# שכבת התעבורה

## הקדמה

מטרתה העיקרית של שכבת התעבורה הוא לספק את הקישור הלוגי (ולא הפיזי) בין תהליכים שבשני רכיבי קצה. הכוונה בקישור לוגי הוא ששכבת התעבורה מתעסקת אך ורק בשני רכיבי הקצה, כלומר **בשליחה** מרכיב אחד **ובקבלה** ברכיב השני. שכבת התעבורה אינה מתעסקת במסלול הפיזי בדרך הכולל את הקו והנתבים, למסלול זה אחראיים שכבות הרשת והקו. שכבת התעבורה עוטפת את המידע המועבר בשכבת האפליקציה ומעבירה אותו אל הרכיב בקצה השני. בנוסף, שכבת התעבורה מחלקת את המידע למקטעים (segments) מספיק קטנים כדי שיהיה ניתן להעביר אותם בקו הרשת.

פרוטוקולי שכבת התעבורה ממומשים רק ברכיבי הקצה ולא בנתבים שבדרך. יש שני פרוטוקולים אפשריים: UDP ו-TCP. על ההבדלים ביניהם נלמד בהמשך הפרק.

## פורט

כאשר נשלחת הודעה אל שרת, שכבת התעבורה צריכה לדעת מהו כתובת ה-IP של השרת כדי לדעת לאן לשלוח את ההודעה. אמנם לכל שרת ישנם מספר שירותים שהוא מספק, וכל שרות הוא בעצם תהליך שונה המקשיב להודעות. אם כן, כיצד ידע השרת איזה תהליך הוא נדרש להפעיל? לשם כך, לכל תהליך בשרת ישנו מספר הנקרא "פורט" (Port) שאליו הוא מאזין.

שכבת התעבורה מוסיפה לכל מקטע שנשלח שני שדות: פורט יעד ופורט מקור. פורט יעד כדי להפעיל את התהליך הנכון בשרת, ופורט מקור כדי שהשרת ידע איזה תהליך להפעיל אצל הלקוח המאזין לתשובה. ניתן לדמות זאת לשליחת מכתב בדואר, כתובת ה-IP משמשת ככתובת הבניין - מזהה השרת, ואילו הפורט הוא מספר הדירה בתוך הבניין - מזהה התוכנה.

המשימה של יצירת המקטע עם שדה הפורט המסמן את היעד נקראת "multiplexing". והמשימה של שליחת ההודעה אל התהליך המתאים נקראת "demultiplexing".

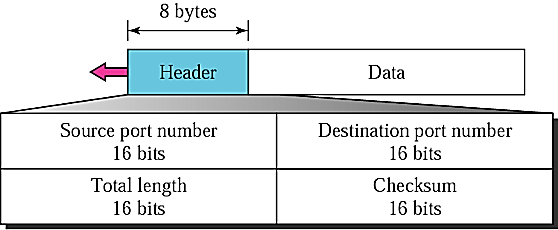
## פרוטוקול UDP

זהו פרוטוקול תעבורה שעושה את הכי מעט שפרוטוקול תעבורה יכול לעשות. פרוטוקול UDP מבצע multiplexing ו-demultiplexing, כלומר שיודע ליצור ולשלוח מקטע אל הפורט המתאים בשרת, ובנוסף יש בו מנגנון קל של בדיקת שגיאות - CRC או Checksum (נלמד עליהם בהמשך). לפרוטוקול זה יש חסרונות לעומת פרוטוקול TCP ולכן במקרים מסוימים לא נרצה להשתמש בו, אמנם למינימליות שלו יש גם יתרונות, ולכן יש פרוטוקולים רבים שמשתמשים בו.

החסרונות בפרוטוקול UDP שהוא פרוטוקול לא אמין, ולכן יתכן מצב של איבוד הודעות, או קבלת הודעות בסדר לא נכון, מפני שכל מקטע נשלח באופן עצמאי מהאחרים. בנוסף, אין יצירת קשר ולכן יתכן שליחה לשרת "מת".

### מתי משתמשים

המקרים שבהם כן נרצה להשתמש בפרוטוקול UDP הם כאשר נצטרך את אחד המאפיינים הבאים:

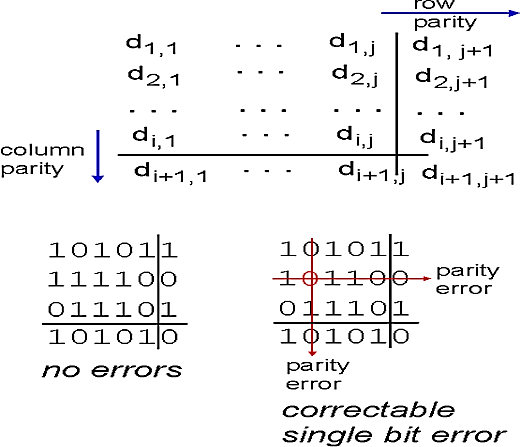
* **אין צורך בחיבור אמין** - כאשר המידע המועבר הוא קצר, ולכן גם אם ההודעות ילכו לאיבוד ניתן לשלוח את המידע שוב. לדוגמא: פרוטוקול DNS משתמש בפרוטוקול UDP בשכבת התעבורה, מפני שגם אם החבילה תלך לאיבוד ניתן לשלוח הודעה נוספת לשרת DNS משני.
* **תקשורת מהירה** - אינו מבזבז זמן על יצירת קשר ועל וידוא שכל המידע הגיע, ולכן קצב שליחת ההודעות מהיר יותר.
* **צריכת משאבים נמוכה** - שומר הרבה פחות מידע ולכן צורך פחות משאבים. גודל של header UDP הוא 8 בתים (Bytes), ואילו של TCP הוא 20 בתים.

### מבנה UDP header

מכיל בסך הכל ארבעה שדות, כל אחד בגודל 2 Bytes (16 ביטים): שדה פורט מקור, פורט יעד, אורך המידע, ושדה לבדיקת שגיאות.

## מנגנוני בדיקת שגיאות

נלמד על שלושה מנגנונים לבדיקת שגיאות שמשתמשים בהם בפרוטוקולי תעבורה שונים. כל המנגנונים מוודאים בהסתברות גבוהה יחסית שהמידע שנשלח הוא אכן המידע שהגיע ליעד. אמנם אינם בטוחים לחלוטין בזיהוי כל השגיאות. מנגנונים אלו עובדים כך שהאימות מתבצע הן על ידי הצד השולח והן על ידי הצד המקבל.



### Parity check

**שולח:** בשיטה זו אנו מסתכלים על המידע כאל מטריצה עם שורות ועמודות. מטרתנו שבכל שורה ועמודה מספר האחדות יהיה זוגי, לשם כך נוסיף סיבית ביקורת. אם השורה/עמודה אי-זוגית אזי נוסיף 1 בשורה/עמודה, ואם היא כבר זוגית נוסיף 0 בשורה/עמודה. בצורה זו נשלח את המידע. אם המידע מכיל רק שורה אחת אזי נוסיף 1 במקרה הצורך רק באותה שורה.

