תקשורת ומחשוב

מחברת סיכומים מההרצאות של ד"ר דביר עמית זאב

תקשורת ומחשוב

נערך ונכתב על-ידי דור עזריה

2021

ספר זה לא נבדק על ידי מרצה, יתכן שימצאו טעויות.

לסיכומים נוספים שלי במדעי המחשב ומתמטיקה:
https://dorazaria.github.io/

מוזמנים לעקוב אחרי 😊:
LinkedIn: https://www.linkedin.com/in/dor-azaria/
GitHub: https://github.com/DorAzaria

תוכן עניינים

הקדמה

ה**אינטרנט** (בעברית: **מִרְשָׁתֶת**) היא רשת תקשורת נתונים בעלת היקף כלל עולמי. הרשת נוצרה כתוצאה מחיבורים רבים בין רשתות מחשבים - חיבורים אשר איפשרו תקשורת בין מחשבים רבים ברשתות רבות. היקף הרשת, כמות המידע העצומה האגורה בה, והפעילות הרבה שמתרחשת הודות לה - הפכו את האינטרנט לגורם רב משמעות ולזירת ההתפתחות הכלכלית והתרבותית.
ישנם מליארדי מכשירים המחוברים לרשת:

מארח (host) - הוא ציוד אשר שולח ומקבל הודעות ישירות דרך הרשת – לציוד זה יש כתובות MAC וכתובת IP. למשל מחשבים, מדפסות, מצלמת רשת, סורק, שרתים וכו'...הם הקצוות או "המסגרת" באויר והם מה שאנו מכירים ביום יום.
כתובת IP - היא מספר המשמש לזיהוי נקודות קצה, כגון מחשב, ברשתות תקשורת שבהן משתמשים בפרוטוקול התקשורת IP, כגון רשת האינטרנט.

אם נכנס פנימה נראה את ה- **Packet switches** שהם נתבים (routers) ומתגים (switches).
חבילה (Packet) - היא אוסף נתונים שאליו מצורפים כתובת המקור, כתובת היעד ונתוני בקרה, כך שהרשת תוכל לנתב את החבילה ליעד הנכון.
נתבים (Routers) - נתבים משמשים לניתוב נתונים מרשת אחת לרשת אחרת. לדוגמה, מרשת האינטרנט, לרשת הפרטית של המשתמש הביתי.
מתגים (Switches) - הוא רכיב ברשת מחשבים המחבר בין צמתים שונים ברשת, בין אם הם מכשירי קצה (כגון מחשבים) ובין אם הם מרכיבי רשת בסיסיים.

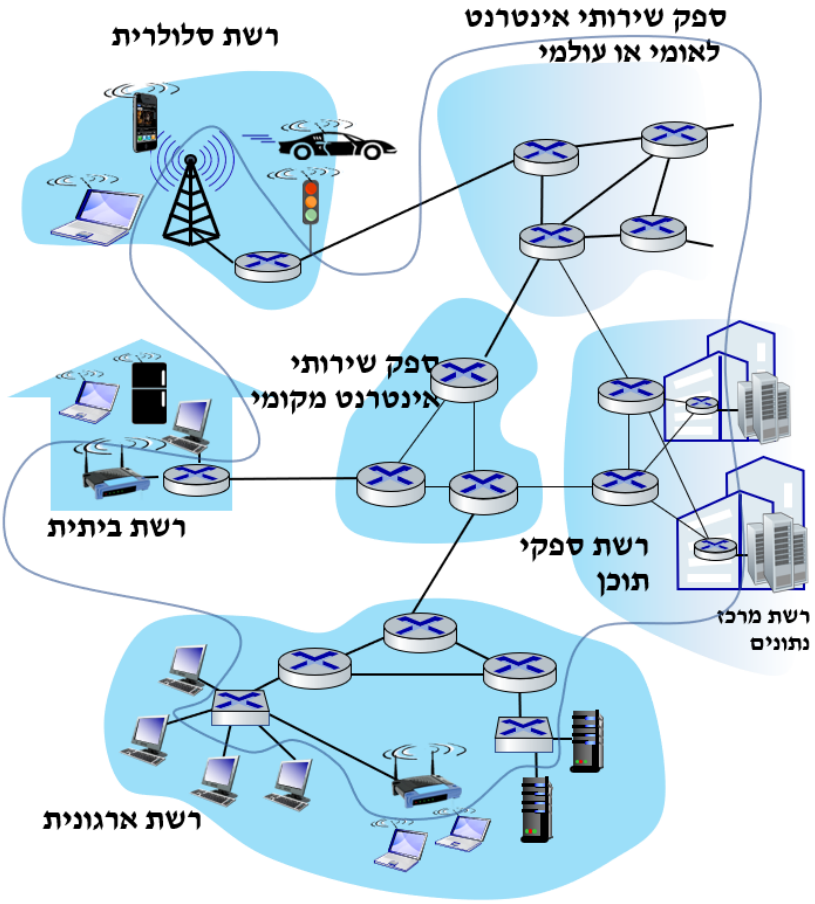
אבל איך כל העסק הזה מתחבר ביחד? -
בעזרת **קווי תקשורת (Communication links),** חוטיים ואל-חוטיים.
כמו למשל fiber, copper , או wifi, לווינים, סלולר ורדיו. כמובן לכל אחד מהם יש קצב שידור (Transmission rate) שונה אשר נמדד במידת רוחב פס (**Bandwidth**) כלומר כמות הנתונים שניתן להעביר בקו תקשורת בזמן נתון, רוחב פס נמדד בצורת - (bps - bits per second).

נוצרים רשתות המיועדות לגופים שונים (**Networks**), כמו למשל בזק בינלאומי - היא רשת עם הלקוחות שלה, האוניברסיטה היא רשת.

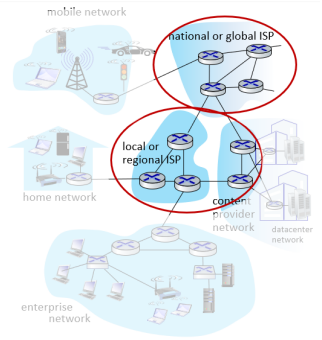
האינטרנט הוא בעצם "רשת של רשתות" כלומר ספקי השירות האינטרנט (**ISP**s) מחוברים זה לזה.

ספק האינטרנט (ISP) - הוא הגוף המאפשר למשתמש להתחבר לפרוטוקול TCP/IP המקשר בינו לבין רשת האינטרנט העולמית. כמו למשל בזק בינלאומי, פרטנר ו-HOT.
ניתן להסתכל על האינטרנט בצורה של "תשתית של אפליקציות".

הצורה שבה כל הספקים יודעים "לדבר" אחד עם השני וכל הגופים ברשת מתקשרים הוא ה**פרוטוקול תקשורת (Protocols)** - תפקידו ליצור שפה משותפת בין גופי רשתות, הוא שולט בקבלה ושליחה של הודעות. כמו למשל HTTP (Web), streaming video, Skype, TCP, IP, WiFi, 4G, Ethernet.



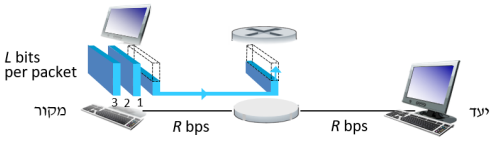
ליבת הרשת



בליבת הרשת יש רשת של נתבים המחוברים זה לזה.
המשתמש מחלק את המידע לחלקים (חבילות) שעוברות לנתב והנתב מעביר את החבילה בכמות רוחב הפס הכי גדולה שהוא יכול לאפשר לאיפה שהחבילה אמורה להגיע.

