות משלוח אמין של שכבת התעבורה, שרות משלוח אמין של שכבת קישור נתונים מושג ע"י אישור על קבלה (acknowledgments) ומשלוח מחדש (retransmissions).

□ שרות משלוח אמין של שכבת קישור נתונים בד"כ משמש עבור ערוצים שנוטים לקצב שגיאות גבוה, כגון ערוצים אלחוטים, עם מטרה של תיקון שגיאות מקומי בערוצים שבהם קורות שגיאות במקום לאלץ שליחת נתונים מקצה לקצה ע"י פרוטוקול שכבת התעבורה או האפליקציה.

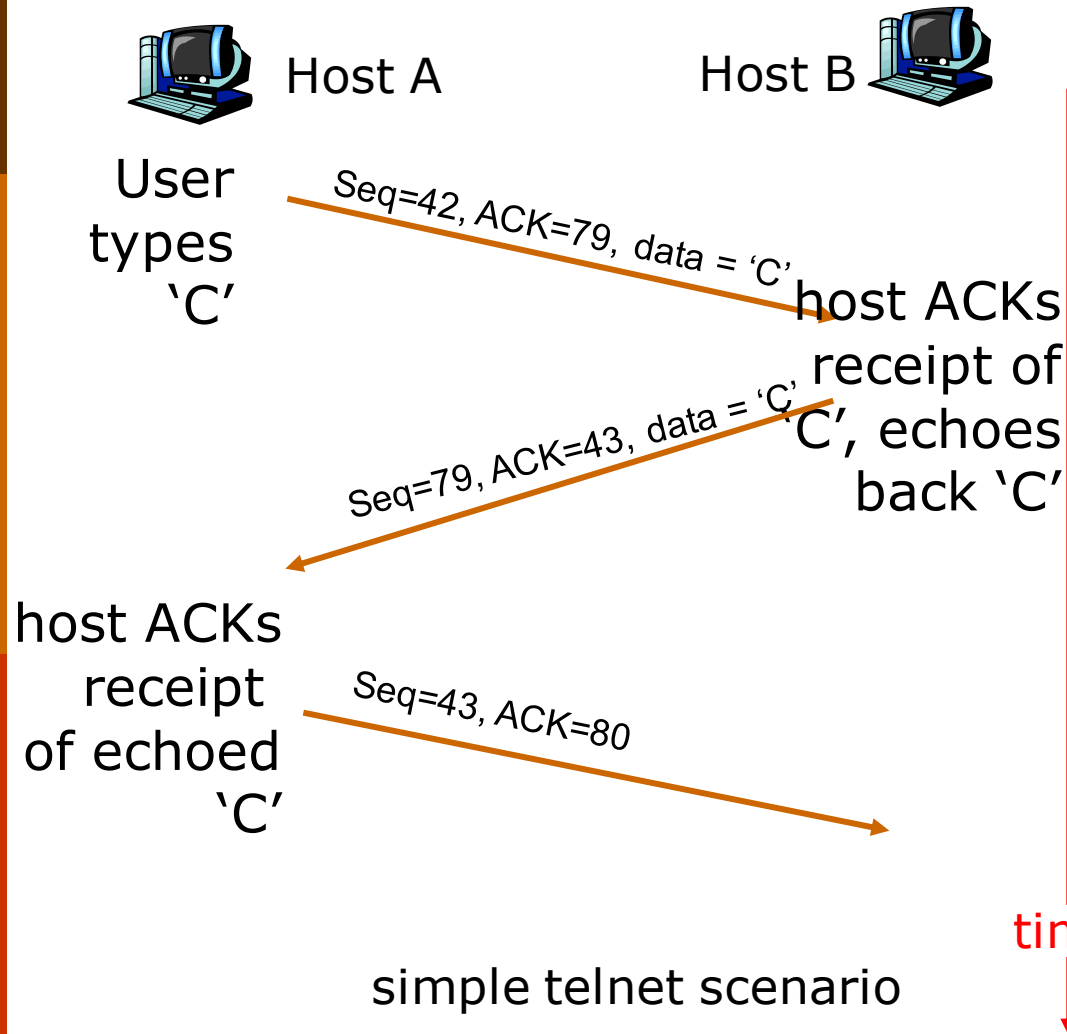
□ בכל מקרה, שרות משלוח אמין של שכבת קישור נתונים לעתים קרובות הוא נחשבת עם תקורה מיותרת עבור ערוץ עם קצב שגיאות נמוך, הכולל סיב אופטי, קבל קואקסיאלי וערוצי זוג שזור רבים. מסיבה זו, פרוטוקולי שכבת קישור נתונים פופולרים רבים לא מספקים שרות מישלוח אמין.

# פרוטוקול TCP - מאורעות 73 השולח



- כאשר מתקבל מידע מרמת היישום:
  - נוצר סגמנט עם מס' סידורי שהוא מספרו הסידורי של ה-byte הראשון באותו סגמנט.
  - אתחול ה-timer אם אינו מופעל כבר. (ה-timer הוא עבור ההודעה הישנה ביותר שלא התקבל עליה ACK).
- Timeout: שליחה מחדש של סגמנט שגרם ל-timer ואתחול ה-timer.
- קבלת ACK: אם מתקבל ACK על חבילות שעדיין לא הגיע להן ACK, אז מתבצע עדכון של הסגמנטים אשר עליהם צריך לעשות ACK.
- במקרה זה שולחים סיגמנט עם 1 byte.

# פרוטוקול TCP - מאורעות בצד המקבל



- מס' סידורי: סגמנט עם מס' סידורי שהוא מספרו הסידורי של ה-byte הראשון באותו סגמנט.
- ACK: מס' סידורי של ה-byte הבא אותו מצפים לקבל מהצד השולח.
- הצד המקבל מתמודד עם סגמנטים שהגיעו לא לפי הסדר בכך שהוא מתעלם מחבילות ישנות, ושם בחוצצים או מתעלם מחבילות עתידיות.



# TCP Round Trip Time and Timeout

שאלה: אך לאמוד את RTT ?

תשובה:

□ כדי לאמוד זמן RTT יש לקחת דגימות שהן הזמן משליחת סגמנט ועד לקבלת ה-Ack. מתעלמים משליחות כפולות ומבצעים ממוצע בין הדגימות.

□ יש שימוש במקדם ביטחון שגדל ככל שיש הבדלים יותר גדולים בין ערכי הדגימות. לבסוף נקבע ערך ה-timeout ע"פ זמן ה-RTT המשוער אליו נוסיף את מקדם הביטחון: (ערך טיפוסי של  $\alpha$  הוא 0.125).

שאלה: אך נאמוד את הערך של ה-Timeout ב-TCP ?

תשובה:

□ ישנו קושי בקביעת ערך ה-Timeout ב-TCP. ערכו צריך להיות גדול מערך ה-RTT שמשתנה.

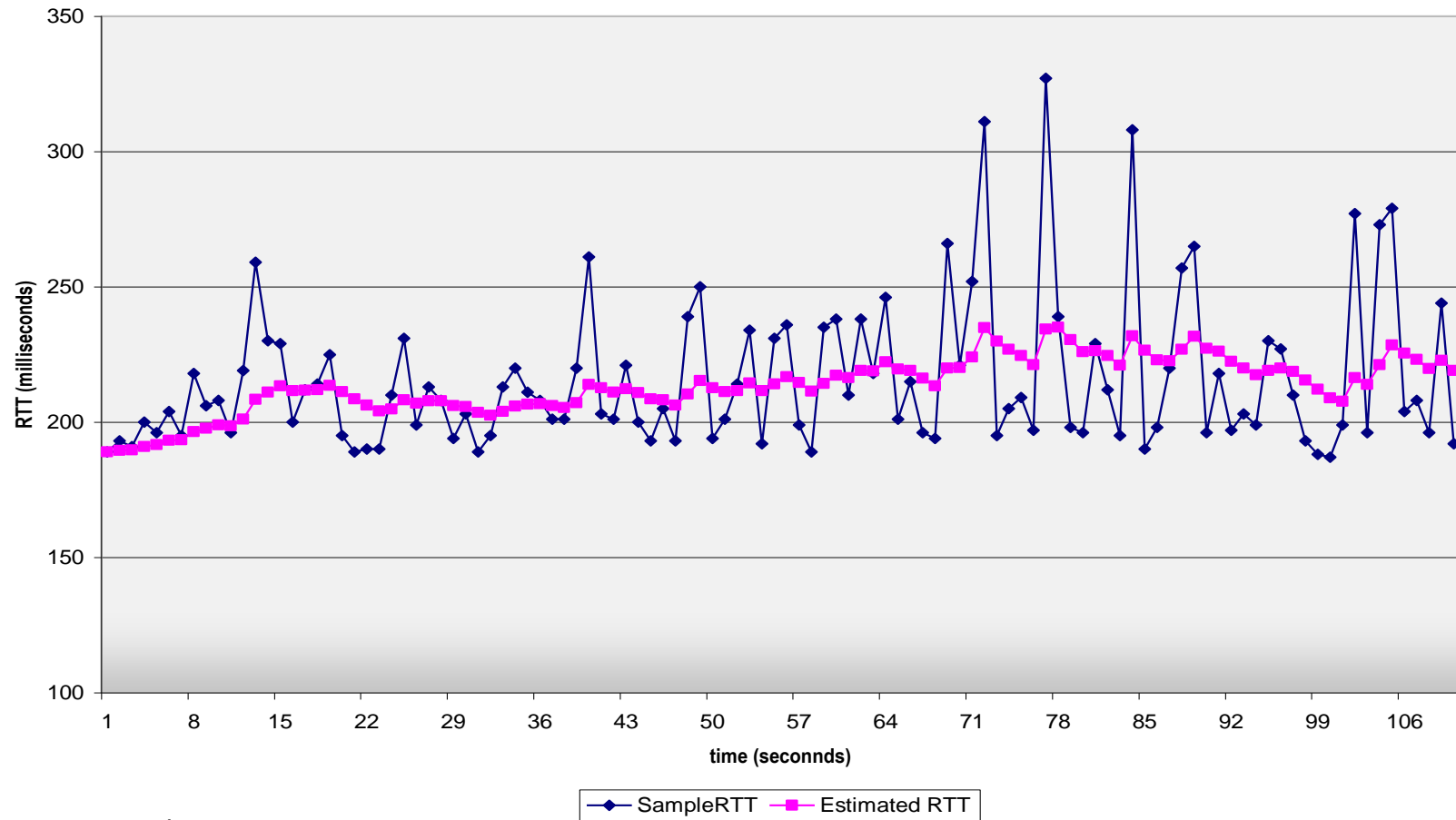
□ עלינו לבחור RTT שבמידה ויהיה קצר מדי הדבר יגרום להרבה שליחות כפולות (retransmission) ובכך העלאת התקורה (המידע המצורף להודעה ברשת כדי להבטיח העברה נקייה מטעויות ליעד הנכון), ובמידה ויהיה ארוך מדי הדבר יגרום לאיטיות ואובדן מידע.

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

**SampleRTT** – מודד את הזמן משליחת הסיגמנט ועד לקבלת האישור (ack).

# RTT *קצוץ אקראי*

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr



תקשורת מחשבים ואלגוריתמים מבזרים  
(חורף 2010)

# שכבת ההובלה

---

□ עקרונות של בקרת עומסים.

□ בקרת עומסים TCP.

□ שרותי שכבת ההובלה.

□ ריבוב/פילוג.

□ תעבורה חסרת קשר: UDP.

□ עקרונות תעבורת מידע

אמינה ברשת.

□ תעבורה מונחת קשר: TCP

■ מבנה המקטע.

■ תעבורת מידע אמינה.

■ בקרת זרימה.

■ ניהול הקשר

# TCP תעבורת מידע אמין

---

- TCP יוצר שירות rdt על קצה של שירות ה-IP הלא אמין.
- מקטעי Pipelined.
- הצטברות של acks.
- TCP משתמש ב-timer (תזמן) יחיד לשליחה מחדש.
- השליחה מחדש מתעוררת כתוצאה מ:
  - פסק זמן של אירועים.
  - שיכפול של acks.
- בתחילה לצורך הפשטות ה-TCP השולח:
  - מתעלם מ-acks משוכפלים.
  - מתעלם מבקרת הזרימה ועומסי הזרימה.