**מקבל:** מוודא שאכן מספר האחדות בכל שורה ועמודה הם זוגיים. אם מוצא שיש רק ביט אחד שגוי (כמו בציור משמאל) מתקן אותו, ואם יש יותר יודיע שיש שגיאה. אם אין שגיאות מוריד את סיביות הביקורת.

### CRC

**שולח:** נסמן את המידע שמעוניין לשלוח ב-D (Data). בנוסף קיים מספר בינארי באורך r+1 ביטים שידוע גם למקבל וגם לשולח, נסמנו ב-G (Generator). מטרתנו למצוא מספר בינארי באורך r, שנסמנו R, כך שאם נוסיף אותו ל-D מצד ימין (שקול לפעולה ), אזי החילוק של <D,R> ב-G (מודולו 2), ייתן 0. נחשב את R באמצעות הנוסחא: *. ונכניס אותו לשדה* CRC*.*

***מקבל:*** *מוסיף את* R *ל-*D *כמו שתיארנו לעיל ומחלק אותו ב-*G. *אם מתקבל תוצאה שונה מ-0 מודיע שקרתה שגיאה בדרך.*

זוהי שיטה חזקה לזיהוי שגיאות ולכן מאד נפוצה ברשתות כיום.

### Checksum

**שולח**: מחלק את ההודעה שהוא רוצה לשלוח לקטעים של 16 ביטים. אם מספר הביטים לא מתחלק ב-16, בקטע האחרון מוסיף אפסים שישלימו ל-16. מבצע פעולת XOR בין כל הקטעים (פעולה זו שקולה לחיבור כל הקטעים וזריקת ה-carry, שהם הביטים שנוספים תוך כדי החיבור, כך שנקבל בסוף מספר בן 16 ביטים). על המספר שהתקבל מבצע פעולת NOT השקול להחלפת כל ביט בהופכי שלו. המספר שהתקבל נכנס לשדה ה-checksum בהודעה.

**מקבל**: גם כן מחלק את המידע לקטעים של 16 ביטים כמו שמתואר לעיל. מבצע פעולת XOR בין כל הקטעים מלבד שדה ה-checksum. לבסוף עם המספר שקיבל מבצע XOR גם עם שדה ה-checksum. המספר שאמור להתקבל לאחר פעולה זו הוא מספר עם 16 ביטים שכולו אחדות. במידה ולא מוחזר ערך זה יודיע שיש שגיאה.

כאמור, מנגנון זה אינו בטוח לחלוטין שכן יכול להיות שתהיה בעיה גם בצד השולח וגם בצד המקבל, ואז ה-checksum לא יבחין בזה. בפרוטוקול UDP כל המידע שנשלח, ואילו בפרוטוקול TCP רק על חלק מהמידע data/header.

## עקרונות העברת מידע אמין בערוץ לא אמין

כאשר שולחים מידע בערוץ אמין אזי בוודאי שהמידע יגיע ליעד הרצוי ללא איבוד חבילות, בסדר הנכון, וללא שגיאות. אמנם כיצד ניתן ליצור העברת מידע אמין בערוץ לא אמין? לשם כך נצטרך להשתמש בפרוטוקול המקיים את העקרונות הבאים:

### מעקב שהחבילות הגיעו לפי הסדר

לכל חבילה יהיה מספר סידורי (sequence number), לפי סדר שליחת החבילות, כך שהצד המקבל יוכל לעקוב שכל החבילות הגיעו כסדרם. אם המקבל יזהה שיש רווח במספור החבילות שקיבל המסקנה תהיה שיש איבוד מידע. המספור ההתחלתי יהיה אקראי ולא יתחיל ב-0, בגלל החשש מערבוב שתי שיחות שנערכות במקביל וחשש מניחוש המספר על ידי גורם שלישי.

### וידוא שחבילה הגיעה

השולח רוצה לוודא עבור כל חבילה שהוא שולח שהיא אכן הגיעה למקבל, והוא לא צריך לשלוח אותה שוב. לשם כך, המקבל עבור כל חבילה שהוא מקבל שולח חזרה הודעת הכרה, הנקראת ack (acknowledgment), עם המספר הסידורי של החבילה שהוא קיבל. כמובן שגם אובדן של הודעת ack אפשרית. אם המקבל מזהה שיש רווח במספרים הסידוריים של החבילות שהוא מקבל, הוא שולח חזרה הודעת אי-הכרה, הנקראת nack (negative ack), עם המספר הסידורי של החבילות שנאבדו. הודעות ack או nack יכולות להיות עבור חבילה אחת או מספר חבילות.

### טיימר

השולח צריך להגדיר זמן קבוע (timeout) שהוא ממתין לקבלת ack עבור חבילה שהוא שלח. אם זמן זה חלף ולא התקבל ack, השולח יניח שהחבילה אבדה והוא ישלח אותה שוב. לשם כך, יהיה טיימר שמתחיל ברגע שליחת החבילה. חשוב להעיר שאי קבלת ack על חבילה לא בהכרח נובע מאובדן חבילה, אלא יכול לנבוע מ-ack שאבד או עומס ברשת שמעכב את קבלת ה-ack עד שהטיימר חלף, אמנם בכל המקרים נניח שהחבילה אבדה ונשלח שוב. לכן יכולים להיות מקרים בהם המקבל יקבל חבילות עם מספר סידורי זהה. בהמשך נלמד כיצד להגדיר זמן אופטימלי.

### חלון שליחה וקבלה

חלון שליחה מייצג את מספר החבילות שניתן לשלוח שעדיין לא התקבל להם ack. חלון קבלה מייצג את מספר החבילות שאפשר לקבל שעדיין לא נשלח להם ack. Pipelining - חלון שליחה גדול מ-1. בפרוטוקולים מתקדמים גודל החלון משתנה תוך כדי שליחת המידע בהתאם למצב הרשת ויכולת השולח והמקבל.