אחריות הנתב בקבלת ה-**Packet-Switching** לאחר קבלתו מהמשתמש:
תהליך זמן העברה עבור כל חבילה מתבצע כך:

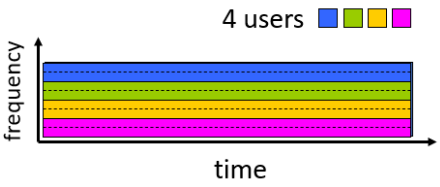
L = 10 Kbits , R = 100 Mbps , one-hop transmission delay = 0.1 msec



1. Transmission delay - כמה זמן לוקח לנו לשדר לתוך הקו. לוקח L/R
2. Store and forward - פעולת הנתב או המתג, הוא מחכה שכל החבילה תגיע אליו ורק אז הוא יקבל החלטה לאן להעביר את המידע.
3. End-end delay - הגיע אל היעד, לפחות 2L/R.

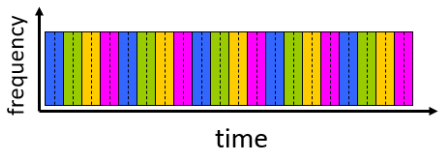
- תור ואובדן של חבילות: אם קצב ההגעה (בקצב bps) לקישור עולה על קצב ההעברה (bps) של הקישור,החבילות יעמדו בתור ויחכו להעברתן בקישור היציאה, חבילות יכולות להאבד אם הזיכרון (buffer) בנתב (router) מתמלא עד אפס מקום. (כלומר אם התור מלא).
- יתרון ה-**Packet-Switching** הוא קבלת שירות לפי דרישת הצרכן והחיסרון שהוא לא מאפשר לי לשמור משאבים (אין הבטחה לרוחב פס קבוע ולהשהייה מסוימת - כי אני לא יודע כמה משאבים יכנסו לי).

חלופה ל-Packet-Switching ופחות פופולרית היא **Circuit-Switching:**
הוא יכול לשמור משאבים, כלומר שומר הקצאות עבור משאב אחד ויחיד.



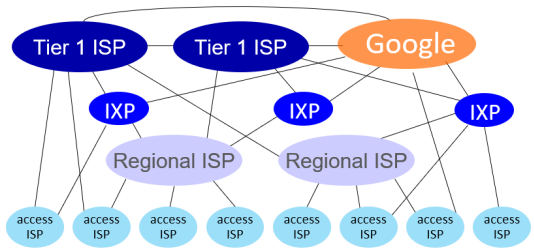
יש 2 שיטות שפועלות באמצעות **Circuit-Switching:**

FDM - Frequency Division Multiplexing:
חלק מרוחב הפס מוקצה להעברת מידע.
כמו שכל תחנת רדיו משדרת בתדר שלה, היא לא מפריעה לאף אחד, היתרון שכל אחד יכול לשדר מתי שהוא רוצה ואיך שהוא רוצה.

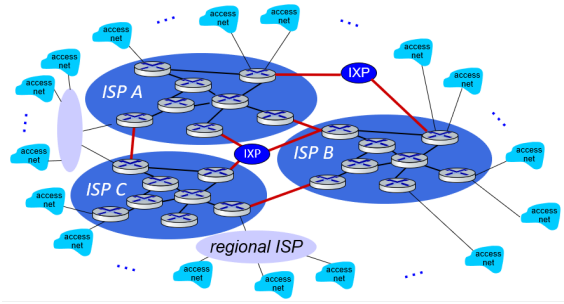


TDM - Time Division Multiplexing:
רוחב הפס מתחלק במספר המשתמשים כך שבכל פעימה כל רוחב הפס מוקצה עבור משתמש אחד.

מבנה האינטרנט



אנחנו המארחים, מתחברים לאינטרנט באמצעות ספקי שירותי אינטרנט (ISP).
ה- ISPs חייבים להיות מחוברים זה לזה כך שבין שתי מארחים יהיה את האפשרות לשלוח חבילות ביניהם - אפשר להבין מכך שמבנה האינטרנט מורכב מאוד.
היינו רוצים שיהיה את החיבור בין כולם, אך ישנם בעיות פוליטיות וכלכליות וכו' לחיבורים אלה.



עם הצלחת האינטרנט קמו מלא ISPs וקמה השאלה איך נחבר בין ספקי השירות האלה כדי לאפשר העברת חבילות בין מארח מ-ISP A למארח מ- ISP B ולכן נוצר ה- - **IXP** - Internet Exchange Point שתפקידו לחבר בין הספקים.

עם הזמן קמו ISP אזוריים שתפקידם לחבר כמה access net מאותו איזור אל ה-ISPs הגדולים.
וגם קמו עוד ISP גדולים כמו גוגל שראו יתרון רב בניהול ISP משלהם לטובת שירות גלישה טובה יותר ומהירה יותר וכמובן גם רווחית יותר.

השהיות

הלקוח משדר חבילה ⏪
⏪**השהיית זמן שידור** - משך הזמן הדרוש לדחיקת כל הביטים של החבילה לחוט.⏪
⏪ **השהיית התפשטות** - מנהל כמה זמן לוקח לנו לשדר מיציאה החוט עד לנתב שלנו, המעבר מתבצע כך שכל ביט בודד מהחבילה עובר עד שכולם מגיעים לנתב.⏪
⏪**השהיית עיבוד** - כשהחבילה מגיעה כמו שצריך מהלקוח או מנתב אחר אל נתב (router) כלשהו, תפקידו לוודא שהחבילה הגיע כמו שצריך ואין שום תקלה במעבר מהלקוח עד אליו ואז הוא צריך להחליט לאן להעביר את החבילה. הוא בודק שגיאות ביטים, מחשב יציאה לקישור ובדר"כ דורש זמן גדול מהרגיל. ⏪
⏪ **השהיית המתנה בתור** - אחרי שהנתב החליט לאן להעביר את החבילה, הוא מעביר את החבילה ללינק המתאים ובמצב הזה יכול להיות תור,זמן ההמתנה משתנה בהתאם לעומסים.
אם החבילה הגיעה לתור מלא היא נזרקת משם ויש מקרים שהיא נשלחת חזרה ללקוח וכך התהליך חוזר על עצמו⏪ או שהיא לא חוזרת ללקוח בכלל ופשוט נאבדת.

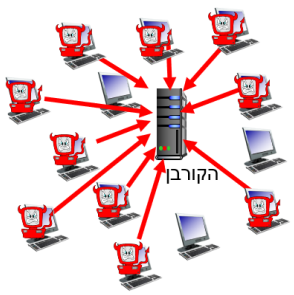
כדי לראות בעיניים את המסלול הזה, יש תוכנה שנקראת **traceroute** שניתן להפעילה בcmd,
הוא מודד את העיקוב מהמקור אל הנתב לאורך האינטרנט עד ליעד.