# TCP *מחזוריות קנדל?*

```
NextSeqNum = InitialSeqNum
```

```
SendBase = InitialSeqNum
```

```
loop (forever) {  
    switch(event)
```

```
    event: data received from application above
```

```
        create TCP segment with sequence number NextSeqNum
```

```
        if (timer currently not running)
```

```
            start timer
```

```
        pass segment to IP
```

```
        NextSeqNum = NextSeqNum + length(data)
```

```
    event: timer timeout
```

```
        retransmit not-yet-acknowledged segment with  
        smallest sequence number
```

```
        start timer
```

```
    event: ACK received, with ACK field value of y
```

```
        if (y > SendBase)
```

```
        {
```

```
            SendBase = y
```

```
            if (there are currently not-yet-acknowledged segments)
```

```
                start timer
```

```
        }
```

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

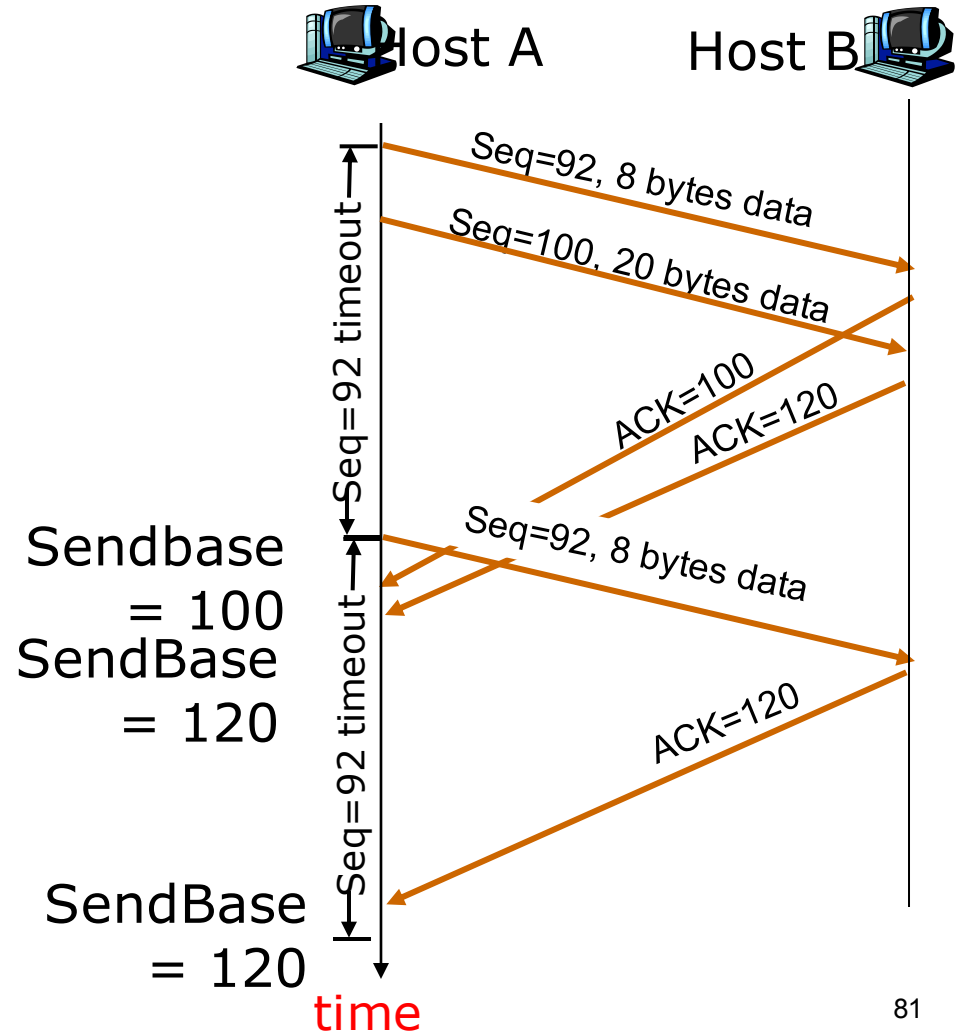
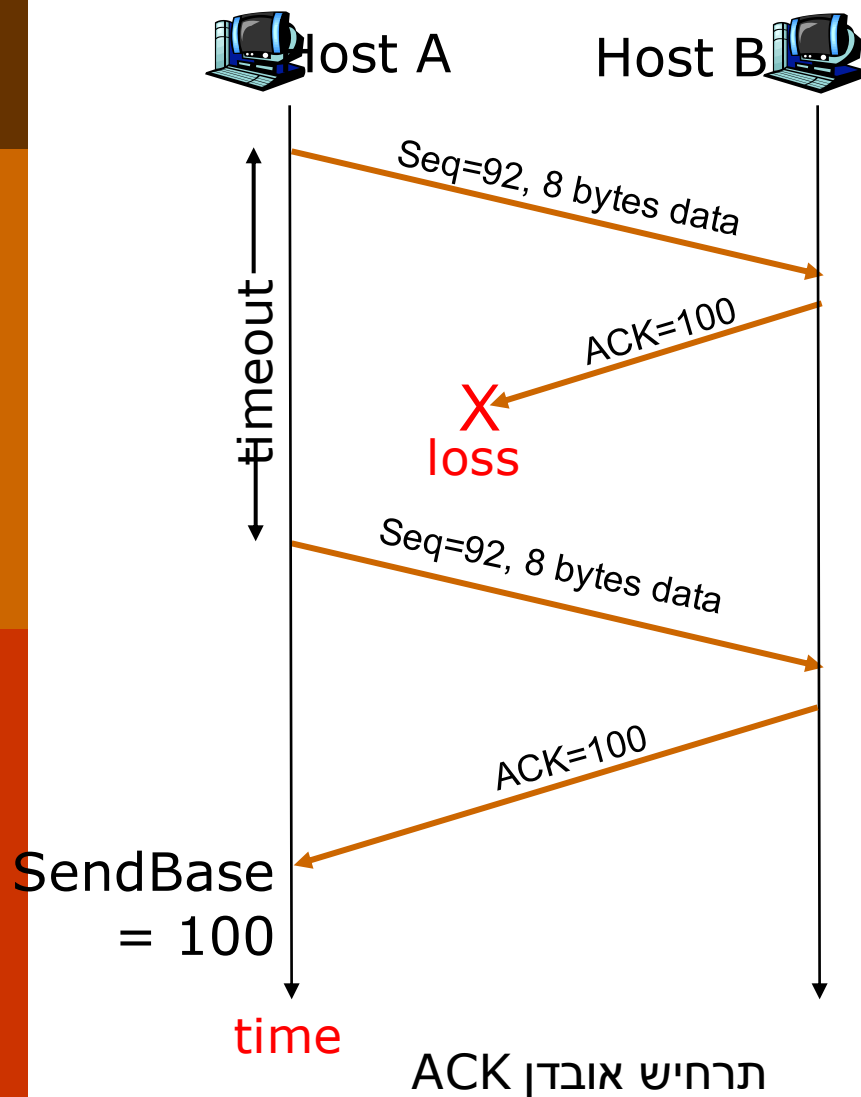
## Comment:

- **SendBase-1**: last Cumulatively ack'ed byte

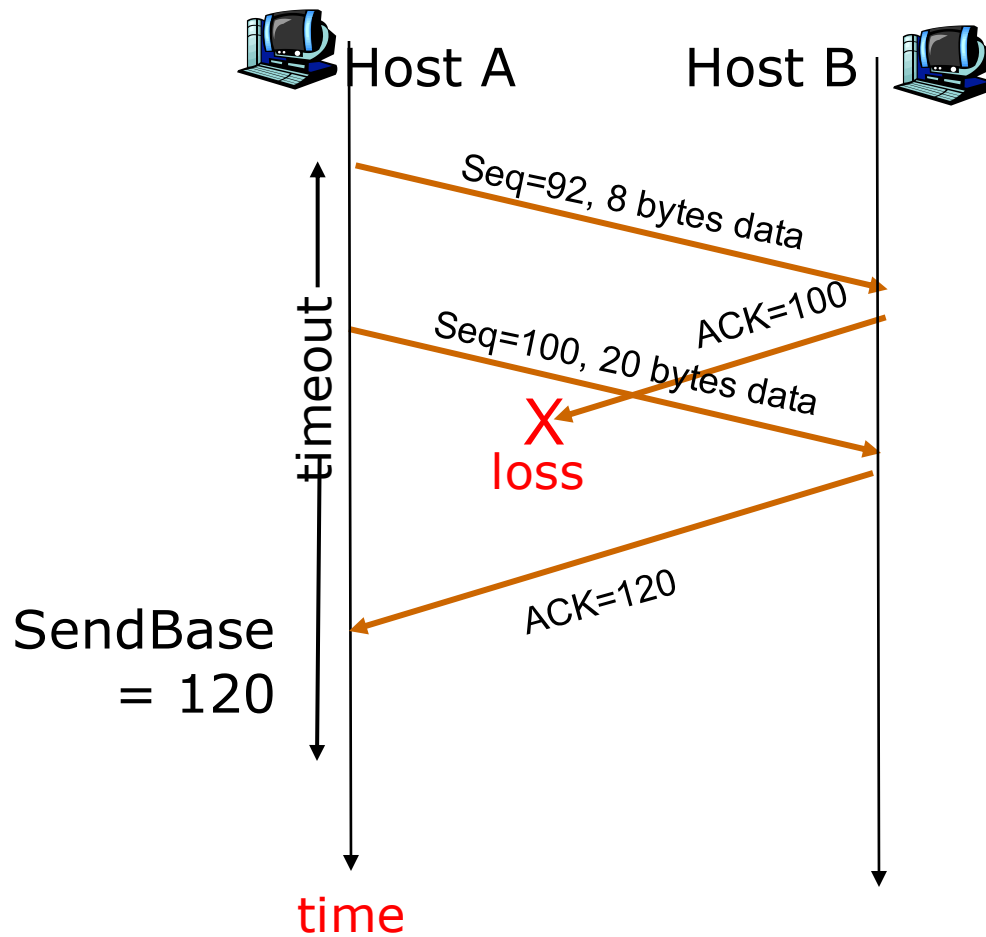
## Example:

- **SendBase-1 = 71**;  
**y = 73**, so the rcvr wants 73+;  
**y > SendBase**,  
so that new data is acked

# תרחיש שליחה מחדש ב-TCP



# תרחיש שליחה מחדש TCP-ה



## TCP-ACK timer

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expected seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap



# שכבת ההובלה

---

- שרותי שכבת ההובלה.
- ריבוב/פילוג.
- תעבורה חסרת קשר: UDP.
- עקרונות תעבורת מידע אמינה ברשת.
- תעבורה מונחת קשר: TCP
  - מבוא.
  - מבנה המקטע.
  - תעבורת מידע אמינה.
  - בקרת זרימה.
  - ניהול הקשר
- עקרונות של בקרת עומסים.
- בקרת עומסים TCP.

# בקרת זרימה - TCP

■ **הבעיה:** האפליקציה במחשב המקבל יכולה להיות איטית יותר מאשר הרשת ← לא יהיה מקום למידע המתקבל.

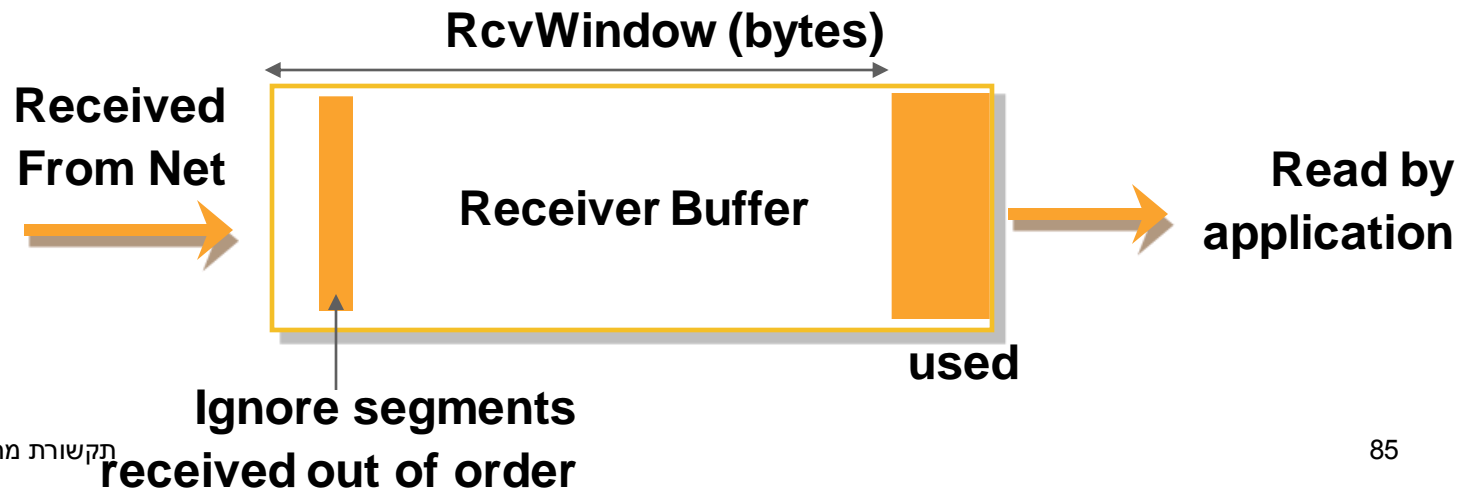
■ **בקרת הזרימה:** למנוע מהמחשב המקבל גלישת מידע בחוצץ, כלומר נחפש מנגנון שבאמצעותו דואגים שלא תשלח יותר אינפורמציה מאשר המקבל יכול לשמור אצלו בחוצצים. ולא נגיע למצב שבו החוצצים מלאים כאשר האפליקציה עוד לא קראה מהזיכרון.