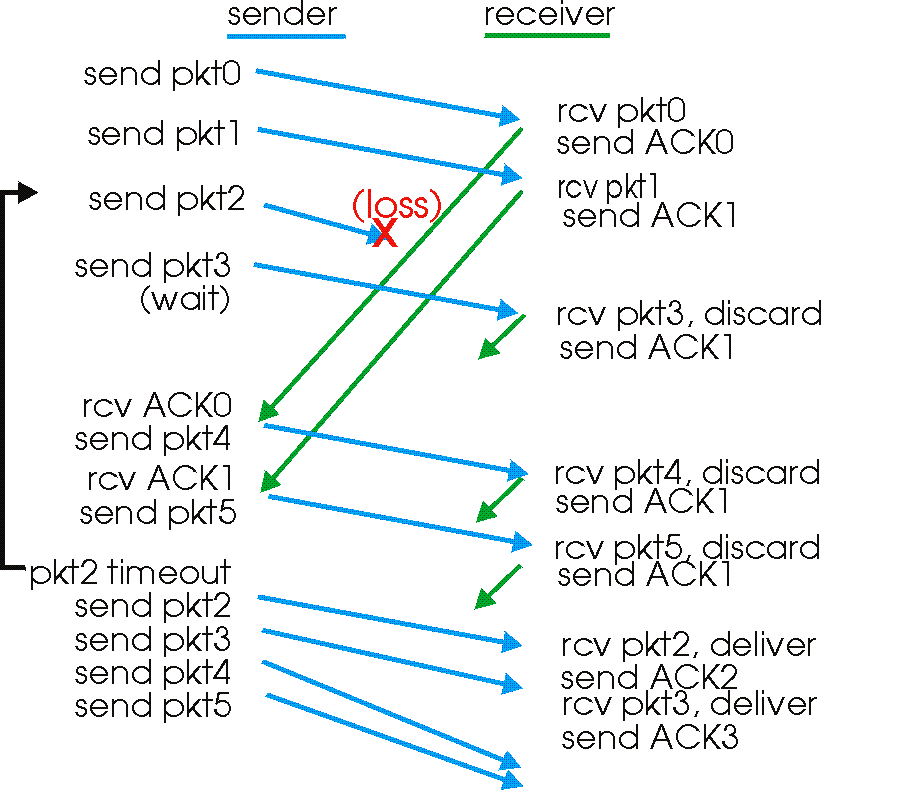
### בדיקת שגיאות

מנגנון שבאמצעותו נבדוק שאכן המידע הנשלח הוא אותו מידע שהתקבל. מנגנון זה הוא צורך בסיסי ואף קיים גם בפרוטוקול UDP שאינו פרוטוקול אמין. סוגי המנגנונים לעיל סעיף ד'.

## פרוטוקולים אמינים

כל הפרוטוקולים בסעיף זה הם פרוטוקולים אמינים להעברת מידע על גבי ערוץ לא אמין. הם מקיימים את כל העקרונות שפירטנו בסעיף קודם.

### Stop & Wait

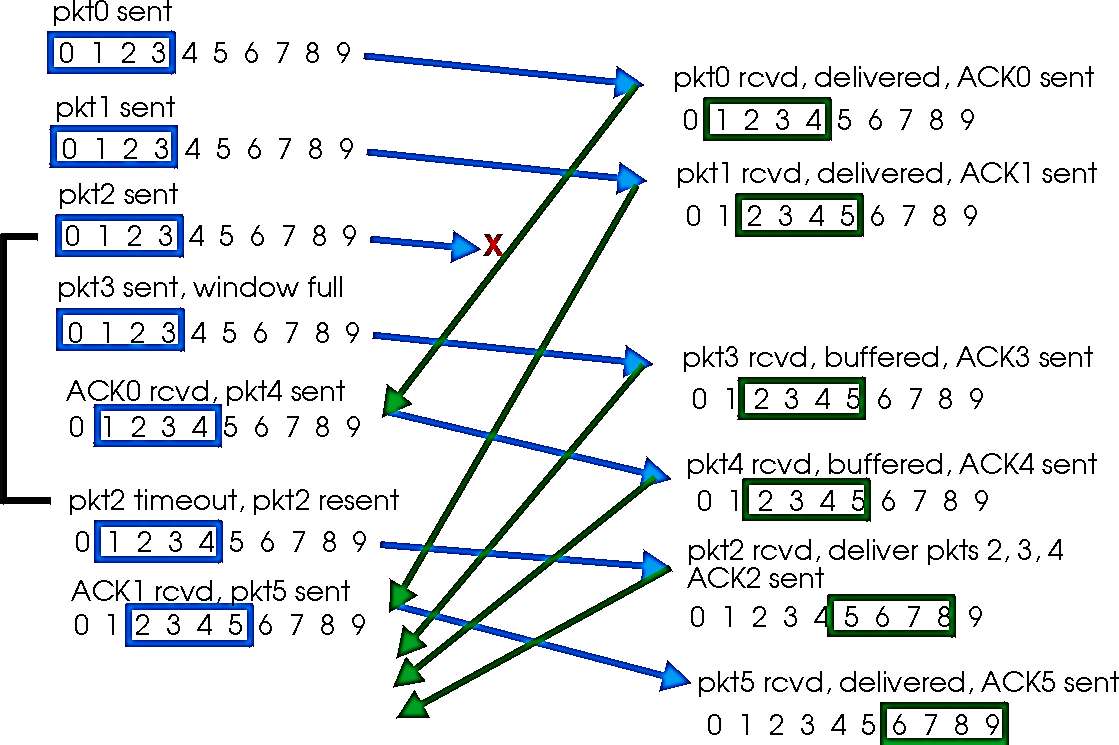
המיוחד בו הוא שהחלון שליחה והחלון קבלה בגודל של 1. כלומר, השולח יכול לשלוח רק חבילה אחת, ואינו שולח חבילה נוספת עד שמקבל על החבילה הקודמת ack. אם הטיימר עבר ולא קיבל ack, שולח את אותה חבילה שוב. מסיבה זו פרוטוקול זה נחשב ללא יעיל.

### Go-Back-N

בפרוטוקול זה גודל חלון השליחה הוא קבוע N וגודל חלון הקבלה הוא 1. השולח והמקבל שומרים את המספר הסידורי של החבילה הבאה שהם אמורים לשלוח/לקבל. המקבל שולח ack חדש רק עבור חבילה שמתאימה למספר זה, אם מגיעה חבילה אחרת סימן שחבילה אבדה, ולכן ישלח ack של החבילה הקודמת התקינה שהגיע.

בצד השולח, חבילה תוסר מהחלון רק כאשר יתקבל עליה ack. כל עוד יש מקום פנוי בחלון שולח חבילה. כאשר מתרחש timeout שולח את **כל החבילות שבחלון**, גם חבילות שכבר שלח. שיטה זו נקראת "ack מצטבר". בנוסף למספר החבילה הבאה שצריך לשלוח, השולח שומר מצביעים לתחילת החלון וסוף החלון. בדוגמא N=4.

### Selective Repeat

בפרוטוקול זה לשולח יש חלון שליחה קבוע בגדול N, ולמקבל חלון שליחה קבוע בגדול M, כאשר MN. בנוסף, יש טיימר נפרד עבור כל חבילה שנשלחת.

המקבל שומר בחלון את כל החבילות שצריכות להגיע. אם מקבל חבילה שנמצאת בחלון, לא משנה מיקומה, שומר אותה בצד ושולח עליה ack. כאשר מקבל את החבילה הראשונה בחלון, מקדם את החלון עד שמגיע לחבילה שאינה שמורה בצד. אם מקבל חבילה שאינה בחלון אך כבר שמר אותה בצד - שולח עליה ack בכל מקרה. ואם לא שמר אותה בצד, כלומר עדיין החלון לא הגיע אליה, מתעלם.