אבטחת הרשת

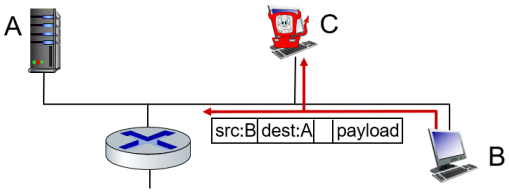
כשבנו את האינטרנט לא באמת לקחו בחשבון את עניין האבטחה.
יש כל מיני סוגים של malware (תוכנות שנוצרו בכוונת תקיפה למחשבים), אם זה וירוס שצריך להפעיל אותו או אם זה בוטים שמשתלטים על המחשבים שלנו ללא ידיעתנו ומנצלים אותם לצורך תקיפה משותפת עבור גוף תקשורת מסויים.

Denial of Service - DoS -התקפה שנועדה לכבות מחשב או רשת, מה שהופך אותה לנגישה למשתמשים המיועדים לה.

- Distributed Denial of Service - DDoS
כמות גדולה של נתונים מציפים את הקורבן ממקורות רבים ושונים
(אנחנו המקורות השונים ואנחנו לא מודעים לזה).
מצד המגן - קשה מאוד לעצור את ההתקפה כי היא מגיעה מכמות גדולה של מקורות.



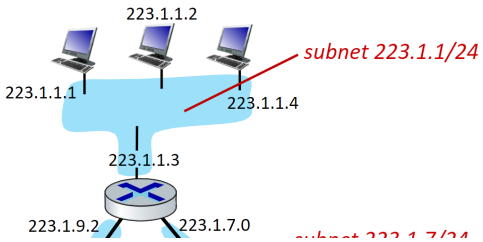
Packet Sniffing - כמו למשל שימוש ב wireshark שאפשר להקליט את המידע של התעבורה, עם המידע הזה אנו יכולים לשכפל אותה ולהגיד שאנחנו בכלל מחשב אחר B למשל, אני רוצה להציק לשרת ואני לא רוצה שיעלו עלי אז אני אגרום להם לחשוב שהמקור הוא B ולא אני C - זה נקרא **IP Spoofing**.



כתובת IP

כתובת פרוטוקול אינטרנט (כתובת IP) היא תווית מספרית המוקצית לכל מכשיר המחובר לרשת מחשבים שמשתמשת בפרוטוקול האינטרנט לתקשורת. כתובת IP משרתת שתי פונקציות עיקריות: זיהוי ממשק מארח או רשת וכתובת המיקום שלה.תפקידו לחבר בין מארח / נתב לכבל פיזי. לרוב הנתבים יש כמה חיבורים כאלה. למארח בדרך כלל יש אחד או 2 חיבורים כמו אינטרנט חוטי ואינטרנט אלחוטי וכו'...

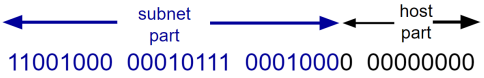
Subnets - חלוקה הגיונית של רשת IP הנוהג לחלק רשת לשתי רשתות או יותר.
זה תת-רשת שלא עוברת דרך ראוטר, כלומר בין מחשבים באותו התת-הרשת אין להם צורך להשתמש בראוטר כדי לתקשר אחד עם השני. כלומר אם אני מחובר לאינטרנט הביתי עם המחשב ואני שולח הודעה לאדם נוסף בבית שגם הוא מחובר באותו האינטרנט (אותו ראוטר) אז אנחנו נתקשר תחת אותו subnet.



אם נסתכל בציור מימין נראה כי הסביבה הכחולה זה תת-הרשת שלנו שיוצא מהראוטר ומחוברים אליו 3 מחשבים, במקרה זה כל מחשב יחזיק את את אותה כתובת בעלת 223.1.1 אבל המיקום האחרון בכתובת שייכת למחשב ספציפי כלומר ייחודי אליו, כלומר 233.1.1/24 אז אפשר 24 כתובות שונות בתת הרשת הזאת (24 מחשבים).

subnet-mask - מסכה שנותנת את הכתובת של הרשת בה נמצא המחשב.

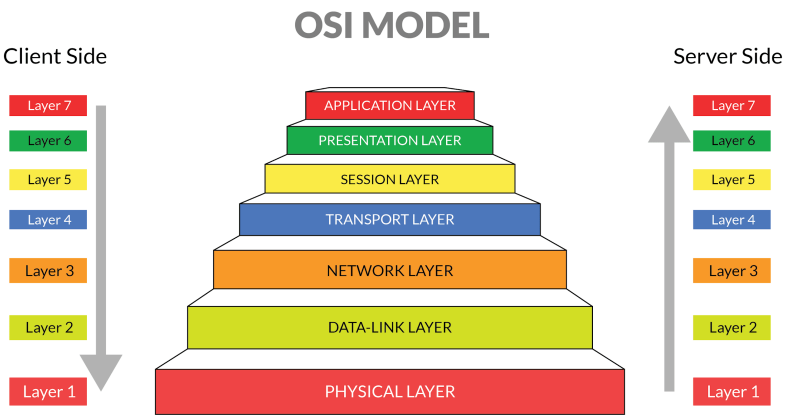
network-classes - ארכיטקטורת כתובות רשת מחלקת את שטח הכתובות עבור פרוטוקול האינטרנט גרסה 4 (IPv4) לחמש מחלקות כתובת. כל מחלקה, המקודדת בארבעת הביטים הראשונים של הכתובת, מגדירה גודל רשת שונה, כלומר מספר מארחים עבור כתובות (מחלקות A, B, C), או רשת ריבוי שידורים (מחלקה D). טווח הכתובות החמישי (E) שמור למטרות עתידיות או ניסיוניות. ידיעת שיעורי רשת הופכת לבעיה כשאתה מתמודד עם ניתוב.



- **CIDR** -Classless InterDomain Routing
- **DHCP**- Dynamic Host Configuration Protocol
  כלומר לקבל כתובת באופן דינמי מהשרת.

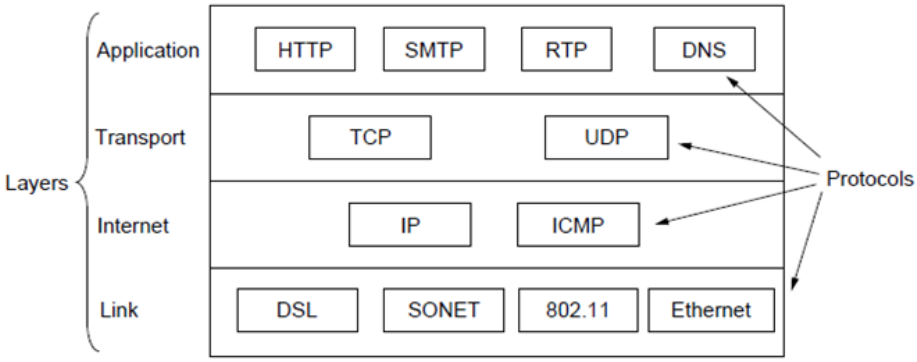
מודל ה- OSI

- **שכבת ה-Application** אחראית על נתינת שירותי תקשורת למשתמש.
- **שכבת ה-Presentation** אחראית על הכנת המידע באמצעות תרגום, הצפנה וכד' על מנת שנוכל לשלוח או לקבל את המידע בפורמט המתאים.
- **שכבת ה- Session** אחראית לשליטה בחיבור וסנכרון.
- **שכבת ה-Transport** אחראית על העברת הודעה process ל-process, בשכבה זו יש כבר אמינות כי היא דואגת שהמידע יעבור באופן שלם.
- **שכבת ה- Network** אחראית על שליחת חבילות מ-host אחד לאחר- מעביר מipn ל-ip אחר.
- **שכבת ה- Data-Link** אחראית להעביר מידע שנקרא frame מנקודה לנקודה.
- **שכבת ה- Physical** אחראית להעביר ביטים של ממש מנקודה לנקודה אחרת, ע"י סיבים אופטים, זרמי חשמל וכדומה...