■ נעשה זאת ע"י הגבלה של מקטעים ללא ACK במעבר.

■ המקבל מקצה מקום למידע שהוא אמור לקבל מהשולח – RcvBuffer ומסמן לשולח כמה מקום נותר לו פנוי לאורך זמן החיבור – RcvWindow: כלומר

$$: RcvWindow = (|Rcvbuffer| - |used|) / MSS$$

■ **MSS (Max Segment Size)** - גודל סגמנט מקסימלי בביטים.



## בקרת זרימה – TCP

- **פתרון:** השולח מגביל את נפח המנות שהוא יכול לשלוח ללא קבלת ACK, נפח זה הוא כנפח ה-RcvWindow. זאת כדי למנוע הצפה של החוצצים של המקבל.
- ה-ACKים שמתקבלים בשולח, לא תמיד מגיעים לפי סדר השליחה המקורי.
- סיבות לכך שהחוצץ מלא:
  - חבילה ישנה לא התקבלה (בשולח לא התקבל ACK עליה).
  - האפליקציה במקבל לא קראה עדיין מידע מהחוצצים.

## בקרית זרימה - TCP: שדה RcvWindow

Source Port		Destination Port	
Sequence Number			
Acknowledge Number			
Hdr Len	RS V	Flags	RcvWindow
Checksum		Urgent pointer	
Options (variable length)			
Payload (Application Data)			

RcvWindow: □

■ שדה Flow control

ב-TCP header

■ מספר מקסימאלי של

מקטעים ללא ACK

למחשב עמית

היכולים להשלח.

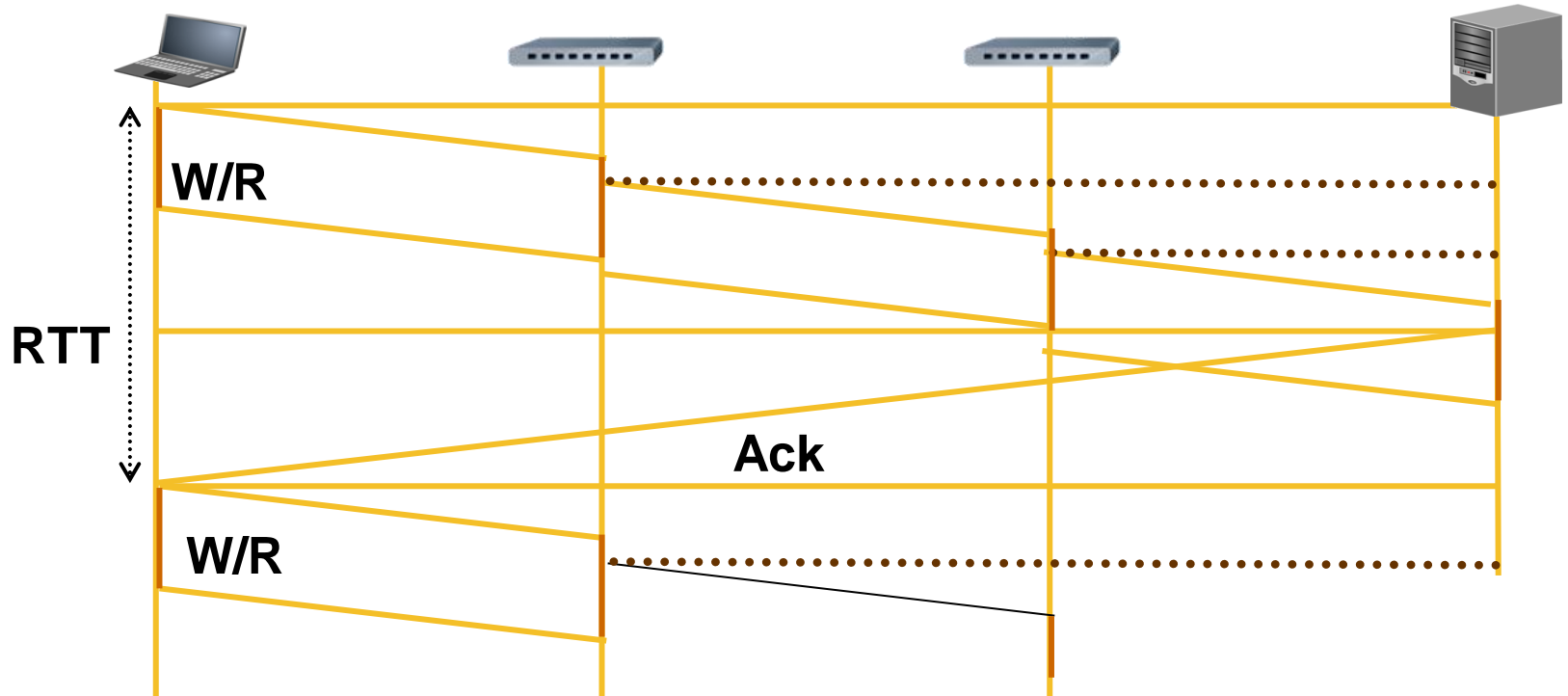
סיכום: שדה ה-RcvWindow הוא מנגנון בקרת זרימה המרכזי של TCP. הוא מציין את המספר המרבי של מקטעים שהשולח יכול לשלוח לפני שהוא מקבל ack. שדה זה נשלח בדרך כלל מתחנת הנמען, כאשר מקבלים acking של מקטע, אבל אם צריך גם המחשב המקבל רשאי לשלוח מקטע כדי להודיע לשולח על עלייה ב-RcvWindow.

# גודל החלון (1)

- גודל החלון הוא אלמנט קריטי עבור הביצועים
  - אם יותר RcvWindow מידי קטן, הביצועים יפגעו.
  - אפילו אם הרשת זמינה, האפליקציות יפגעו.
- דוגמא: נסתכל על רשת התקשורת הפשוטה הבא:
  - לקוח ושרת מחוברים ע"י שני נתבים.
  - $W = \text{גודל החלון (ביטים)}$
  - $R = \text{קצב השליחה של כל הערוצים.}$
  - $MSS = \text{גודל סגמנט מקסימלי בביטים.}$
  - $RTT = \text{Round Trip Time (RTT} \gg MSS/R \text{ (זמן המחזור)}$

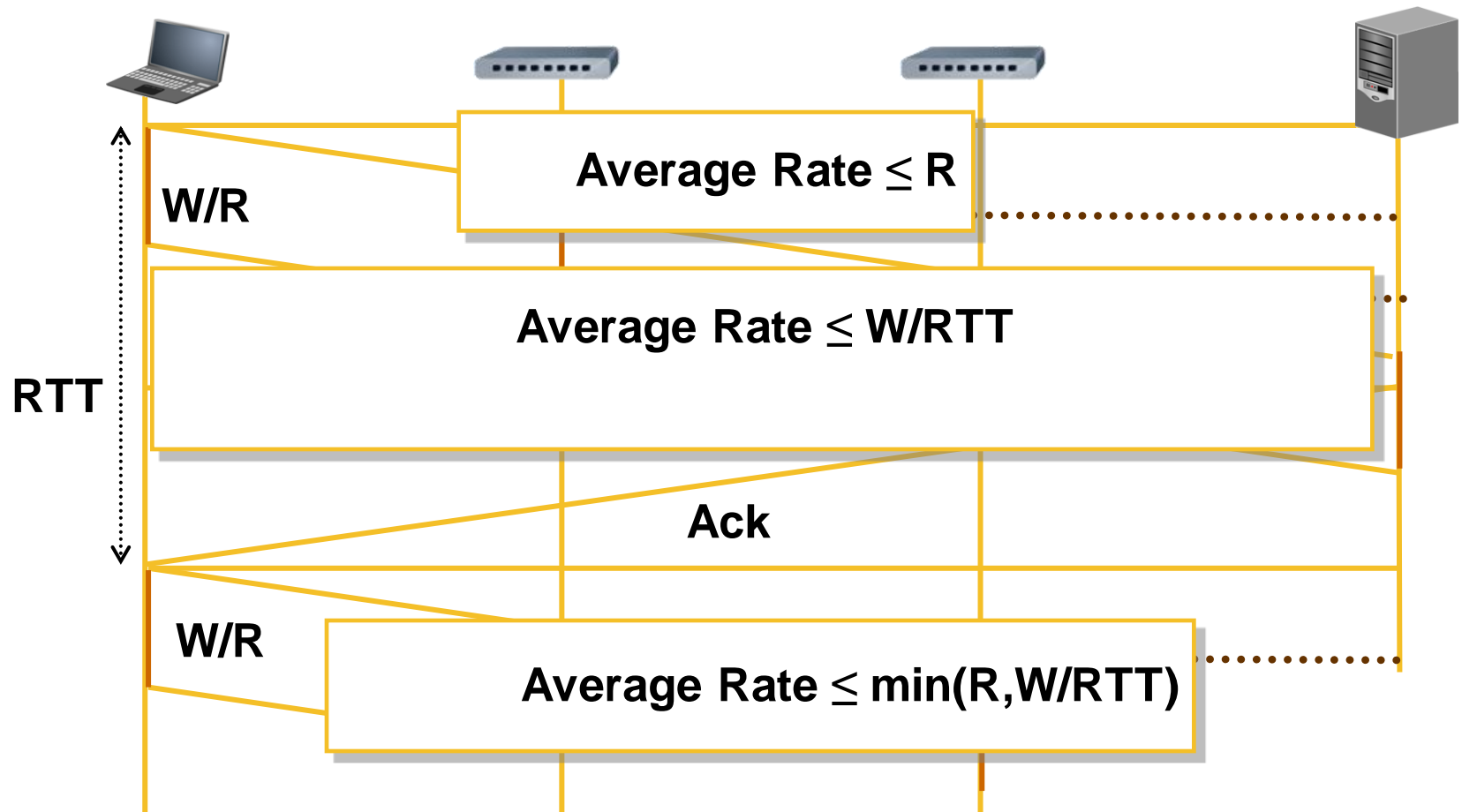


## מודל החלון (2)



כאן אנו רואים את ההשפעה של גודל החלון: נסמן את גודל החלון -  $W$ , ואת הקצב השידור  $R$ , מכאן שזמן שידור הוא  $W/R$ . נניח, כמו באיור, כי זמן המחזור  $RTT$  (הזמן העובר מתחילת שידור החבילה בצד השולח ועד קבלת הביט האחרון של חבילת האישור נשלחת ע"י מחשב היעד עם סיום קליטת חבילת המידע) גדול בהרבה  $W/R$ . מה שקורה עכשיו הוא שאנחנו שולחים את כל חלון  $W$  ויש לי לחכות ל-ack (אשר יקח כמובן  $RTT$ ). ואז במשך  $(RTT - W/R)$  הערוץ לא היה בשימוש!

# הקצב הממוצע



הקצב הממוצע – פונקציה של זמן שימוש הערוץ מתוך זמן האפשרי לשימוש בערוץ נחפש את המינימום מבין הקצב השידור  $R$  והיחס שבין גודל החלון  $W$  לזמן המחזור  $RTT$ .

# חלון קבלה אופטימאלי

□ קצב שידור ממוצע  $\min\{R, W/RTT\}$

□ בקרת זרימה אינה צוואר בקבוק כל עוד:  $W \geq R * RTT$  (גודל החלון  $W$  חייב להיות גדול ממכפלת קצב השידור בזמן המחזור על מנת לא לגלוש מהחוצץ)

□ החוצץ המקבל "ReceiverBuffer" שווה ל:

$$\text{ReceiverBuffer} \geq \text{Bandwidth} \times \text{Delay (עיכוב)} \text{ (רוחב פס)}$$

□ אם החלון הוא קטן מידי: ירידה במהירות החיבור.

□ ערוץ "שמן" - לדוגמא לוויין רוחב פס גדול שווה ל:

Fat links: huge **Bandwidth x Delay** (e.g. satellite)

□ בקרת זרימה אגרסיבית: נשלח RcvWindow ארוך יותר מאשר החוצץ האפשרי.