אצל השולח, כל עוד החלון לא מלא הוא שולח הודעה. כאשר מקבל ack על חבילה שומר אותה בצד, לא משנה המיקום שלה בחלון. רק כאשר מקבל ack על החבילה הראשונה בחלון מקדם את החלון עד שמגיע לחבילה שאינה שמורה בצד. כאשר מתרחש timeout שולח רק את החבילה עבורה התרחש ה-timeout.

להבדיל מ-Go-Back-N פרוטוקול זה אינו שולח את כל ההודעות בחלון כאשר נאבדת חבילה, אלא רק את אלו שנאבדו, ולכן יעיל יותר.

## פרוטוקול TCP

TCP (Transmission Control Protocol) הוא פרוטוקול תעבורה מורכב, המקיים את התכונות הבאות:

* מידע עובר בין שולח אחד למקבל אחד.
* Full duplex - ניתן לשלוח מידע בשני הכיוונים, לקוח לשרת ושרת ללקוח.
* מכוון קישור - לפני שליחת מידע מתבצעת "פתיחת קשר" בין הצדדים.
* המידע מגיע במלואו, לפי הסדר, ללא רווחים, וללא כפילויות. באמצעות ack מצטבר ו-seq num.
* ישנו טיימר לכל חבילה. אם הטיימר הסתיים ולא התקבל ack שולחים שוב.
* בדיקת שגיאות - Checksum.
* Flow Control - התאמת חלון השליחה לפי יכולת המקבל.
* Congestion Control - התאמת חלון השליחה לפי מצב ועומס הרשת.

בגלל כל תכונות אלו המקנות שליחה אמינה ומהירה, פרוטוקולים רבים משתמשים ב-TCP כפרוטוקול התעבורה שלהם, ובראשם HTTP.

## TCP Header

ה-header של TCP לבדו בגודל של **20 בתים** לא כולל המידע שרוצים לשלוח. ומכיל את השדות הבאים:

* פורט היעד ופורט המקור.
* Sequence number - המספר הסדרתי של החבילה הנשלחת. נבחר באופן אקראי בתחילת הקשר על ידי השולח והמקבל בנפרד. בחבילה הבאה המספר הסדרתי לא קופץ ב-1 אלא לפי מספר הבתים בחבילה הנוכחית.
* Acknowledgment number - בהודעה מהשולח למקבל מכיל את מספר החבילה עליה מצפים לקבל ack. בהודעה מהמקבל לשולח מכיל את מספר החבילה הבאה אותה מצפה לקבל.
* גודל חלון - הפנוי של המקבל. בעזרת מידע זה השולח יודע כמה הוא יכול עוד לשלוח.
* דגלים - משתנים המקבלים 0 או 1. שישה דגלים סה"כ והם:

ACK - המציין אישור קבלה של פתיחת קשר.

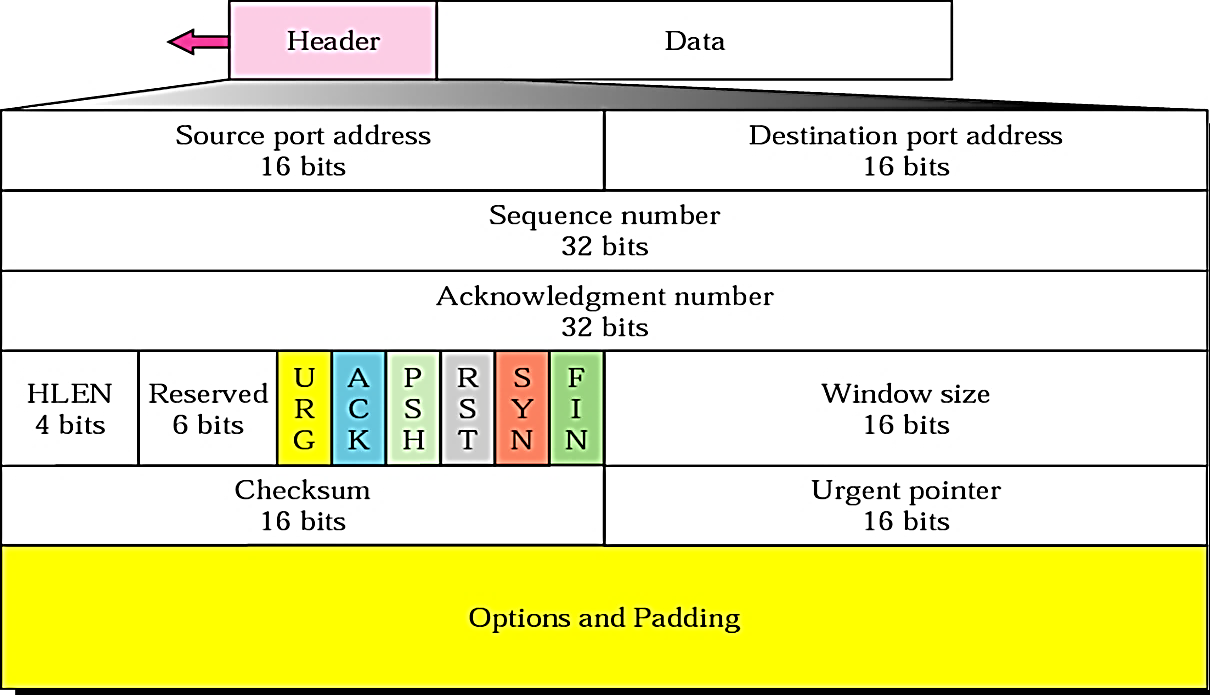
SYN - דלוק רק בחבילה הראשונה ומציין פתיחת קשר.

FIN - המציין סיום קשר, שני הצדדים צריכים לאשר.

RST - גורם למקבל לנתק את החיבור ולהודיע על כך לאפליקציה. קורה לדוגמא כאשר אין שרת שמאזין בפורט המבוקש.

PSH - התראה מהשולח למקבל שאין לו מה לשלוח, ושיעלה כל מה שכבר קיבל לשכבת האפליקציה.

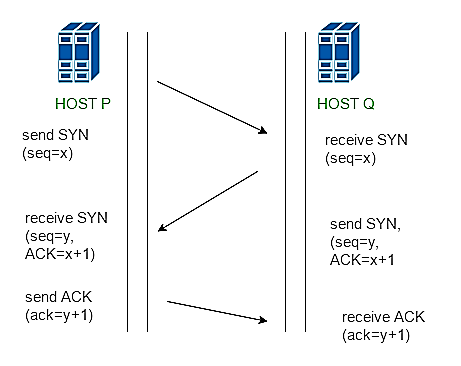
URG -מציין מידע דחוף, לרוב לא בשימוש.

* HLEN - אורך ה-header, שהוא גודל ה-header בבתים.
* Reserved - שמור לשימוש עתידי.
* Checksum - מספר לזיהוי שגיאות.