מושג הכימוס (encapsulation) - מצד השולח - השכבה מקבלת מידע מהשכבה מעליה ומבצעת את תפקידה בזה שהיא מוסיפה header ו-"עוטפת" את עצמה.ומהצד המקבל -"מקלף" את השכבות.

מחסנית פרוטוקול האינטרנט



שכבת האפליקציה

תפקידי השכבה

- **תקשורת עם משתמש הקצה** - שכבת האפליקציה אחראית על התקשורת עם משתמש הקצה - ממשק המשתמש, טיפול בבקשות שונות, וכו'.
- **עיבוד הנתונים** - בהתאם להחלטות שמבצע המשתמש, שכבת היישום יכולה לדחוס את הנתונים, להצפין אותם, או לעבד אותם בכל צורה אחרת לפני שהועברו על-גבי הרשת.
- **ניהול תהליכים** - כל פעולה שהמשתמש מבצע ברשת מוגדרת כתהליך בין תחנת המוצא לתחנת היעד. שכבת היישום אחראית על ניהול התהליכים הללו, כך ששני המחשבים יוכלו להבחין בין נתונים של תהליכים שונים המתקיימים במקביל.

פרוטוקולי השכבה

- HTTP, FTP, SSL, SMTP, POP3, TFTP, NFS, RSP.
- בנוסף לפרוטוקולים משמשות בשכבה זו גם מספר שיטות דחיסה והצפנה דוגמת - ZIP.

יש לא מעט אפליקציות רשת כלומר משהו שיוצר אינטראקציה בין מכשירים. אנו צריכים לדאוג שהאפליקציה תרוץ בתחנות הקצה (מחשב, שרת, טלפון וכו') ותדבר בעזרת הרשת כאשר אנו לא צריכים לטפל בנתבים ומתגים וכדו' תפקידנו לדאוג לפתח את האפליקציה שלנו בלבד.
ישנם 2 ארכיטקטורות לאפליקציות:

1. **ארכיטקטורת Client-Server**:

Client - יוצר קשר ומתקשר עם השרת, הלקוח לא תמיד מחובר והוא יכול לנוע מכתובת רשת אחת לאחר, יכולות להיות כתובות IP דינמיות שונות.
Server - צריך לתת שירות ללקוח, צריך לדעת מה הכתובת שלו.

2. **ארכיטקטורת Peer-to-Peer**:

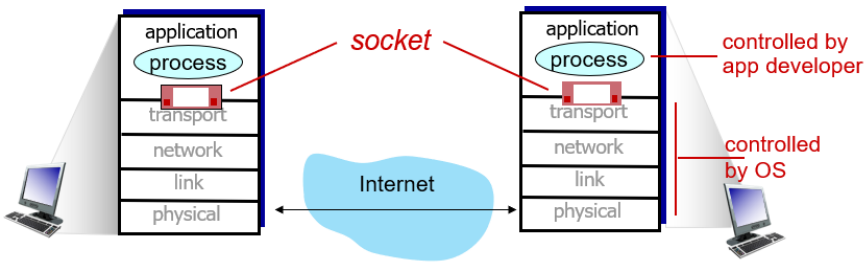
השרת לא תמיד זמין, מערכות הקצה שרירותיות ומתקשרות ישירות.
עמיתים מבקשים שירות מעמיתים אחרים (בדומה ל-torrent, skype) , נותנים שירות בתמורה לעמיתים אחרים.עמיתים מחוברים לסירוגין ומשנים כתובות IP וניהול שלו מורכב.

תהליך התקשורת (Processes communicating)

תהליך (Process) –מופע של תוכנית המנוהל על ידי מערכת הפעלה. שני תהליכים באותו מחשב מתקשרים דרך מערכת ההפעלה. בין מחשבים שונים, תהליכים מתקשרים דרך הרשת.
כאשר שני תהליכים (**process**) רוצים לדבר ביניהם הם צריכים להעביר הודעות אחד לשני.
במקרה שלנו, נוח להגיד שהתהליך הראשון זה הלקוח (client) והשני זה השרת (server) .

Socket

- כדי שתהליך יוכל לקבל/לשולח מידע לרשת/מהרשת חייבים לה "צינור" הנקרא **Socket.**
- שילוב של כתובת IP עם מספר Port מגדיר כתובת שקע (**Socket Address**) .
- Connection מזוהה ע"י צמד Sockets .



קיימים 3 סוגי Sockets :

1. **Stream Socket**: קשר מהימן קצה לקצה - TCP.
2. **Datagram Socket**: קשר Connectionless.
3. **Raw Socket** : גישה ישירה לפרוטוקולים ברמות נמוכות יותר.

במילים אחרות, בתוך host מסוים, ה-Socket הוא ממשק המתווך בין רמת ה- Application ורמת ה-Transport. דוגמא לכך שלשכבת ה-Application קיימת השפעה על שכבת ה-Transport.

בד"כ במחשב פועלים מספר סוגי יישומים בו-זמנית (בעזרת מערכת הפעלה).
כיצד המחשב יודע לסווג את הנתונים המתקבלים, ליישומים השונים הפועלים בו-זמנית?
לא מספיק רק כתובת IP אלא גם אני איזשהו כתובת פנימית איזשהו
Port - פורט פנימי שממפה אותי לשירות שאליו אני רוצה לגשת. יש כל מיני סוגי פורטים:
פורט 80 - ניגשים באישזהו HTTP, פורט 25 - מייל (לא צריך לזכור אלא רק להכיר).
הלקוח יוצר את הקשר עם הסרבר לכן הלקוח צריך לדעת לאיזה פורט לגשת בסרבר אבל כשהוא כבר מגיע לסרבר, הסרבר כבר יודע מאיזה פורט הלקוח ניגש בצד שלו ואז הוא השולח לו את הנתונים שהלקוח דורש בהתאם לדרישות.

פרוטוקולים בשכבת האפליקציה (An application-layer protocol)

כדי להגדיר מה זה הפרוטוקולים צריך שכל פרוטוקול יגיד לנו איזה סוגי הודעות הוא משתמש.
מה הסינטקס של כל הודעה, ומה המשמעות של כל אחד מהנתונים כי לא תמיד ההודעה תיהיה טקסט לפעמים זה יהיה מספרים וכו' וגם כמובן הפרוטוקול צריך לדעת מתי לשלוח ולקבל הודעות.
מבחינת פרוטוקולים יש 2 סוגים כללים:

1. **Open Protocol** - כל אחד יכול לגשת אליהם להגדיר ולממש אותם.
   כלומר אם אנחנו רוצים שאנשים ישתמשו באפליקציה שלנו הם צריכים להכיר את הפרוטוקול של האפליקציה. - זה מאפשר לנו להשתמש בכל מיני דפדפנים שנותנים לנו את אותו השירות פחות או יותר כי הם יודעים לממש את הפרוטוקול לפי ההגדרה שלו כמו עם הפרוטוקול המוכר HTTP.
2. **Proprietary Protocols** - פרוטוקול פרטי כמו skype, אני לא יכול ליצור לי איזשהו סקייפ משלי שידבר עם סקייפ אחר, כולם עובדים עם סקייפ אחד בלבד.