■ לפי הערכת קצב "הניקוז", והסיכון של "הצפת" החוצץ

■ פועל היטב כאשר העיכוב אינו משנה הרבה



## חוזק TCP: משאק קריטי

- חוצץ קבלה TCP: קריטי עבור ביצועי המהירות.
- אבל חוצצים גדולים מגבילים את מספר החיבורים
  - 10,000 חיבורים, 10KB לכל חיבור  $\leftarrow$  100 MB.
  - השלכות על העלויות והביצועים.
- השרת חייב למזער את כמות החיבורים הפתוחים
  - הלקוחות חייב לסגור חיבורים (ולהמתין 30 שניות).
  - שרתים רבים יזמו סגירה, אבל לא ימתינו 30 שניות
  - אם ה-Ack אבד, הלקוח עלול לקבל פסק הזמן (לסגור באופן חריג)

## חלון אדול...

- נדרש עבור הרשת חוצץ גדול, המקבל
  - ברירת מחדל, באופן שמרני: חוצץ המקבל  $<$  חלון.
  - אגרסיבי, בקרת זרימה אופטימאלית: חוצץ  $>$  חלון (ע"י הערכת קצב הניקוז, סיכון של הצפת החוצץ).
- בייטים רבים יכולים להישלח ל-'חיבורים שבורים' או רשת עמוסה.
- תורים גדולים בנתב ← עיכוב, עומסים, אובדנים... לא טוב!
- פתרון: להגביל את גודל החלון גם ע"י בקרת עומסים.

# שכבת ההובלה

---

- שרותי שכבת ההובלה.
- ריבוב/פילוג.
- תעבורה חסרת קשר: UDP.
- עקרונות תעבורת מידע  
אמינה ברשת.
- תעבורה מונחת קשר: TCP
  - מבוא
  - מבנה המקטע.
  - תעבורת מידע אמינה.
  - בקרת זרימה.
  - ניהול הקשר.
- עקרונות של בקרת עומסים.
- בקרת עומסים TCP.

# הקמת קשר – TCP

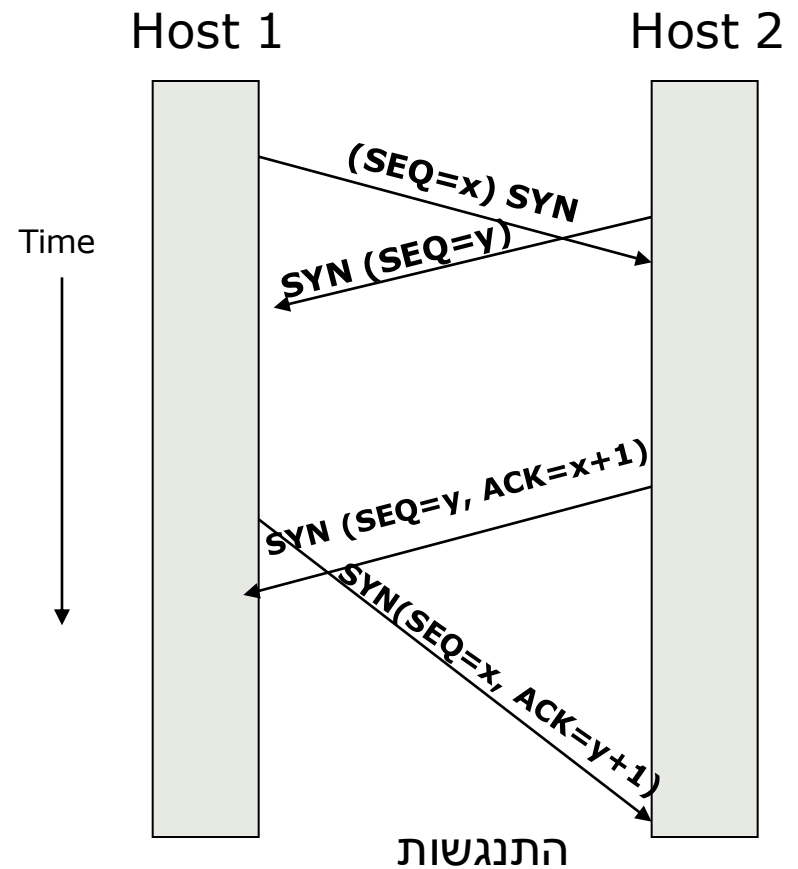
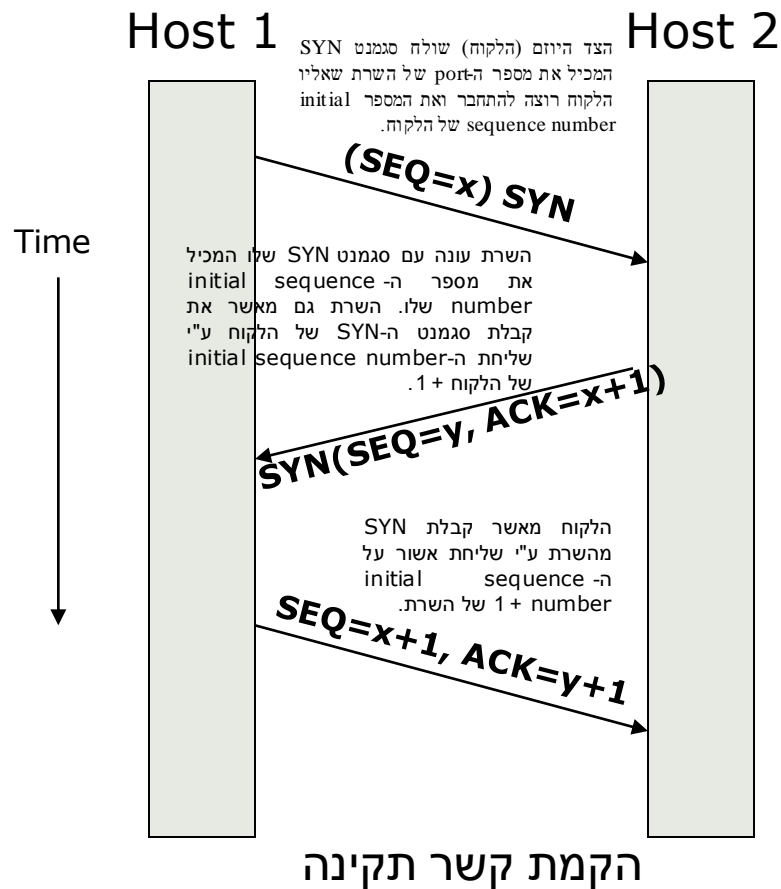
- הקמת הקשר עובדת לפי לחיצת יד משולשת והמספרים הסידוריים בנויים לפי שעון למקרה של התרסקות, כפי שראינו קודם.
- כמו כן, בעת התרסקות של תחנה, היא ממתינה זמן מחזור שלם של הודעה עד לשלב בו היא תתאושש.
- אלגוריתם ההתקשרות three way handshake פועל כך:
  1. הלקוח שולח הודעה ובה ה sequenceNum הראשון (flag=SYN ו-  $x = \text{sequenceNum}$ ).
  2. השרת שמקבל את ההודעה מאשר את קבלת המשלוח, ושולח את המספר הסידורי של המשלוח שלו.  
(SYN+ACK:  $\text{sequenceNum}=y$ ,  $\text{acknowledgment}=x+1$ )
  3. הלקוח מחזיר אישור על הודעת השרת  
(ACK:  $\text{acknowledgment} = y+1$ )
- לשתי ההודעות הראשונות מתוך השלושה נלקח timer, ואם לא מגיעה תגובה בזמן המוגדר, ההודעה נשלחת שוב. ההודעה השלישית לא מגובה וגם אם לא תתקבל התקשרות תתחיל.
- הסיבה לשליחת ה sequenceNum היא שהפרוטוקול דורש שכל צד יבחר בתור מספר הודעה הראשון מספר רנדומלי, ולא אפס או מספר מוסכם אחר כיוון שאם במקרה עדיין מגיעים דרך הרשת סגמנטים מהתקשרויות קודמות – הפרוטוקול לא יתבלבל בינם ובין ההודעות מה-session הנוכחי.

## ניהול חיבור ה-TCP

---

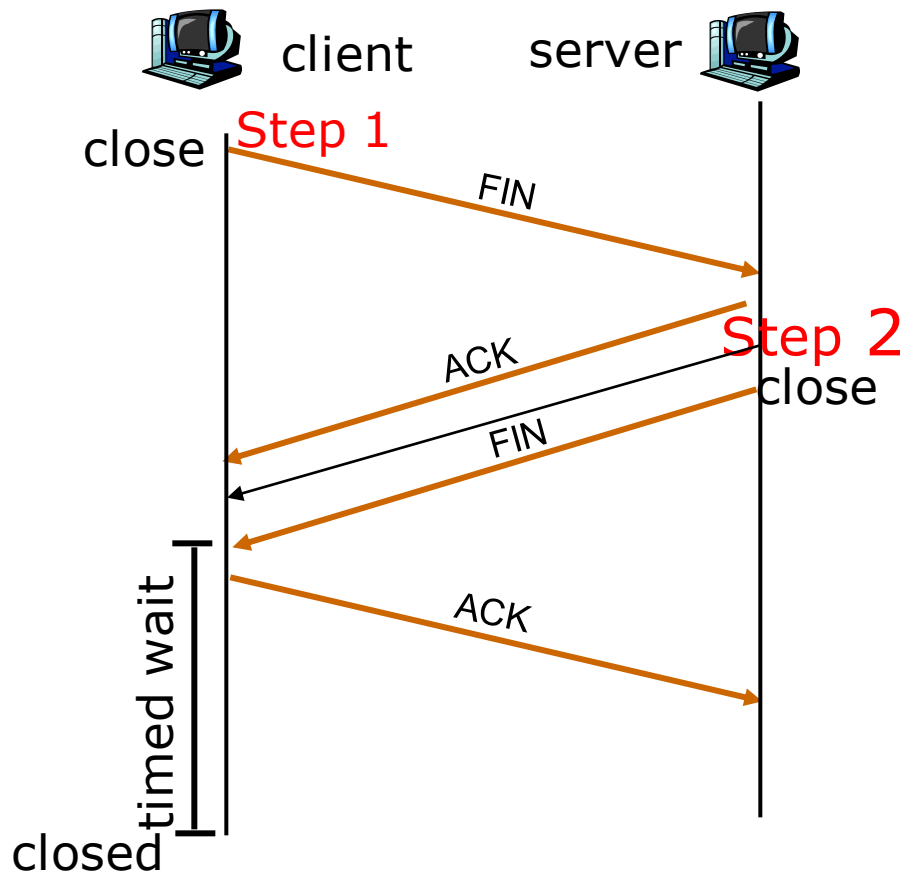
- מספור הבתים הנשלחים יעשה על פי המספר הסידורי. המספר עולה על פי מספר הבתים שנשלחו.
- את החיבור חשוב לסגור כיוון שצריך לשחרר את המשאבים שהוקצו וכדי לדעת שכל ההודעות שנשלחו התקבלו.
- הסגירה יכולה להתחיל מהשרת או מה-client, ע"י שליחת הודעת FIN, עליה ישלח ACK. אם לא התקבל ה-ACK על ה-FIN אז ה-FIN נשלח שוב. יש המתנה לאורך זמן מסוים כדי לוודא שלא מתקבל עוד FIN, כלומר, שיהיה בטוח שהתקבל ה-ACK על ה-FIN. ההמתנה נעשית ב-client ולא בשרת כיוון עסוק ועובד מול הרבה לקוחות.

# הקמת קשר ה-TCP



שליחת SYN: ה-Host שולח מספר סידורי אקראי.  
שליחת ACK: ה-ACK הוא עם המספר הסידורי שהתקבל מה-Host. יכול גם להכיל כבר מידע.

# ניהול חיבור ה-TCP



## סיום ההתקשרות:

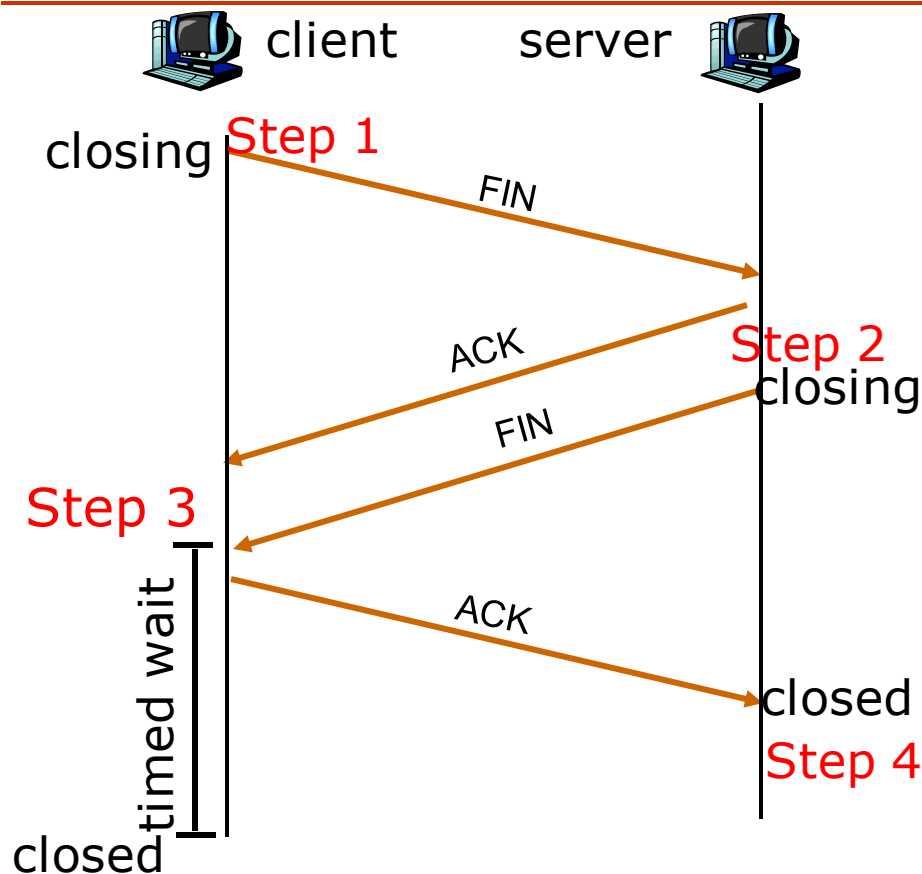
:client closes socket

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

צעד 1: המחשב לקוח שולח  
מקטע "TCP FIN control"  
לשרת.