## פתיחת וסגירת קשר

TCP הוא פרוטוקול מכוון קישור, כלומר לפני שמתחילים לשלוח מידע יש להקים את הקשר באמצעות אישור שתי הצדדים, וכן כאשר מסיימים שליחת המידע סוגרים את הקשר באמצעות אישור שתי הצדדים.

### פתיחת קשר (3-Way-Handshake)

בפתיחת הקשר ב-TCP שלושה שלבים:

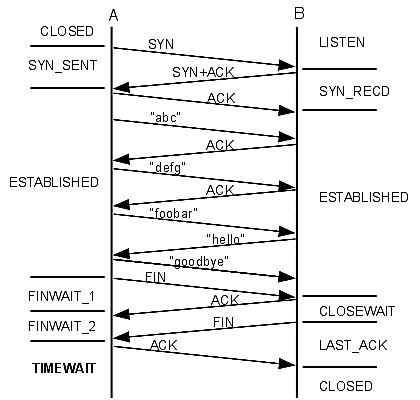
1. **SYN**: הלקוח רוצה לשלוח לשרת הודעה שהוא מעוניין לפתוח קשר. לשם כך הוא פותח socket, בוחר מספר סדרתי x לחבילה הראשונה, ושולח לשרת הודעת TCP SYN שבה דגל ה-SYN דולק.
2. **SYN-ACK**: בצד השרת צריך להיות socket שמאזין לפורט שהלקוח ציין בהודעה. השרת רוצה לעדכן את הלקוח שהוא מסכים לפתוח קשר ולעדכן אותו בחבילה הבאה אותה מצפה לקבל. לשם כך הוא בוחר מספר סדרתי y, ומעדכן Ack num = x+1, כלומר שהחבילה הבאה לה הוא מצפה היא x+1. לבסוף, שולח הודעת TCP SYN-ACK שבה דגלי ה-ACK ו-SYN דולקים.
3. **ACK**: הלקוח, שקיבל את אישור פתיחת הקשר, רוצה לאשר לשרת שקיבל את האישור ולהתחיל לשלוח את המידע. לשם כך הוא מעדכן Ack num = y+1, כלומר שהאישור הבא אותו הוא מצפה לקבל הוא y+1. ושולח הודעת TCP ACK שבה דגל ה-ACK דולק. להודעה זו ניתן גם להתחיל לצרף את המידע שרוצים להעביר.

### סגירת קשר

כדי לסגור את הקשר כל אחד מהצדדים, כאשר הוא מסיים לשלוח את כל המידע שאצלו, שולח הודעת TCP FIN שבה דגל ה-FIN דולק. הצד השני יגיב עם הודעת TCP ACK המאשרת את סגירת הקשר. לאחר ארבעת הודעות אלו (FIN-ACK משני הצדדים), הלקוח מחכה ל-timeout שהוא באורך פעמיים timeout של חבילה רגילה, כדי לוודא שאם יש חבילות שהתעכבו הן לא יאבדו, ואז הקשר נסגר.

לאחר ששולחים FIN לא ניתן לשלוח עוד הודעות בקשר, אלא יש לפתוח אותו מחדש. ולאחר ששולחים ACK על FIN גם לא ניתן לקבל יותר הודעות בקשר.

דרוש תהליך של FIN-ACK משני הצדדים בגלל שיכול להיות שצד אחד סיים לשלוח הודעות אך הצד השני עדיין ממשיך לשלוח. ניתן לקצר את התהליך לשלוש הודעות, בכך שמצרפים גם FIN להודעת ה-ACK הראשונה. בשונה מפתיחת קשר שיוזם אותה הלקוח, סגירת הקשר יכול להתחיל משני הכיוונים, הן הלקוח והן השרת.



### מצבי חיבור TCP - לקוח

CLOSED - אין חיבור.

SYN-SENT - הלקוח שלח הודעה ראשונה לפתיחת קשר.

ESTABLISHED - הקשר מוכן וניתן להעביר מידע.

FIN-WAIT-1 - הלקוח הודיע על סיום, וממתין לאישור השרת.

FIN-WAIT-2 - מחכה שהשרת יודיע על סיום התקשרות.

TIME-WAIT - זמן המתנה לחבילות נותרות.

### מצבי חיבור TCP - שרת

LISTEN - לפני ההתקשרות, השרת ממתין לחיבורים.

SYN-RCVD - אישור חיבור, ממתין ל-ACK מהלקוח.

ESTABLISHED - הקשר מוכן וניתן להעביר מידע.

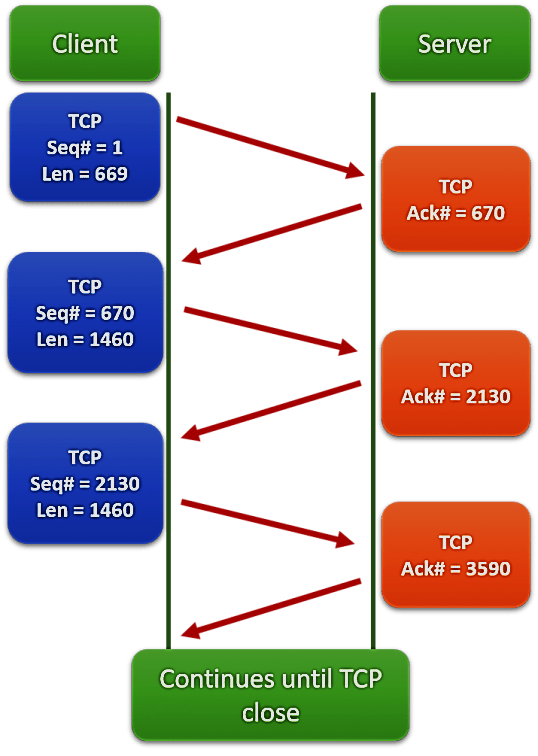
CLOSE-WAIT - הלקוח הודיע על סיום.

LAST-ACK - השרת הודיע על סיום, וממתין לאישור מהלקוח.

CLOSED - אין חיבור.

## זרימת המידע

עבור כל חבילה שהלקוח והשרת שולחים הם צריכים למלא שתי שדות: ack num, seq num. הם ימלאו ערכים אלו לפי העקרונות הבאים:

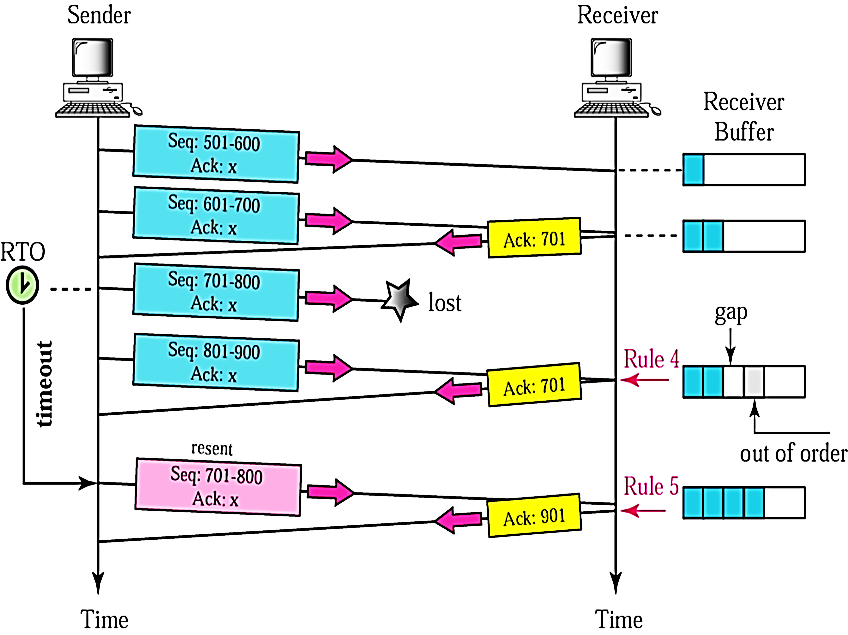
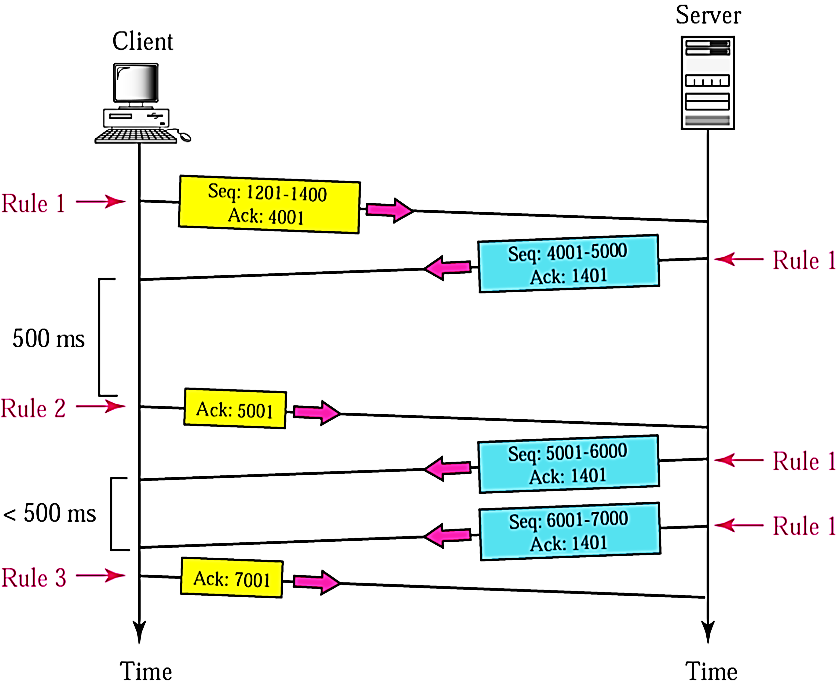
**seq num** - בפרוטוקול TCP גם הלקוח וגם השרת בוחרים בתחילת הקשר כל אחד seq num התחלתי באופן אקראי, בגלל החשש מערבוב שתי שיחות שנערכות במקביל וחשש מניחוש המספר על ידי גורם שלישי. מרגע זה, כל אחד מהם מתחיל לספור את כמות הבתים שהוא שולח. כל חבילה שהם שולחים ה-seq num שלה יהיה מספר הבית הראשון בחבילה.

**ack num** - כאשר כל אחד מהם מקבל חבילה הוא יודע מה המספר של הבית הראשון לפי שדה ה-seq num בחבילה שהוא קיבל, וכן מה מספר הבתים הכולל של החבילה. בחבילה הבאה שהוא ישלח הוא ירצה לאשר לצד השני שהוא קיבל את החבילה. הוא יעשה זאת על ידי כך שב-ack num הוא יכניס את הערך seq num בחבילה שקיבל ועוד אורך החבילה: **ack num = seq num + pkt len**. בכך הוא בעצם אומר קיבלתי את החתיכה האחרונה בחבילה הקודמת שנשלחה, וכעת אני ממתין לקבל את החתיכה שמספרה נמצא ב-ack num.

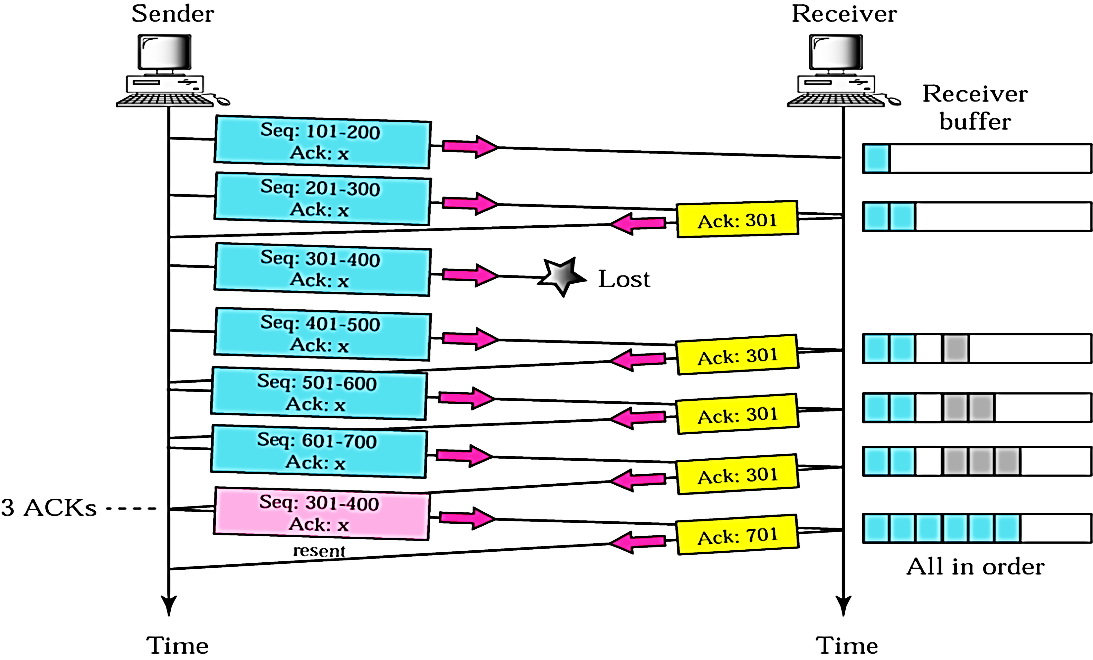
### חוקי שליחת ack

כאשר השרת או הלקוח מקבלים חבילה הם צריכים לשלוח בחזרה הודעת ack (ביחד עם מידע אם קיים), המציינת שהחבילה התקבלה. מנגנון ה-ack ב-TCP עובד לפי עקרונות "ack מצטבר", כלומר כאשר שולחים ack על חבילה זה רק כאשר התקבלו כל החבילות לפניה לפי הסדר. אם התקבלה חבילה שלא לפי הסדר לא ישלח עליה ack והיא תישמר ב-buffer. חוקי מנגנון זה הם:

1. כאשר שולחים הודעת TCP שבה דגל ack דולק, מצרפים בשדה ack num את מספר החבילה אותה מצפים לקבל.
2. במקרה שבו אין יותר מידע לשלוח אך ממשיכים להמתין לחבילות שצריכות להגיע, אזי במקרה זה לא נרצה שעבור כל הודעה שתתקבל ישלח ack, וזאת כדי לצמצם את התנועה ברשת. לכן עבור כל הודעה שהתקבלה לפי הסדר, נמתין זמן קצר (חצי שנייה לדוגמא) או עד שתגיע חבילה אחרת לפי הסדר, ואז ישלח ack על שניהם.
3. אם הגיעה חבילה חדשה שגם עבורה וגם עבור החבילה הקודמת שהגיע לא נשלח ack, ושתיהן לפי הסדר, ישלח ack על שניהם.
4. אם הגיע חבילה עם seq num שלא ציפינו לו, תשלח הודעה של ה-ack num הקודם שאותו מצפים לקבל, והחבילה שלא לפי הסדר תישמר ב-buffer. חוק זה יוביל ל-Fast Retransmission שנלמד בהמשך.
5. אם הגיע חבילה שהייתה חסרה וכבר התקבלו מספר חבילות שהם אחריה לפי הסדר, אזי כשתתקבל חבילה זו נשלח ack עליה ועל כל החבילות אחריה שכבר קיבלנו.
6. אם הגיע חבילה שכבר שלחנו עליה ack, נשלח עליה שוב ack בכל מקרה.



### Fast Retransmission

כאמור לעיל בחוק 4 של מנגנון ack, כאשר מקבלים חבילה לא לפי הסדר שולחים ack num של החבילה הקודמת לה מצפים. Fast Retransmission אומר שכאשר השולח מקבל את אותו ack שלושה פעמים, הוא לא מחכה ל-timeout עבור אותה חבילה אלא כבר שולח אותה שוב, ובכך חוסכים זמן. ה-ack הבא שיקבל יהיה גם על כל מה שכבר שלח ולא התקבל עליו ack, לפי חוק 5.

## Timeout

בפרוטוקול TCP לכל חבילה ישנו טיימר שמתחיל ברגע שליחת החבילה. אם הטיימר הסתיים ולא התקבל ack על חבילה זו היא תשלח שוב. כיצד נגדיר את הזמן בטיימר? אם יהיה קצר מדי יש חשש שיהיו שליחות חוזרות מיותרות, ואם יהיה ארוך מדי אזי במקרה של איבוד חבילה ייקח זמן עד שתישלח שוב. בנוסף, נצטרך להתחשב במצב של הרשת שכל הזמן משתנה, לכן גם הטיימר צריך להיות אדפטיבי.

פתרון לבעיה זו הוא להגדיר את הטיימר באמצעות שני פרמטרים:

**EstimatedRTT** - נשערך כמה זמן אמור לקחת מרגע שליחת החבילה ועד שמתקבל עליה ack. נעשה זאת על ידי שערוך הזמן של החבילה הקודמת והזמן בפועל שלקח לחבילה הקודמת (SampleRTT), כאשר ניתן חשיבות גדולה יותר לשערוך החבילה הקודמת לדוגמא: . כך אנחנו מתחשבים במצב ברשת.

**DevRTT** - מרווח בטיחות שנחשב שנוסיף לפרמטר הקודם. נחשב את המרווח בטיחות באמצעות המרווח בטיחות שחישבנו לחבילה הקודמת וההפרש בין הזמן בפועל של החבילה הקודמת עם השערוך שלה, כאשר ניתן חשיבות גדולה יותר למרווח בטיחות של החבילה הקודמת לדוגמא: .

לסיכום נחשב את ה-timeout של חבילה t כך:

## בקרת זרימה - Flow Control

לכל אחד מהצדדים ישנו buffer שבו הוא מאחסן את החבילות שקיבל, וממנו שכבת האפליקציה קוראת את המידע. יתכן מצב בו המידע מתקבל בקצב מהיר יותר מאשר האפליקציה מספיקה לקרוא, כך שה-buffer מתמלא ואז לא ניתן לקבל יותר חבילות. מנגנון Flow Control מתמודד עם בעיה זו. העקרון הוא שעבור כל חבילה שמקבלים מודיעים לשולח בשדה Window מהו גודל ה-buffer הפנוי. לפי גודל זה השולח מצמצם או מרחיב את חלון השליחה שלו. כתוצאה מכך, כאשר גודל ה-buffer קטן, השולח יאט את קצב השליחה כדי לא להציף את ה-buffer.

כאשר גודל ה-buffer קטן מאוד והחבילה הבאה יותר גדולה מגודל ה-buffer, אזי או שהשולח יחלק את החבילה למספר חבילות קטנות יותר, או שהמקבל יגדיל את ה-buffer בהסכמת שני הצדדים.

## בקרת עומס - Congestion Control

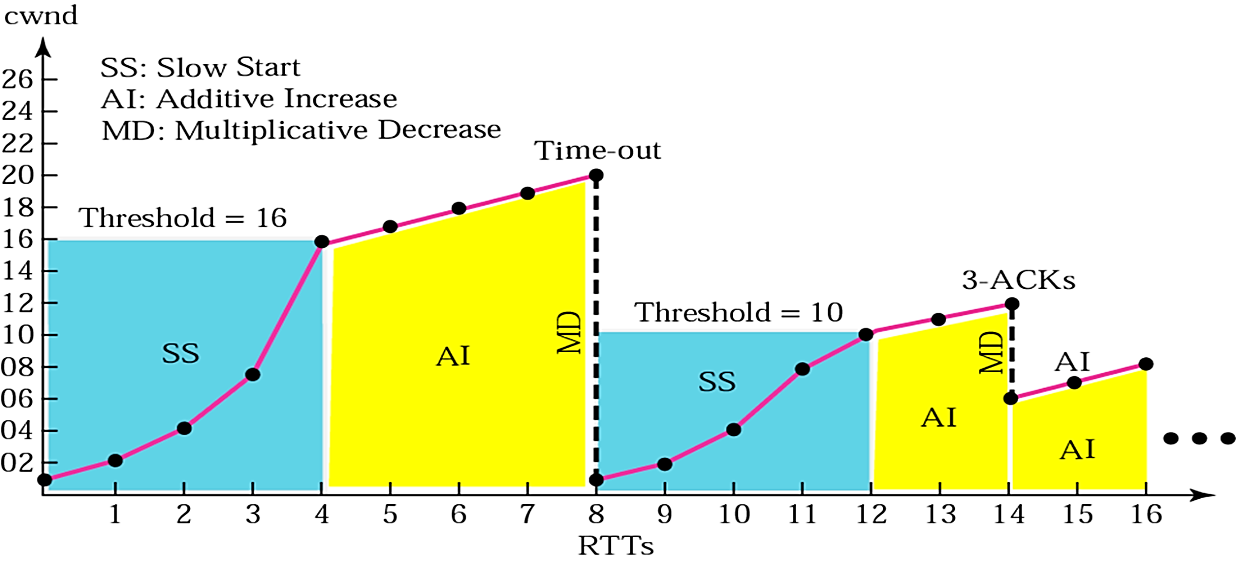
כאשר יש עומס ברשת יהיו הודעות שילכו לאיבוד או יתעכבו, וכתוצאה מכך הודעות אלו ישלחו שוב ואז יצרו עוד עומס וכו'. מנגנון Congestion Control בא לפתור בעיה זו. העיקרון הוא שכאשר מזהים שיש עומס ברשת מקטינים את חלון השליחה ובכך מאיטים את קצב שליחת ההודעות שיתאימו למצב הרשת. ניתן לזהות עומס ברשת בשני דרכים:

* **End-end congestion control** - הרשת אינה מספקת מידע על מצב העומס שלה, אלא מזהים עומס באמצעות אבדן חבילות (timeout) או ריבוי acks על אותה חבילה (Fast Transmission).
* **Network-assisted congestion control** – בדרך זו הנתבים עצמם מדווחים על עומס ברשת, או על ידי כך שמחזירים הודעה לשולח Choke Packet, או שבכל חבילה שעוברת ברשת הם מדליקים דגל שמסמל על עומס. במקרה השני הצד המקבל את החבילה רואה את הדגל ומודיע לשולח על העומס.

### ניהול חלון שליחה

יש שלושה מימושים שבאמצעותם פרוטוקול TCP מנהל את גודל החלון כאשר הוא מזהה עומס:

1. **AIMD** - בתחילת הקשר גודל החלון שליחה הוא חבילה אחת. עבור כל חלון שמתקבל בהצלחה מגדילים את החלון באחד, כך שכל עוד אין עומס החלון שליחה גדל ליניארית. כאשר מזהים עומס מקטינים את חלון השליחה לחצי מגודלו.
2. **Slow Start** - בתחילת הקשר גודל החלון שליחה הוא חבילה אחת. עבור כל חלון שהתקבל בהצלחה מכפילים את גודל חלון השליחה פי 2, כך שכל עוד אין עומס החלון שליחה גדל אקספוננציאלית. כאשר מזהים עומס מקטינים את חלון השליחה לחצי מגודלו.
3. **Congestion Avoidance** - קובעים איזשהו משתנה גבול (Threshold) שנסמנו X, ונפעל באופן שהוא כעין שילוב שתי המימושים הקודמים. עבור כל חלון שהתקבל בהצלחה, כל עוד גודל החלון שליחה קטן מ-X החלון יגדל אקספוננציאלית, אך אם גדול מ-X החלון שליחה יגדל ליניארית. כאשר מקבלים 3 פעמים ack על אותה החבילה (Fast Retransmission), מקטינים את X ואת חלון השליחה לחצי מגודל חלון השליחה הנוכחי. וכאשר מתרחש timeout X יקבע כחצי **מגודל החלון הנוכחי**, והחלון עצמו ירד ל-1. הסיבה להבדל היא ש-3 ack מראה שהרשת עדיין מסוגלת להעביר חבילות ולכן החלון שליחה יורד רק בחצי, אולם ב-timeout יש יותר חשש שהרשת עמוסה ולכן החלון צונח ל-1. דוגמא לשיטה השלישית בתמונה.



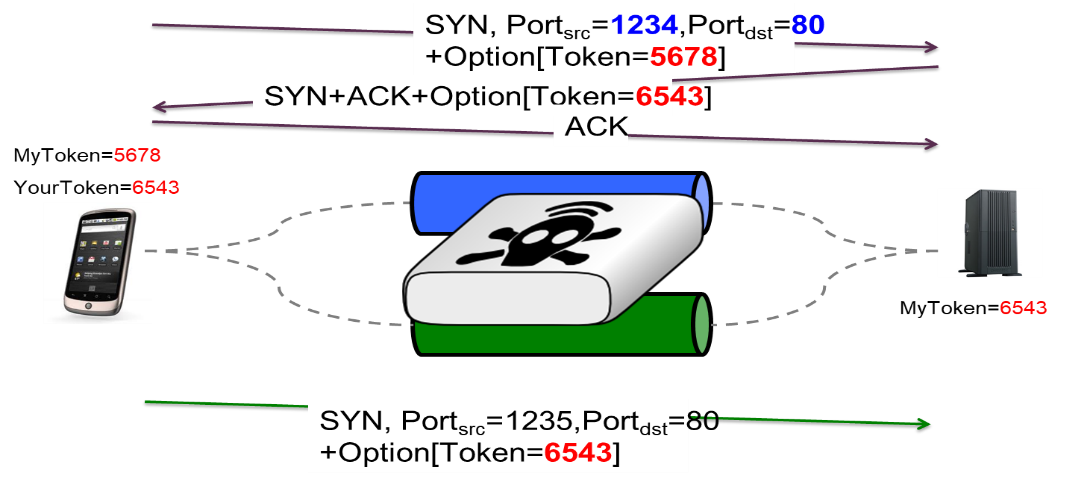
## קביעת גודל חלון שליחה

גודל חלון השליחה נקבע הן על ידי מנגנון Flow Control והן על ידי מנגנון Congestion Control. נסמן את גודל חלון שליחה שמוחזר מ-Flow Control - FW (Flow Window), ואת חלון השליחה שמוחזר מ-Congestion Control - CW (Congestion Window). גודל החלון שליחה בפועל יקבע על ידי המינימום מבין שניהם.

## TCP Fairness

TCP הוא פרוטוקול הוגן המחלק את רוחב הפס בצורה שווה בין המשתמשים בקו. חלוקה שווה זו מושגת באמצעות מנגנון Congestion Control. אפליקציה יכולה להגדיל את רוחב הפס שהיא מקבלת על ידי פתיחת מספר קישורי TCP במקביל, ואכן דפדפני אינטרנט רבים עושים זאת.

## TCP Multipath

מכשיר המחובר לשתי רשתות בו-זמנית, כמו לדוגמא פלאפון שיכול להיות מחובר לרשת סלולרית ו-WiFi במקביל, יכול להשתמש בשני רוחבי הפס ובכך לקצר את זמן ההורדה/שליחה. לשם כך צריך להודיע לשרת ששני התהליכים הפונים אליו הם מאותה אפליקציה כדי שידע לפצל את המידע בין שניהם. עושים זאת באמצעות שדה ה-Options בפרוטוקול TCP. בתהליך אחד נצרף Token להודעת TCP של הפתיחת קשר. השרת ישמור את ה-Token שלנו וישלח Token משלו בחזרה, גם כן ב-Option של TCP. לאחר מכן, בתהליך השני נשלח את ה-Token שקיבלנו מהשרת. כעת השרת יודע ששני התהליכים מקושרים לאותו מכשיר ויפצל ביניהם את המידע.