דרישות האפליקציה משירות התעבורה

- העברה אמינה ומלאה של המידע.
- חלק מהאפליקציות צריכות עמידות בזמנים ודיליי כמו משחקי מולטיפלייר במחשבים.
- תפוקה של נתונים כמו למשל איכות צפייה של מולטימדיה.
- אבטחה.

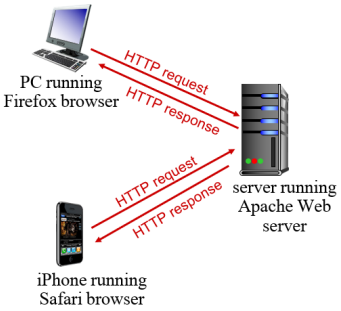
סוגי שירותים בשכבת התעבורה

1. **TCP Service -** דואג לשירות אמין כלומר דואג שהמידע יעבור בין 2 התהליכים ואם המידע לא הגיע הוא ישלח אותה שוב לאן שצריך. הוא מבטיח גם לא להעמיס את הצד השני במידע.
2. **UDP Service -** שירות לא אמין - שולח את המידע ולא מבטיח שהמידע יגיע אל היעד.

TLS - אחראי על האבטחה בשכבת התעבורה עם TCP,מאפשר הצפנה מקצה לקצה.

HTTP: פרוטוקול שכבת האפליקציה של האינטרנט

דף האינטרנט מורכב מאובייקטים, כך שכל אחד מהם יכול להיות מאוחסן בשרתי אינטרנט שונים.
האובייקט יכול להיות קובץ HTML, תמונת JPEG, יישומון ג'אווה, קובץ שמע, ...
דף אינטרנט מורכב מקובץ HTML בסיסי הכולל מספר אובייקטים שהפניה אליהם ניתנת לכתובת אתר.



- מצד הלקוח: דפדפן שמבקש ומקבל הלקוח
  (באמצעות פרוטוקול HTTP) וגם "מציג "אובייקטים ברשת.
- מצד השרת: שרת האינטרנט שולח
  (באמצעות פרוטוקול HTTP) את האובייקטים.

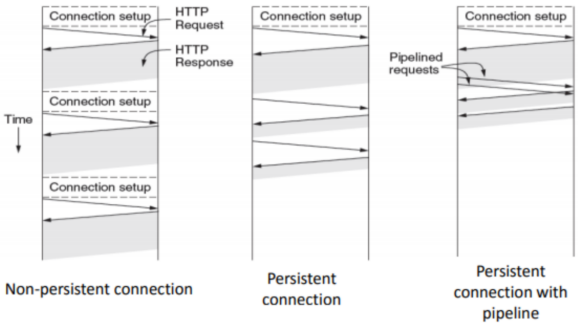
ה-HTTP משתמש ב-TCP שמבצע את פעולת החיבור מהלקוח לשרת
- (תהליך "לחיצת-היד") הוא יוצר את הסוקטים בפורט 80.
HTTP לא זוכר שום דבר על הלקוח מבחינת הבקשות וכו'...

RTT - Round Trip Time - זה פרק הזמן של העברת החבילה מהלקוח אל השרת ובחזרה.

ישנם 2 סוגים של חיבורי HTTP :

1. **Non-Persistent HTTP בסימון (1.0)** - פתיחת התחברות TCP, שליחת אובייקט אחד ויחיד ולאחר מכן התנתקות מה-TCP.
   זה שיטה לא טובה כי יש היום הרבה נתונים ברשת וזה יכול לגרום לעיכובים רבים כי אני כל הזמן פותח וסוגר את ההתחברות.

2. **Persistent HTTP בסימון (1.1)** - פתיחת התחברות TCP, שולח כמה אובייקטים ביחד ולאחר מכן התנתקות מה-TCP. - הרבה יותר יעיל ומשפר לי את זמן הגלישה.
   לגישה זו יש 2 תתי-גישות:
   א. **NP** - אנו נשלח בקשה רק לאחר שקיבלנו את התשובה לבקשה הקודמת.
   ב. **P** - נשלח כמה בקשות ואז נקבל את התשובות ביחד.



HTTP Request Messages

- POST method - מבקשת ששרת אינטרנט יקבל את הנתונים הכלולים בגוף הודעת הבקשה, ככל הנראה לאחסונם.
- GET method - משמש לבקשת נתונים ממקור מוגדר.
- HEAD method- כמעט זהה ל- GET, אך ללא גוף התגובה.
- PUT method- משמש לשליחת נתונים לשרת כדי ליצור / לעדכן משאב.

HTTP response status code

הסטטוס מופיע בשורה הראשונה בהודעת הקבלה של server-to-client, סטטוסים כמו:

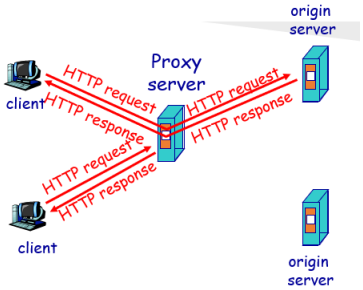
- OK 200 - הבקשה הצליחה, האובייקט המבוקש בהמשך הודעה זו.
- Moved Permanently 301 - האובייקט המבוקש הועבר, מיקום חדש צוין בהמשך בהודעה זו.
- Bad Request 400 - הודעת בקשה אינה מובנת על ידי השרת.
- Not Found 404 - המסמך המבוקש לא נמצא בשרת זה.
- HTTP Version Not Supported 505.

HTTP Header

מאפשרת ללקוח ולשרת להעביר מידע נוסף עם בקשת HTTP או תגובה.

כדי שחווית הגלישה תהיה טובה יותר בין אם זה קיצור זמן הורדת הדף ועוד כמה,
נזכיר ש-HTTP לא זוכר את הלקוח ואת הדרישות שלו ולכן נוצרו שיפורים המסייעים לטובת הלקוח:

- **HTTP Cookies** - אתרי אינטרנט ודפדפן לקוח משתמשים בעוגיות כדי לשמור על מצב כלשהו בין תיעודים כלשהם - זה יכול לתת לי המלצות טובות יותר, מצב הפעלת משתמש והתאמה לצרכי הגולש. החיסרון הוא הפגיעה בפרטיות כי הוא יודע הכל על הגולש.
- **Web Caches - Proxy Servers** (מטמון) - צמצום את זמן התגובה לבקשת הלקוח, המטמון קרוב יותר ללקוח, מאפשר לספקי תוכן להעביר את התוכן בצורה יעילה יותר.

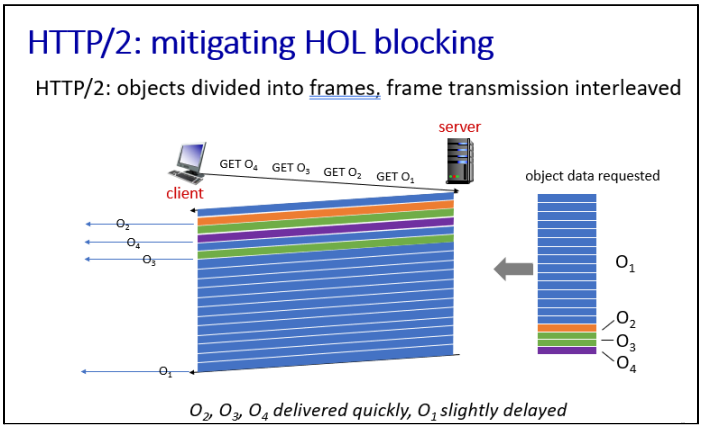


Proxy

שרת proxy משמש כשער בינך לבין האינטרנט.
זהו שרת מתווך המפריד בין משתמשי קצה לבין אתרי האינטרנט שבהם גולשים. שרתי פרוקסי מספקים רמות שונות של פונקציונליות, אבטחה ופרטיות בהתאם למקרה השימוש שלך, לצרכים שלך או למדיניות החברה שלך.