צעד 2: השרת מקבל **FIN**,  
מגיב עם **ACK**, סוגר  
תקשורת ושולח **FIN**.

# ניהול חיבור ה-TCP



צעד 3: הלקוח מקבל **FIN** ומחזיר **ACK**.

■ הזנת "timed wait" - תענה עם **ACK** לקבלת **FINs**.

צעד 4: השרת מקבל **ACK**, תקשורת נסגרת.

**סיכום:** כדי לסיים קשר TCP יש צורך בשליחה של 4 סגמנטים. הסיבה לכך נעוצה בתכונה של TCP הנקראת "half-close". מאחר וקשר TCP הוא full-duplex, כלומר מידע יכול לזרום בשני הכיוונים ללא תלות בצד השני, כל צד צריך לסיים את הקשר שלו ללא תלות בצד השני. הכלל אומר שכל צד יכול לשלוח **FIN** ברגע שהוא סיים להעביר את המידע. כאשר TCP מקבל **FIN** הוא חייב להודיע לאפליקציה על כך שהצד השני סיים את הקשר בכיוון שלנו. כמו ביצירת קשר אומרים שהצד ששולח את ה-**FIN** מבצע "active close". הצד השני שמקבל את ה-**FIN** הראשון מבצע "passive close".



## ניהול חיבור ה-TCP

---

- כדי לסגור את החיבור, אנו חייבים לוודא שני הצדדים נסגרו (כל ההודעות שנשלחו).
- המחשב המגיב מקבל את הודעה שהמחשב היוזם סגר את התקשורת בסדר, אבל היוזם יכול רק להמתין לראות אם הוא לא יקבל שוב FIN.
- יש לכן אפשרות קלה שהיוזם יסגור עם שגיאה ואילו המגיב יסגור היטב וכל ההודעות משני הצדדים יועברו בצורה נכונה

# ניהול חיבור ה-TCP

□ מצבי הלקוח:

■ Fin\_Wait1 (FW1): אין יותר שליחות, מחכה ל-Ack על ה-FIN ששלח.

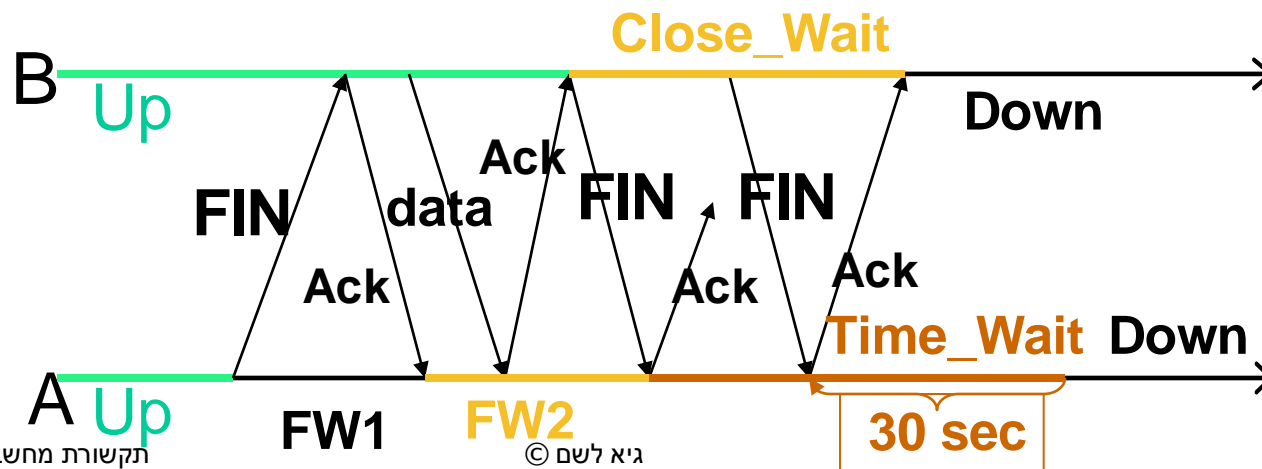
■ Fin\_Wait2 (FW2): כנ"ל, עדין מחכה, ל-FIN.

■ Time\_Wait: מחכה 30sec לשלוח ACK אם אנו נקבל FIN חדש.

□ היוזם: שולח FIN, מחכה ל-Ack ול-FIN.

□ המגיב: ACK ל-FIN, ההודעה האחרונה, FIN

■ מתבצע כאשר מקבלים ACK ל-FIN.



# שכבת ההובלה

---

□ עקרונות של בקרת עומסים.

□ בקרת עומסים TCP.

□ שרותי שכבת ההובלה.

□ ריבוב/פילוג.

□ תעבורה חסרת קשר: UDP.

□ עקרונות תעבורת מידע

אמינה ברשת.

□ תעבורה מונחת קשר: TCP

■ מבנה המקטע.

■ תעבורת מידע אמינה.

■ בקרת זרימה.

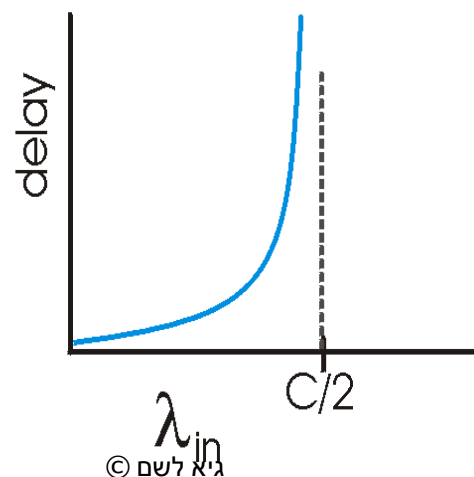
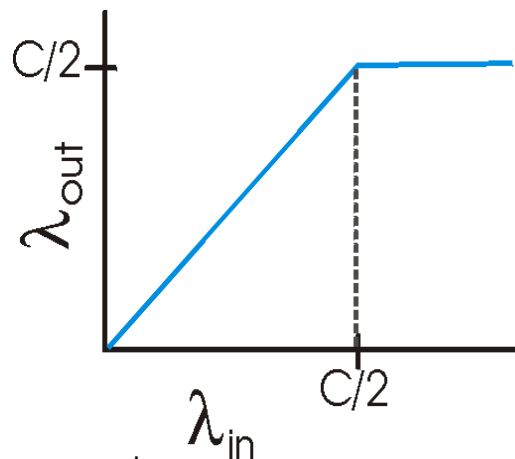
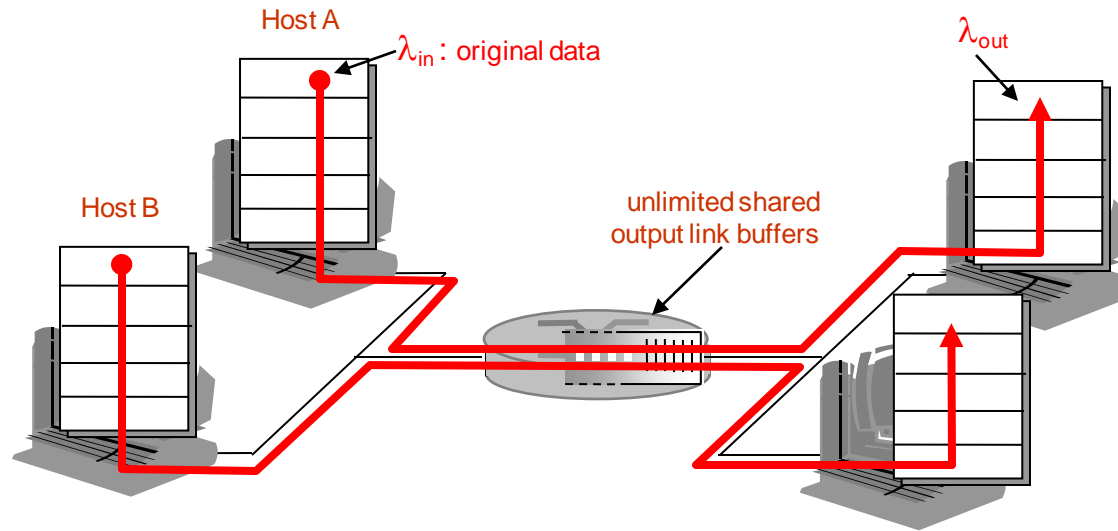
■ ניהול הקשר

## צקרונות בקרת עומס

---

- בעומס יתר תורי ההמתנה בנתבים נעשים ארוכים וכך נוצרות השהיות ארוכות.
- כמו כן, מנות הולכות לאיבוד.
- ההבדל בין בקרת עומס לזרימה הוא שבעומס מתמקדים בחוצצים של הנתבים ובזרימה בחוצצים של מחשבי הקצה.
- אין שליחת אישורים מהנתבים, תפקידם היחיד הוא העברת החבילה ליעד הבא.

# סיבות וצלויות של עומס - מקרה 1



תקשורת מחשבים ואלגוריתמים מבזרים  
(חורף 2010)

גיא לשם ©

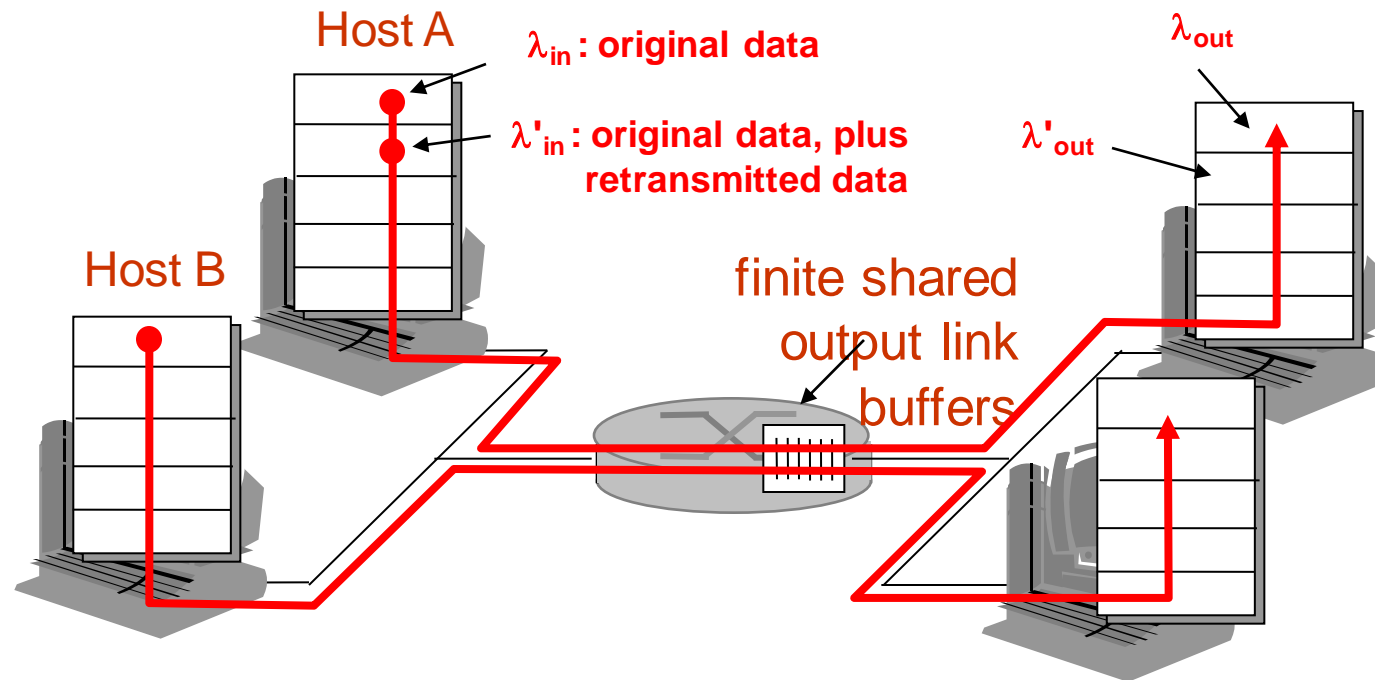
- ישנם 2 שולחים ו-2 יעדים וביניהם נתב עם חוצץ אינסופי.
- אין שליחה חוזרת.
- $\lambda$  - קצב שידור ממוצע של בתיים לשניה.
- עיכוב גדול כאשר יש עומס.
- התפוקה מקסימלית