גרסאות ה-HTTP

- **HTTP/1.x -** הוא יוצר הרבה חיבורים במקביל ולא דוחס את ה-header וגם לא עובד בסדרי עדיפויות.
- **HTTP/2 -** דוחס את ה-header, הוא מאפשר לנו-**Push** שזה בעצם היכולת של השרת לשלוח תגובות מרובות לבקשת לקוח יחיד, מספיק לו התחברות TCP אחת בלבד.
  יש לו **Stream** - כלומר הצמד של request ושל ה-response.
  בנוסף גרסה זאת מחלקת את האובייקטים ל-**frames** וכך התעבורה עוברת מהר יותר.
  הבעיה בגרסה הזאת היא שעדיין TCP יחיד זה כמו צינור אחד שמכניס בתוכו הרבה streams אבל אם משהו הולך לאיבוד אנחנו נהיה בבעיה כי ה-TCP צריך לשדר את זה עוד פעם וזה יכול לעכב לי שידורים אחרים וגם אין לו אבטחה.



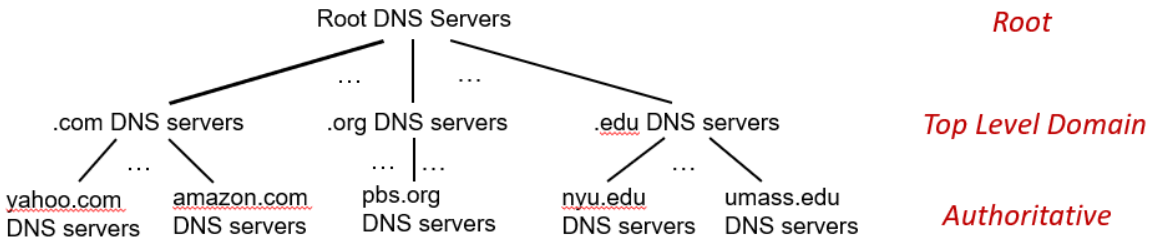
- **HTTP/3 -** יותר מאובטחת, מבקר כל שגיאת אובייקט ועומס באמצעות UDP.

DNS: Domain Name System

זה מערכת לתרגום כתובות URL ל-IP. המערכת בנויה משרתים המחזיקים מאגרי מידע בצורה מבוזרת לפי היררכיות כדי להתמודד בעיקר עם בעיות של נקודת כשל יחידה, עומס בתעבורה וגם בעיית מרחק הפיזי מתחנות היעד.

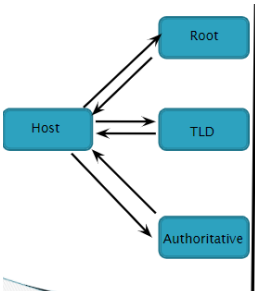
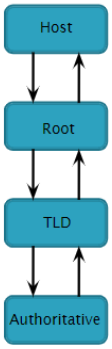
אם נרצה לגלוש לאתר מסוים אנחנו לא צריכים את הכתובת IP שלו אלא את ה-URL שלו ובשביל הדוגמה, כאשר אנו נלחץ ENTER התהליך הראשון שקורה זה תהליך של DNS כלומר ההמרה הזאת מהשם URL אל כתובת ה-IP, בתהליך הזה מעורב פרוטוקול.
תהליך זה מאפשר לנו להשתמש בשמות שלא באמת אמיתיים, מאפשר לבזר את העומס למשל יש כמה webservers אז בתהליך DNS אפשר ליצור מצב שאני מפנה חלק מהלקוחות לשרת אחר.

יש לנו כאן מערכת מאוד מורכבת כי מדובר בהרבה מאוד מידע ולכן אנו מבזרים אותו ועושים לו היררכיה והיא מחולקת בדר"כ ל-3 שכבות:
השכבה העליונה היא Root DNS Servers השכבה השניה היא Top Level Domain
והשכבה השלישית היא Authoritative.
בדר"כ כאשר אנו הולכים לאיזשהו כותבת למשל amazon.com אז מה שקורה זה שהלוקח פונה ל-root והוא יפנה אותנו ל-Top Level Domain של com ואז הוא יפנה אותנו ל-שרתי DNS של amazon.com
.



Local DNS name servers

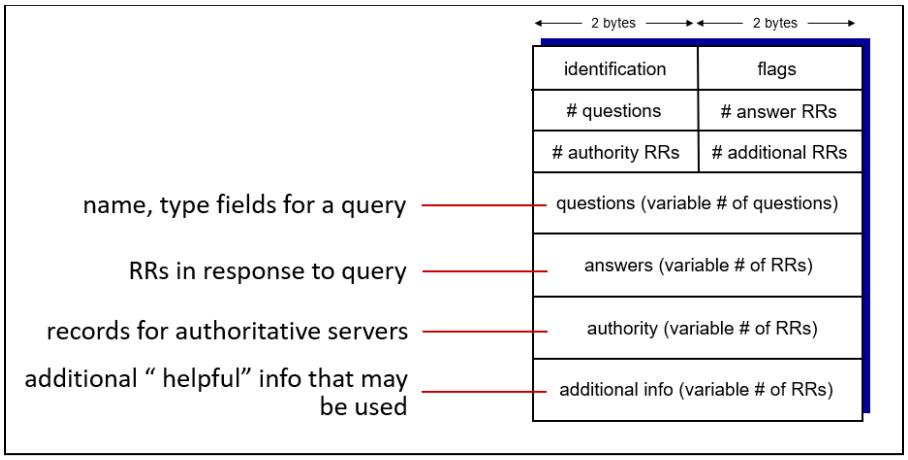
כאשר המארח מבצע שאילתת DNS, השאילתה נשלחת לשרת ה- DNS המקומי שלו.
שרת זה אינו שייך בהכרח להיררכיה, הוא נמצא באמצע בין הלקוח לבין ההיררכיה הוא עושה את כל השאילתות, יכול לשמור את התשובות שהוא מקבל וזה יכול לחסוך לנו את הזמן.
שרת מקומי לרוב יהיה באותו subnet של המשתמש.יש 2 תהליכי שאילתות, איטרטיבי ורקורסיבי...

איטרטיבי	רקורסיבי
 בשיטה זו, המחשב המבקש, שולח בקשה לאחד משרתי הRoot-, לאחר קבלת התשובה (כתובת של שרת TLD) הוא פונה (בעצמו) לשרת ה-TLD ולאחר קבלת תשובה, הוא פונה (שוב בעצמו) לשרת המהימן (Authoritative).	 שיטה זו, המחשב המבקש שולח בקשה אחת לשרת ה-Root, שרת הRoot-, **לא** מחזיר תשובה עדיין אלא שולח בעצמו את הבקשה לשרת הTLD- שאינו מחזיר תשובה עדיין אלא שולח את השאילתא לשרת המהימן, שמחזיר תשובה לשרת הTLD-, שמחזיר תשובה לשרת ה-Root, ושמחזיר תשובה למחשב המבקש.