## סיבות וצלויות $fe$ צומס – מקרה 1 (הסבר)

- ישנם 2 שולחים ו-2 יעדים וביניהם נתב.
- בנתב יש חוצץ אינסופי. גודל הערוץ (רוחב פס) הוא  $C$ .
- לנתב נדרש זמן כדי לקבוע לאן לשלוח את המנה שהוא מקבל.
- אם נתב מקבל משני השולחים בו-זמנית אז קצב השליחה שלו לכל יעד יהיה קטן מ- $C/2$ . זאת כיוון שכל פעם בממוצע הוא שולח מנה שהתקבלה משולח אחר + הזמן שלוקח לו להחליט לאן לשלוח את המנה. לכן כאשר קצב קבלת המנות אצל הנתב הוא  $C/2$  או יותר החוצצים יתמלאו עד אינסוף (קצב הקבלה גדול מקצב השליחה) וכך גם ההשהיה תלך ותגדל.
- $\lambda$  – קצב שידור ממוצע של בתים לשניה.
- הסבר לגרף שמאל: קצב השליחה מנקודת הקצה  $\lambda_{in}$  הולך וגדל לעומת קצב הקבלה אצל נקודת היעד  $\lambda_{out}$  שלא עולה על  $C/2$  בגלל קצב השליחה מהנתב.
- הסבר לגרף ימין: ככל שקצב השליחה מנקודת הקצה  $\lambda_{in}$  מתקרב ל- $C/2$  כך ההשהיה נעשית גדולה יותר (עד אינסוף).

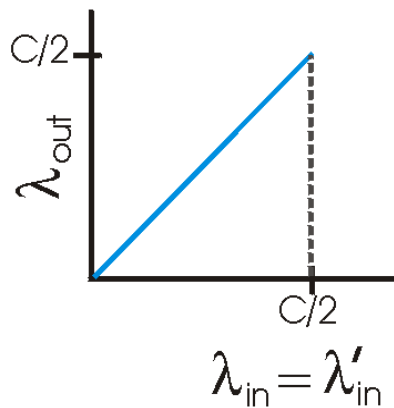
## סיבות וצלויות של צומס - מקרה 2

- ישנם 2 שולחים ו-2 יעדים
- נתב אחד עם מספר חוצצים סופי.
- שליחה חוזרת של חבילות אבודות.

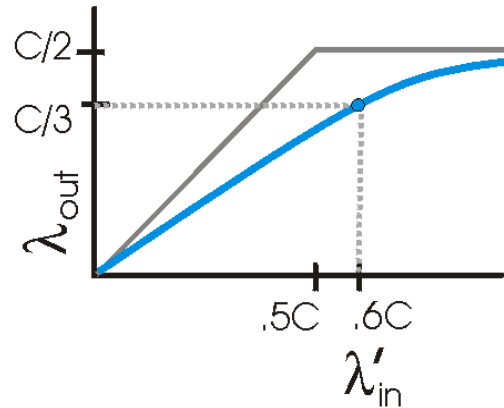


## סיבות וצלויות $fe$ צומס - מקרה 2 (הסבר)

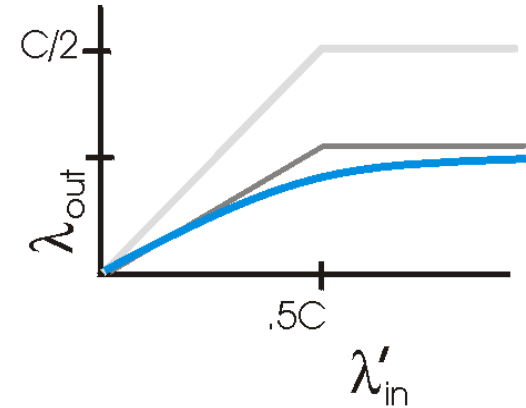
- תמיד  $\lambda_{in} = \lambda_{out}$  (good put - השידור אמין).
- שליחה חוזרת מושלמת רק באובדן  $\lambda'_{in} > \lambda_{out}$ .
- שליחה חוזרת של העיכוב (לא אובדן) החבילה מבצעת  $\lambda'_{in}$  ארוך יותר (מהמקרה המושלם) עבור אותו  $\lambda_{out}$ .



(a)



(b)



(c)

$\lambda'_{in}$  - קצב השידור כולל השידור החוזר, גדול מ- $\lambda_{in}$ .

**"מחיר" של העומס:**

□ יותר עבודה (חוזרת) עבור goodput נתון

□ אין צורך בשליחה חוזרת: הערוץ נושא מספר עותקים של החבילה.



## סיבות וצלויות *fe* צומס – מקרה 2 (הסבר)

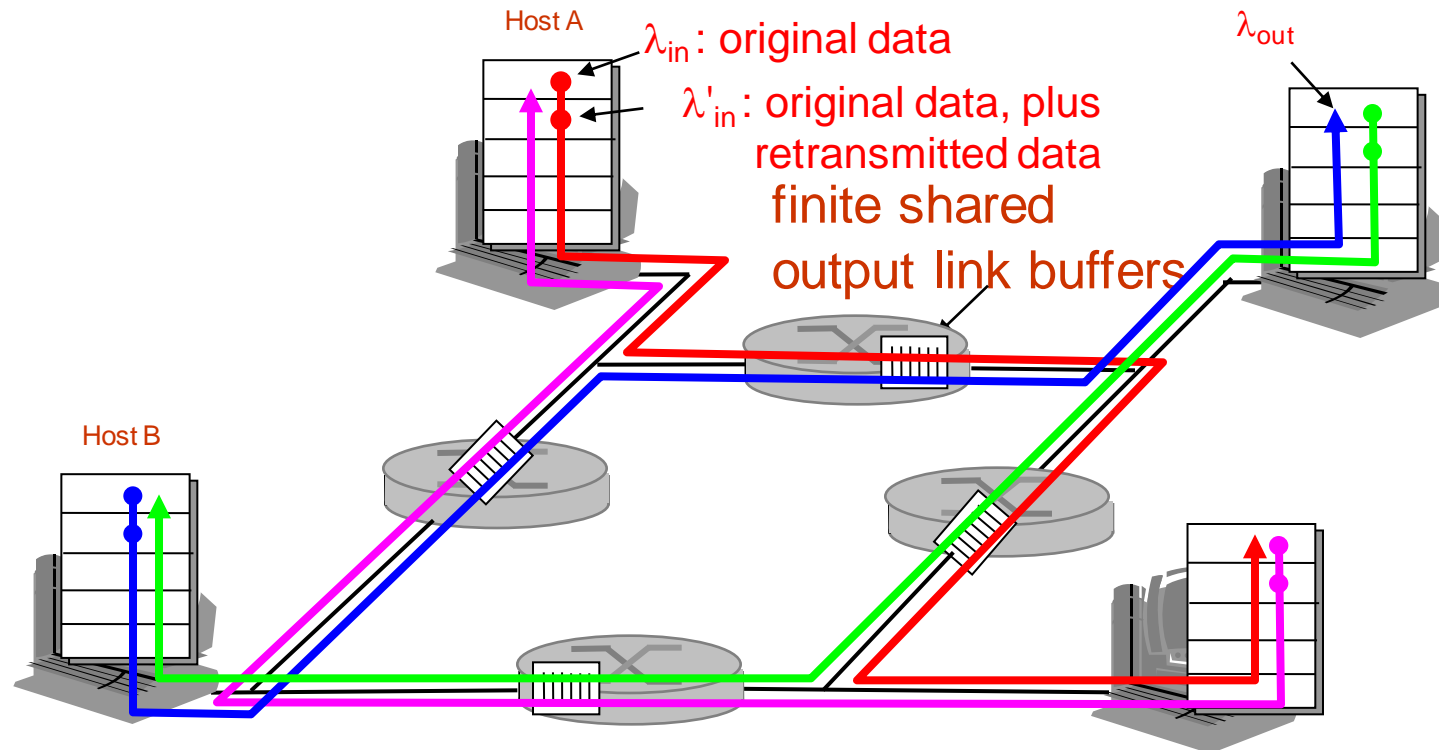
- כמו במקרה 1 אך הפעם יש מספר חוצצים סופי. מספר חוצצים סופי גורם לכך ש- $\lambda_{out} = \lambda_{in}$  כלומר, השידור אמין, כל חבילה חייבת להגיע. יש גם שידור חוזר במקרה של גלישה בחוצץ או רעש או שיתבצע timeout.
- $\lambda'_{in}$  – קצב השידור כולל השידור החוזר, גדול מ- $\lambda_{in}$ .
- $\lambda'_{out}$  – קצב קבלת ההודעות ברמת ה-TCP כולל הודעות חוזרות, ההודעות שמגיעות משובשות לא עולות לרמה האחרונה לכן הוא גדול מ- $\lambda_{out}$ .
- הסבר לגרף שמאל: כאשר אין שידור חוזר, כלומר אין עומס והתור של הנתב ריק או לא מלא אז  $\lambda_{in} = \lambda'_{in}$ . מצב זה נוצר כאשר קצב השליחה קטן מ- $C/2$  וכל השולחים מקפידים לבצע את חוקי ה-TCP.
- הסבר לגרפים האמצעי והימיני: כאשר יש עומס ונוצר מצב של איבוד מנות הדורש שידור חוזר אז קצב קבלת המנות  $\lambda_{out}$  קטן כיוון שחלק מהמנות שהוא היה אמור לקבל ונשלחו, הוא לא קיבל.
- שימוש ב-GoBackN אשר דורש לשלוח את כל המנות שהתקבלו אחרי המנה שלא קיבלה ACK מחדש יחמיר את העומס.

■ ההשהיה תהיה ארוכה יותר כי יהיו יותר הודעות להעביר (המקוריות + המשובשות).

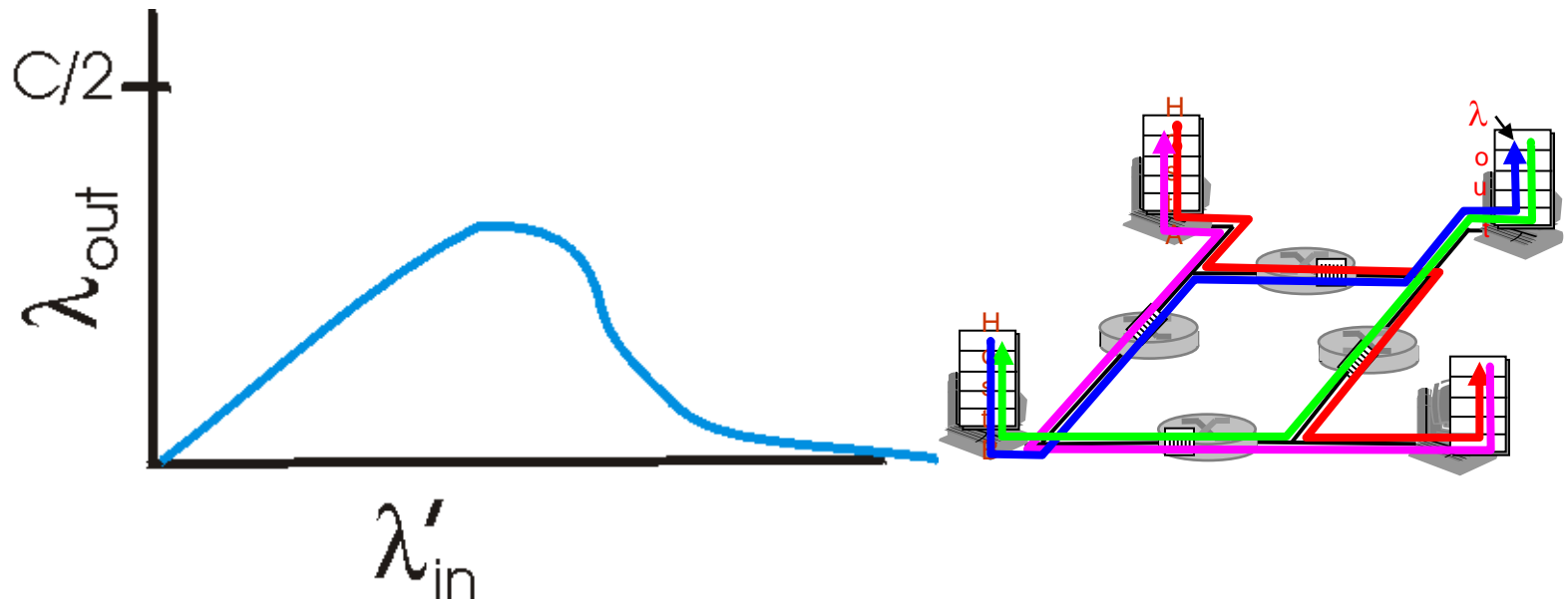
# סיבות וצלויות של צומס - מקרה 3

שאלה: מה קורה כאשר  $\lambda_{in}$  ו-  $\lambda'_{in}$  גדלים?

- ארבע שולחים.
- Multi-hop paths נתיב רב-HOP
- פסק זמן / שליחה מחדש



### סיבות וצלויות של עומס - מקרה 3



מחיר נוסף של העומס:

□ כאשר חבילה אובדת, כל יכולת השליחה של זרם נתונים

המשמש לחבילה זו היה בזבז !

## סיבות וצלויות *fe* צומס – מקרה 3 (הסבר)

---

- הוספת עוד ערוצים ונתבים ועוד נתיבי הגעה לכל יעד.
- הסבר לגרף: אם שולח אחד יתחיל לשדר מקצב מהיר יותר הוא יגרום בסופו של דבר למצב של deadlock. זאת כיוון שע"י שליחה מוגברת הוא יגרום לעומס בנתבים, מה שיביא לאיבוד מנות ושידור חוזר שלהם.
- השידור החוזר של המנות יגרום לעוד עליה בעומס וחוצצי הנתבים יתמלאו, כל זה יגרום לקריסה של הרשת.

# שכבת ההובלה

---

□ עקרונות של בקרת עומסים.

□ בקרת עומסים TCP.

□ שרותי שכבת ההובלה.

□ ריבוב/פילוג.

□ תעבורה חסרת קשר: UDP.

□ עקרונות תעבורת מידע

אמינה ברשת.

□ תעבורה מונחת קשר: TCP

■ מבנה המקטע.

■ תעבורת מידע אמינה.

■ בקרת זרימה.

■ ניהול הקשר

# בקרת עומס TCP

- **בקרת עומס מקצה לקצה** – מחשבי הקצה מבקרים את העומס ע"י בדיקה של כמות ההודעות המתקבלות משידור חוזר ורמת ההשהיה. פרוטוקולים חדשים חייבים להיות "ידידותיים" ל-TCP, כלומר לא יכולים לפגוע בתפקוד ה-TCP.
- **בקרת עומס בעזרת הרשת** – הנתבים מספקים מידע באורך ביט יחיד – יש או אין עומס. כמו כן, מסמנים את קצב שידור השליחה האפשרי.
- במידה וישנם נתבים ו/או חיבורים איטיים ברשת בין השולח למקבל עלולות להיווצר בעיות כאשר חבילות יאבדו או התעכבו כתוצאה מעומס זמני. בכדי לווסת את קצב כתיבת המידע לרשת ע"י השולח כך שכל המידע יגיע למקבל, משתמשים בין השאר, באלגוריתם "Slow start". אלגוריתם "Slow start" מוסיף עוד חלון ל-TCP של השולח, **Congestion Window**, הנקרא גם **cwnd** או **CongWin**.
- השולח מגביל את הביתים לשליחה לא רק ע"י **בקרת הזרימה** (למנוע מהמחשב המקבל גלישת מידע בחוצץ) אלה גם ע"י חלון העומס – **CongWin**. ה-**CongWin** הוא **מספר הבתים שנשלחו שעדיין לא קיבלו עליהם ACK**. ה-**CongWin** נותן אינדיקציה טובה על העומס ברשת ובעזרתו ניתן לדעת את הקצב הרצוי לשליחת המנות.
- ערכו של ה-**CongWin** משתנה בהתאם לזיהוי אובדן חבילות ברשת. ניתן לזהות אובדן חבילות ע"י המאורעות:

1. **Timeout**

2. **שלוש הודעות ACK זהות רצופות ( 3 duplicated ACK )**

# בקרת עומס TCP

- בקרת העומס היא מקצה לקצה.
- השולח מגביל את הבייטים לשליחה לא רק ע"י בקרת הזרימה אלא גם ע"י חלון העומס - CongWin.
- $\text{LastByteSent} - \text{LastByteAked} \leq \min(\text{CongWin}, \text{RcvWin})$
- ה-CongWin הוא מספר הבתים שנשלחו שעדיין לא קיבלו עליהם ACK.
- ה-RcvWin הוא מקום שנותר בפנוי בחוצץ לאורך זמן החיבור נעריך: ■
- ה-CongWin נותן אינדיקציה טובה על העומס ברשת ובעזרתו ניתן לדעת את הקצב הרצוי לשליחת המנות.
- ערכו של ה-CongWin משתנה בהתאם לזיהוי אובדן חבילות ברשת. ניתן לזהות אובדן חבילות ע"י המאורעות:
- ארוע אובדן: **timeout**, או שלוש הודעות **ACK** זהות רצופות.
- השולח מקטין את הקצב (CongWin) אחרי אובדן.
- שני אופנים לבקרת עומס:
- **Slow Start** (להתחיל לאט) – עם בניה מעריכית.
- **Congestion Avoidance** – להימנע מגודש.

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

# TCP מתחיל אט

ה-cwnd משמש כבקרת זרימה של השולח, בעוד ה-Advertised window (גודל החלון המרבי איתו התחנה המקבלת מוכנה לעבוד) משמש כבקרת זרימה של המקבל.

השולח מתחיל ע"י שידור של סגמנט אחד והמתנה ל-ack. כאשר ה-ack מתקבל גדל ה-cwnd לגודל של שני סגמנטים ואז השולח משדר שני סגמנטים.

כאשר מתקבלים ack על הסגמנטים גדל ה-cwnd להיות בגודל 4 סגמנטים וכך הלאה. התנהגות זאת מתארת גידול אקספוננציאלי של גודל חלון ה-cwnd.

במידה וישנן נתבים ו/או חיבורים איטיים ברשת בין השולח למקבל עלולות להיווצר בעיות כאשר חבילות יאבדו או התעכבו כתוצאה מעומס זמני.

בכדי לווסת את קצב כתיבת המידע לרשת ע"י השולח כך שכל המידע יגיע למקבל משתמשים, בין השאר באלגוריתם "Slow start".

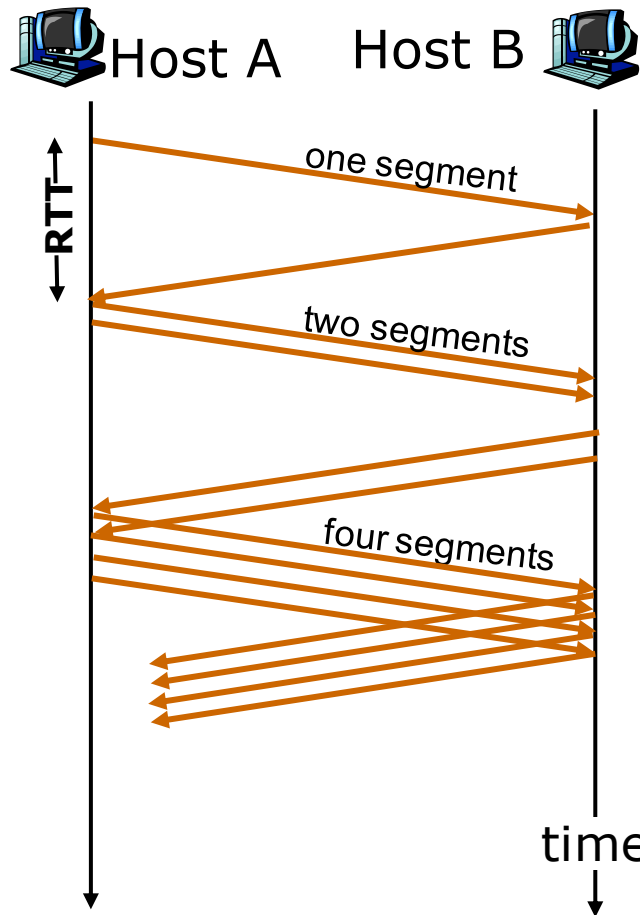
אלגוריתם "Slow start" מוסיף עוד חלון ל-TCP של השולח, Congestion Window, הנקרא גם cwnd. כאשר נוצר קשר TCP חדש עם מחשב ברשת אחרת, ה-cwnd מאותחל לגודל של סגמנט אחד.

בכל פעם שמתקבל ack, ה-cwnd גדל בסגמנט אחד.

השולח יכול לשדר עד שהוא מגיע למינימום בין ב-cwnd ל-Advertised window.



# TCP Slow Start



□ קצב בניה מעריכי

□ הקצב הראשוני איטי ( $\text{CongWin} = 1\text{MSS}$ ) אבל הוא עולה במהירות.

□  $\text{MSS} = \text{Maximal Segment}$ .

□ עד ארוע איבוד ראשון, הקצב עולה ע"י:

■ הכפלת  $\text{CongWin}$  לכל  $\text{RTT}$ .

■ ע"י הגדלה של  $\text{CongWin}$  עבור כל

קבלת  $\text{ACK}$ .

□ במקרה של אובדן: נשנה את הקצב על מנת

למנוע עומסים (ע"י פרוטוקול -  $\text{AIMD}$ :

Additive Increase, Multiplicative

(Decrease).

**סיכום:** בתחילת החיבור ערך ה- $\text{CongWin}$  הוא מינימלי. כל עוד אין אובדן מנות מכפילים את גודל

ה- $\text{CongWin}$  פי 2. כאשר מתחיל איבוד מנות מקטנים את ה- $\text{CongWin}$  פי 2 וצץ מגדילים אותו

# תצורה f – TimeOut – f 3dupACKs

## רעיון

- הרעיון מאחורי האלגוריתם הוא ההבדלה בין שני המאורעות שנגרמות עקב עומס ברשת.
- האלגוריתם מבדיל בין חומרתן של שתי המאורעות. שלוש ACK זהים רצופים מעידים על עומס גדול אך חמורים פחות מ-timeout. לכן, כאשר מקבלים שלוש ACK מפחיתים לחצי את ה-CongWin וכאשר מקבלים timeout מפחיתים למינימום האפשרי.
- אחרי שמתרחש timeout וגודל ה-CongWin קטן למינימום, ה-CongWin מתחיל לגדול אקספוננציאלית עד שהוא מגיע לחצי גודל ה-CongWin שבו התרחש ה-timeout (נקודת הסף) ואז הוא גדל לינארית.

□ אחרי 3 ACK זהים:

■ CongWin קטן בחצי  
(multiplicative decrease)

■ החלון לכן גדל באופן ליניארי  
(additive increase)

□ אחרי Timeout:

■ התחל מחדש חיבור.

■ CongWin נקבע ל-1 MSS.

■ גידול מעריכי עד לערך סף, ואח"כ גידול ליניארי.

# מופע ההתחלה האיטית

- ההתחלה איטית מתחילה עם חלון בגודל מקטע 1
  - אבל במהירות מאיץ: גדל ב-מקטע 1 לכל ACK (כלומר פי 2).
  - מהחבילה השניה, בהתחלה האיטית: ניתן לשלוח 2 מקטעים על ACK (זוג).
  - אחרי קבלת ה-ACK עבור  $W$  מקטעים:  $W + W = \text{Cong. Window}$
- לאחר אובדן האינדיקציה ( timeout או 3 dup acks )
  - $W = 1$ ,  $W_{th} = \lfloor W/2 \rfloor$ , נשלח מחדש חבילה ללא ACK ראשונה.
  - מאחר  $W = 1$ : מחכה ל-ACK (על חבילה אחת או יותר) הממשיך משם.
- כאשר,  $|window| \geq W_{th}$  (ערך הסף של ההתחלה האיטית):
  - משנה למופע 'מניעת עומסים'.