ב-database יש לנו רשומות(RR) וכל רשומה יש לה פורמט (name, value, type , ttl) ולפי הפורמט הזה אנחנו יודעים מה האינפורמציה שאני רוצה לקבל בהודעה.
ttl = time to live כלומר כמה זמן אני יכול לשמור את הרשומה הזאת, ה-type משפיע לי במה אני מצפה לראות ב-name וב-value:

אם type=NS	אם type=A
• ה-name יהיה ה-domain למשל google.com. • ה-value של של ה-hostname שהוא אמין לאותו domain.	• ה-name יהיה hostname. • ה-value יהיה ה- IP Address.
אם type=CNAME	**אם type=MX**
• ה-name יהיה לי את ה-alias name הכוונה שיש לי את השם שנוח לי לעבוד איתו כמו למשל www.google.com אבל באמת צריך לחפש אותו בכתובת אחרת שלא נראית לעיני הגולש. • ה-values יהיה ה- שם כינוי שלו.	• ה-value יהיה השם של המסר שמחובר לאותו השם.

כך נראה פרוטוקול ההודעות של DNS,להודעות בקשה ותשובה של DNS יש את אותו הפורמט:



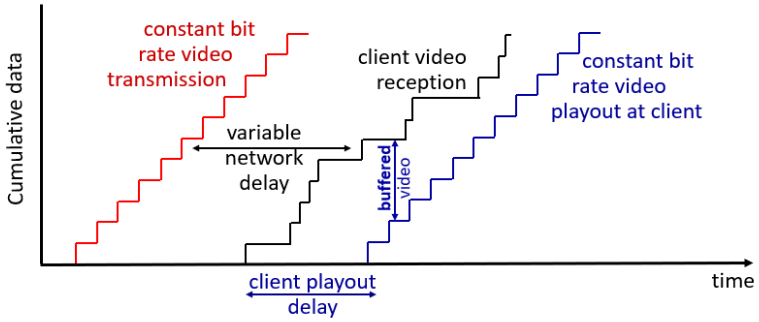
DNS Security

- DDoS Attacks - לפוצץ את שרתי ה-root בתעבורה כבדה.
  להפציץ שרתי DNS בתעבורה כבדה.
- ניצול DNS עבור DDoS כך שנשלחות שאילתות עם כתובת מקור מזויפת: יעד IP.
- שליחת שגיאות מזויפות לשרת DNS.

Video Streaming and CNDs - תכני מולטימדיה

תעבורת וידאו הוא הצרכן העיקרי של רוחב הפס באינטרנט כמו למשל Netflix, YouTube וכו'..
וידאו מוגדר מבחינתו כאוסף של תמונות המוצגות לנו בקצב קבוע, יש כל מיני פתרונות לחסיכת רוחב הפס באמצעות טכניקות תעבורת נתונים המתאימים לוידאו ולא פוגעים באיכות באופן פטאלי.

האתגר העיקרי הוא שרוחב הפס של שרת-לקוח משתנה לאורך זמן.
שינוי רמות העומס ברשת (בבית, ברשת הגישה, בליבת הרשת, בשרת הוידאו) ואובדן פקטות יכול לעכב את הפעלת הווידאו או לגרום לאיכות וידיאו ירודה.
לכן קיים ה-buffer שבזמן הפעלה טוען מראש כמות וידאו כדי למנוע תקיעות ולצופה יהיה את האפשרות לצפות חלק לכל אורך הסרטון.



הרצון שלנו ליצור מערכת שרת-לקוח שתוכל לתת לי את הסרטים ולהתמודד עם כל מיני בעיות ברשת וגם עם כל מיני רצונות של הלקוח ולתת לו את האיכות הכי טובה באותו הרגע.

DASH : Dynamic, Adaptive Streaming over HTTP

זה הבסיס לכל streaming של מולטימדיה

מצד השרת אני לוקח את הסרט מחלק אותו לחתיכות,כל חתיכה מקודדת בכמה איכויות ונבנית מעין מפה של URLS שכל אחד מהם מצביע על חתיכות שונות.

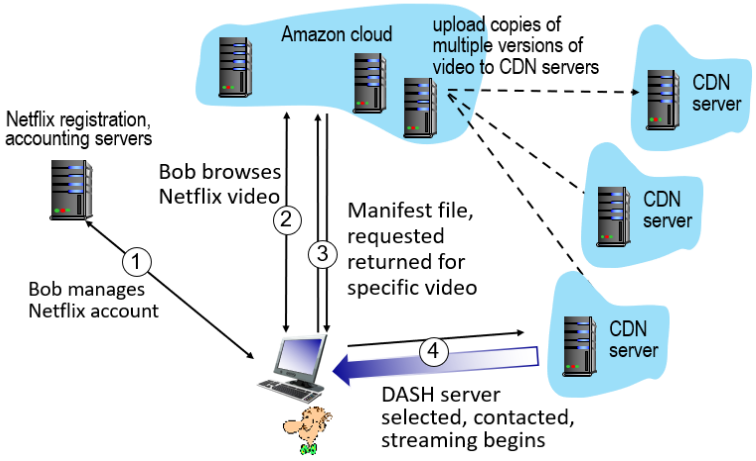
מצד הלקוח - מודד מעת לעת את רוחב הפס של השרת ללקוח, מבקש חתיכה אחת בכל פעם,

בוחר בקצב קידוד מקסימלי בהתחשב ברוחב הפס הנוכחי ויכול לבחור שיעורי קידוד שונים בנקודות זמן שונות (תלוי ברוחב הפס הזמין בזמן).

CDNs

זה אוסף של שרתים שנמצאים או בתוך ה-access network שלך או מחובר ליד IXP שלשמה אני משכפל תכנים כמו סרטים תמונות מוזיקה וכו'...

המטרה היא לספק זמינות וביצועים גבוהים על ידי הפצת השירות באופן מרחבי ביחס למשתמשי הקצה.הוא עוזר לי לדחוף את התוכן קרוב אלי (לא בהכרח גיאוגרפית) אלא מהרבה מאוד בחינות.



שכבת התעבורה

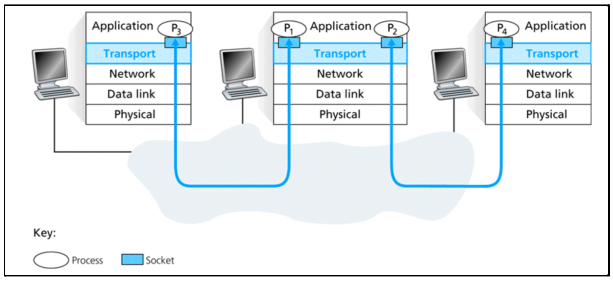
כשאנו מדברים על שירות התעבורה הכוונה שאנו יוצרים מעין חיבור לוגי בין תהליכים שנמצאים אצל המארח כלומר מבחינת המארחים הם כאילו מדברים אחד עם השני.