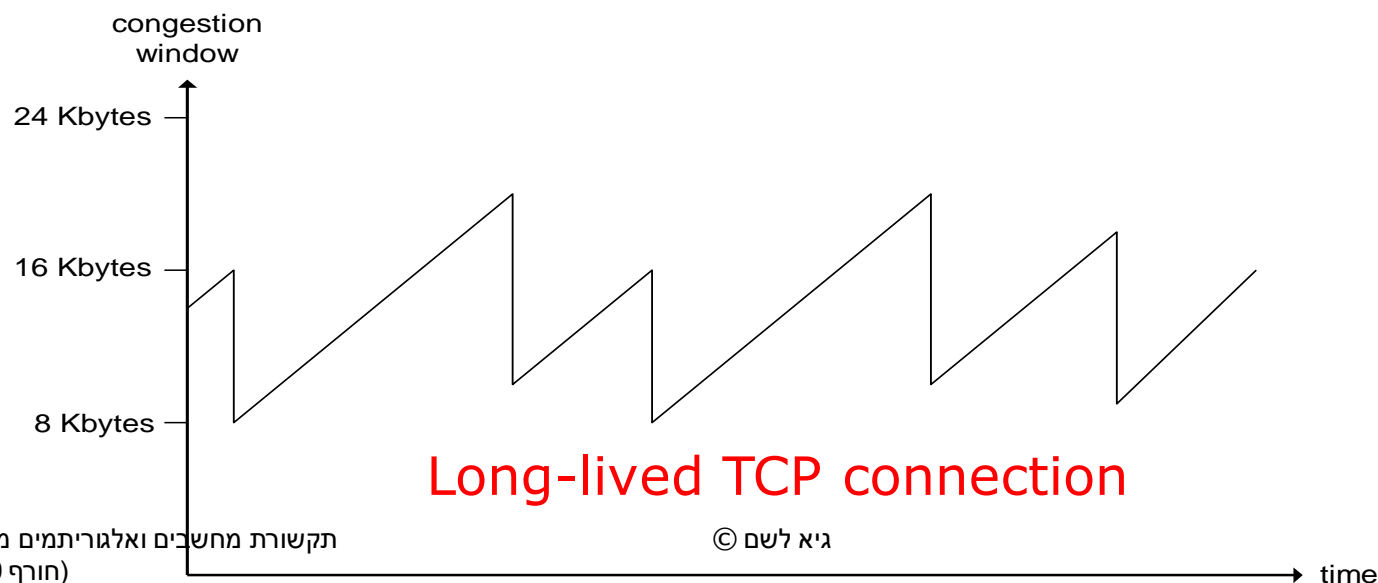
# TCP מניעת עומס

(AIMD: Additive Increase, Multiplicative Decrease)

ה- AIMD מגדיר כיצד הפרוטוקול ישנה את גודל חלון השליחה בתנאים של עומס וחוסר עומס. השיטה מגדירה שכאשר אין עומס, החלון יגדל בצורה ליניארית, וכאשר יש עומס (כלומר, איבוד חבילות) החלון יקטן ע"י חלוקה ב-2. משתמשים בשיטה זו בפרוטוקול ה-TCP במצב של "congestion avoidance" כדי לווסת את גודל החלון, ומכאן את קצב השליחה לפי מצב הרשת. בנוסף, שיטה זו מבטיחה שלאורך זמן, כל המשדרים ייקבלו קצב שידור שווה – כלומר מבטיחה הוגנות.

Additive Increase: כאשר אין איבוד מנות מגדילים את ה-CongWin במקסימום סגמנט אחד בכל RTT (מגבירים על פי מה שנקבע ב-handshake).

Multiplicative Decrease: לעומת זאת, אם יש איבוד מנות אז מקטינים את ה-CongWin בחצי אחרי 3 ACK משוכפלים. כלומר, העליה בקצב היא מתונה מאוד והירידה חדה מאוד



## סיכום בקרת עומס

- כאשר CongWin נמצא מתחת לנקודת הסף השולח המצב slow start – גידול אקספוננציאלי.
- כאשר CongWin מעל נקודת הסף השולח נמצא במצב של המנעות מעומס (Congestion Avoidance) – גידול לינארי.
- כאשר מתקבלים שלושה ACK, נקודת הסף נקבעת ל- CongWin/2 וה- CongWin נקבע לנקודת הסף.
- כאשר יש timeout נקודת הסף נקבעת ל- CongWin/2 ו- CongWin נקבע למינימום.

# אלגוריתם

## Congestion Avoidance

האלגוריתם המשותף ( אלגוריתם זה נקרא גם מסוג Tahoe ):

אתחול קשר קובע את cwnd לגודל ssthresh לגודל סגמנט אחד, ו-65535 bytes.

פונקצית הפלט של TCP אף פעם לא שולחת יותר מהמינימום בין cwnd והחלון שמפורסם ע"י המקבל (Advertised Window). Congestion Avoidance זהו מנגנון בקרת זרימה של השולח. לעומתו פרסום החלון זהו מנגנון בקרת זרימה של המקבל. הראשון מבוסס על הערכת השולח את ה-Congestion ברשת בעוד השני (Advertised Window) קשור לגודל החוצצים של המקבל.

כאשר Congestion קורה (ע"י פקיעת ה-timeout או קבלת מספר ack-ים זהים), חצי מגודל החלון הנוכחי (המינימום בין cwnd והחלון המפורסם ע"י המקבל (Advertised Window), אך לפחות 2 סגמנטים) נשמר ב-ssthresh. בנוסף, אם ה-Congestion נגרם כתוצאה מפקיעת Timeout, cwnd נקבע להיות סגמנט אחד (כלומר Slow Start).

כאשר מתקבל ack על מידע חדש, נגדיל את cwnd. הדרך בה cwnd גדל תלוי באם אנחנו מבצעים Slow Start או Congestion Avoidance.

- Slow Start משמש לאתחל את זרימת המידע בקשר. בנקודה כלשהי נגיע לקיבול המקסימלי של ה-Router, וחבילות "יזרקו". Congestion Avoidance היא הדרך להתמודד עם חבילות שאבדו. השיטה מתוארת ב-Jacobson 1988.
- ההנחה של האלגוריתם היא שאיבוד חבילות הנגרם מנזק הוא מאוד קטן (הרבה פחות מ-1%), לכן איבוד חבילה מאותת על Congestion ברשת בין המקור ליעד. ישנן שתי אינדיקציות לאיבוד חבילה: פקיעת ה-timeout, וקבלת מספר ack-ים זהים (Duplicate Ack).
- Congestion Avoidance ו-Slow start הינם אלגוריתמים בלתי תלויים בעלי מטרות שונות. אבל, כאשר קורה Congestion אנחנו רוצים להאט את קצב שידור החבילות לרשת, ואז לעבור ל-Slow Start בכדי לחדש את קצב השידור.
- Slow Start ו-Congestion Avoidance צריכים שני משתנים לכל קשר:

1. חלון ה-Congestion (Cwnd).

2. Slow Start threshold size (ssthresh).

# אלגוריתם

## Fast Retransmit and Fast Recovery

האלגוריתמים בד"כ ממומשים ביחד בצורה זו אלגוריתם זה נקרא גם **Reno** :

כאשר Duplicate Ack שלישי מתקבל,  $ssthresh$  נקבע למינימום בין חצי ה- $cwnd$  הנוכחי לבין גודל החלון המפורסם ע"י המקבל. שידור חוזר של הסגמנט החסר.  $Cwnd$  נקבע להיות  $3 * ssthresh$  החדש (3 כפול גודל סגמנט).

בכל פעם שעוד Duplicate Ack מתקבל, נגדיל את  $cwnd$  ב-1 ונשדר חבילה באם הערך החדש של  $cwnd$  מאפשר זאת.

כאשר מתקבל Ack שמאשר מידע חדש, נקבע את  $Cwnd$  לערך של  $ssthresh$ . ה-Ack הזה צריך להיות ה-Ack על השידור החוזר מסעיף 1, בזמן RTT לאחר השידור החוזר. בנוסף ה-Ack הזה אמור לאשר את כל הסגמנטים שנשלחו בין שליחת הסגמנט שאבד ועד לקבלת ה-Duplicate Ack השלישי. זהו Congestion Avoidance, מכיוון שאנחנו מאיטים לקצב שידור של חצי ממה שהיה כאשר הסגמנט אבד.

בהמשך לאלגוריתם המעודכן יצא **New Reno** :

New Reno הוא שינוי קטן ב-Reno. הוא שונה מ-Reno בכך שהוא לא מפסיק את ה-Fast Recovery עד אשר כל המידע שהוא שלח לפני שהוא נכנס למצב Fast Recovery, מאושר.

בכך הוא מתגבר על הבעיה ש-Reno מקטין את ה- $Cwnd$  מספר פעמים. שלב ה-Fast Retransmit הוא זהה ל-Reno. ה-Fast Recovery מתחיל כמו Reno, אבל כאשר Ack חדש מגיע אז ישנם שתי אפשרויות:

אם ה-Ack מאשר את כל החבילות ששודרו לפני שנכנסנו למצב Fast Recovery אזי הוא יוצא ממצב Fast Recovery וקובע את ה- $cwnd$  להיות  $ssthresh$  ועובר למצב Congestion Avoidance כמו ב-Tahoe.

אם ה-Ack מאשר חלק מהחבילות ששודרו לפני שנכנסנו למצב Fast Recovery אז אנחנו מניחים שהסגמנט הבא אחרי האשר אבד גם כן ולכן אנחנו מבצעים שידור חוזר שלו. וקובעים את מספר ה-Duplicate Acks (חוזר 2010) שקיבלנו ל-0.

שינויים באלגוריתמי ה-Congestion Avoidance הוצעו ב-1990 ע"י Jacobson. TCP צריך לשדר מיד ack (Duplicate Ack) כאשר מגיע סגמנט לא לפי הסדר.

אסור שה-Duplicate Ack יעוקב.

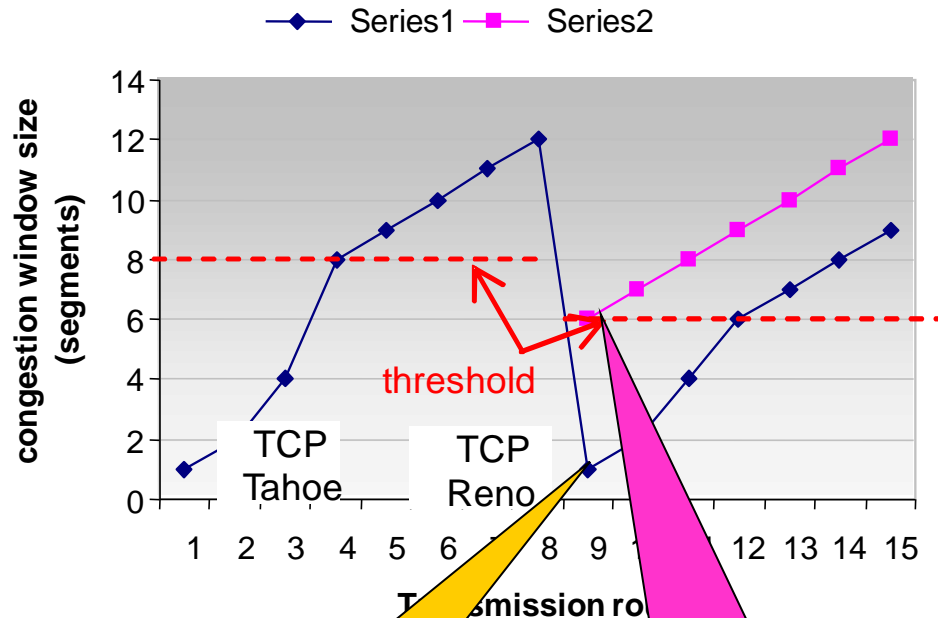
המטרה של ה-Duplicate Ack היא להודיע לצד השולח שהגיע סגמנט לא על-פי הסדר, וגם להודיע לאיזה Sequence number (כלומר איזה סגמנט) הצד מקבל מצפה.

מכיוון שלא יודעים אם ה-Duplicate Ack נגרם כתוצאה מאיבוד חבילה או בגלל הגעת חבילות בסדר לא נכון, מחכים לקבל מספר קטן של Duplicate Acks.

ההנחה היא שאם הבעיה היא רק בהגעת החבילות לא בסדר הנכון אז יהיו רק אחד או שניים Duplicate Acks לפני שתגיע החבילה החסרה. אם שלושה או יותר Duplicate Acks מגיעים ברצף זהו סימן שכנראה החבילה אבדה.

במקרה זה מבצעים שידור חוזר של החבילה שכנראה אבדה, מבלי לחכות לפקיעת ה-timeout. אלגוריתם זה נקרא Fast Retransmit. לאחר מכן Congestion Avoidance מבוצע אך לא Slow Start. זהו אלגוריתם ה-Fast Recovery.

# Multiplicative Decrease Refinement



TCP Series 1  
(Tahoe):  
Restart after  
3 Dup Acks

TCP Series 2  
(Reno) – 3 Dup  
Acks `only`  
halves CongWin

□ תחילי, גירסת של TCP-Tahoe, גם 3 ACKs זהים וגם ל- timeout גורמים להתחלה מחדש.

■ CongWin נקבע ל-1  
MSS.

■ חלון גדל מעריכי.

■ לערך הסף, ואז גדל ליניארי.

□ חדש יותר, גירסת TCP-Reno, מגיב ל-3 ACKs זהים והוא יותר נוח: רק multiplicative decrease