פעולות פרוטוקולי תעבורה במערכות קצה:
השולח: מפרק הודעות אפליקציה לקטעים (**segments**) ועובר לשכבת הרשת.
המקבל: מרכיב מחדש את ה-**segments** להודעות ועובר לשכבת האפליקציה.
שני פרוטוקולי תעבורה הזמינים ליישומי אינטרנט הם **TCP, UDP**.
נזכור ששכבה זו מחברת בין פרוסים ושכבת הרשת מחברת בין הוסטים זאת אומרת התעבורה צריכה לקחת מהפרוסים את ההודעות ולהעביר אותם לרשת.
הרשת לא מבטיחה שהמידע יגיע, וגם לא מבטיחה שיגיע לפי הסדר, יכול להיות שביטים יהיו תקולים ובשביל זה אנחנו צריכים את שכבת התעבורה.

פורמט הנתונים שניתן לזהות על ידי IP נקרא **IP-datagram.**
הוא מורכב משני רכיבים, כלומר **מה-header** ומהנתונים, אותם יש להעביר.
לשדות **ב-datagram**,למעט הנתונים, יש תפקידים ספציפיים בביצוע העברת הנתונים.

Multiplexing and Demultiplexing

מבוססים על datagrams , segments וערכי השדות של ה-header.
זה תהליך שקורה בכל השכבות.



- **socket** הוא הממשק שדרכו תהליך (יישום) מתקשר עם שכבת התעבורה.
- כל תהליך יכול להשתמש בשקעים רבים.
- שכבת התעבורה במכונה מקבלת רצף של קטעים משכבת הרשת שלה.
- מסירת קטעים לשקע הנכון נקראת **demultiplexing**.
- הרכבת קטעים עם המידע הדרוש והעברתם לשכבת הרשת נקראת ריבוב.
- **multiplexing ו-demultiplexing** נדרשים בכל פעם שנוצר שיתוף תקשורתי.

איך **Demultiplexing** עובד?
המארח מקבל IP datagrams, לכל datagram יש כתובת IP של המקור וכתובת IP של היעד.
כל datagram נושא segment אחד של שכבת התעבורה ולכל segment יש port של המקור ושל היעד.
המארח משתמש בכתובות IP וב-port כדי להפנות את ה-segment אל ה-socket המתאים.

⏪ **UDP: demultiplexing** משתמש במספר של ה-port של היעד (בלבד).

⏪ **TCP: demultiplexing** באמצעות 4-tuple:

1. כתובות IP של המקור
2. כתובת IP של היעד
3. מספר ה-port של המקור
4. מספר ה-port של היעד

ומשם ה-demux: משתמש בכל 4 הערכים האלה כדי להכיוון את ה-segment אל ה-socket הנכון.

UDP: User Datagram Protocol

- מתבסס על IP-Protocol שלא מבטיח הגעה לפי סדר והגעה באופן כללי. ההודעות יכול להאבד והוא גם לא מבטיח שלא יהיה שגיאות.
- UDP עוזר יותר בנושא השגיאות בזה שהוא מוסיף ל-header שלו שדה מסוים כך שהצד השני שמקבל יכול להיעזר בשדה הזה כדי להבין האם התרחשה שגיאה.

למה UDP?

1. הוא יכול לשדר מהר ופשוט כלומר הוא לוקח את המידע מהאפליקציה , מוסיף header , ושולח ל IP.
2. יש לו header קטן.
3. יכול לעבוד כאשר יש עומס בשונה מה-TCP שמוריד את הקצב בעת עומס.
4. לא מבזבז RTT כמו ב-TCP שעוד לפני קבלת מידע אני עושה פתיחת קשר.

מי משתמש ב-UDP?

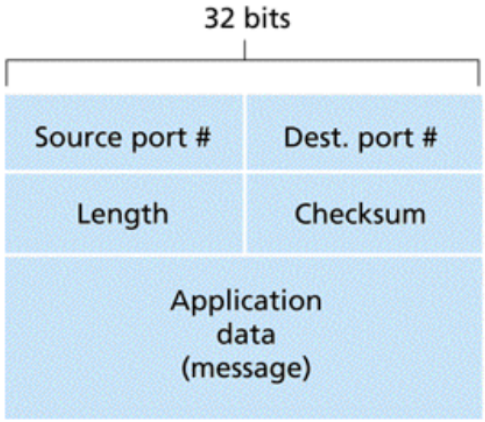
- אפליקציות סטרימינג (amazon prime, netflix, disney plus וכו'...)
- DNS
- SNMP - (פרוטוקול תקשורת לניהול התקני רשת ברשתות IP)
- HTTP/3 הוא דוגמה לפרוטוקול שהפסיק להשתמש ב-TCP והוא עובר ל-UDP אבל כדי לעשות את זה הוא צריך להעביר משימות שהיו ב-TCP כמו נניח ה-reliability וה-congestion control משכבת התעבורה אל שכבת האפליקציה.

UDP Segment Structure

יש לו 4 שדות בלבד, כל שדה בנוי מ-16 ביטים ואת ה-header שלו.

- Source port - מזהה את תהליך UDP ששלח ה-segment
- Dest. port - מזהה את תהליך ה- UDP אשר יטפל בנתוני האפליקציה.

- Length - מציין את אורך ה-segment, כולל ה-header בביטים
- Checksum - מזהה שגיאות.



UDP Checksum

הוא לוקח את הביטים מסדר אותם כמו במטריצה שהרוחב שלה הוא 16 , מחשב את הסכימה של הביטים ובסוף הוא מפעיל NOT כלומר ההופכי שלו, הצד השני לוקח את המידע אם הוא יעשה סכימה עם ה-checksum של השולח יצא לו רצף של אחדות - כלומר שהכל תקין, אחרת יש שגיאה.

העקרונות של העברת מידע בצורה אמינה

מבחינתנו אנחנו היינו רוצים ליצור מצב שיש לי תהליך של שולח-מקבל ואני רוצה ליצור מצב שאני יוצר לאפליקציה איזשהו שירות אמין כאילו הערוץ ביניהם הוא אמין.

reliable service *abstraction*

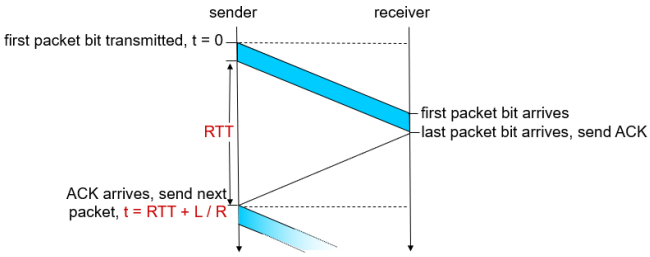
לכן היינו רוצים:

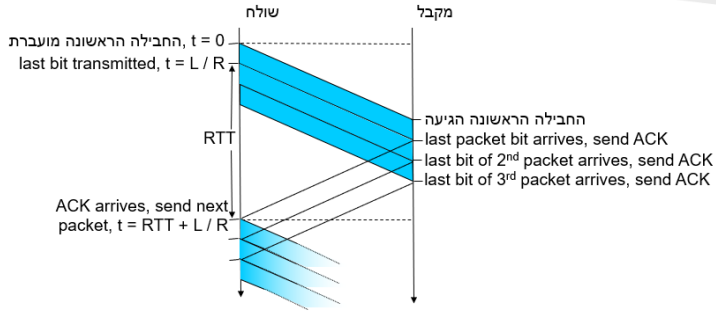
מנגנון	שימוש
Checksum	תפקידו לזהות שגיאות של ביטים בחבילה העוברת
Timer	משמש להעברה מחודשת אם החבילה או ה- ACK אבדו (או התעכבו).
Sequence number	מספור רציף של זרימת חבילות. משתמש ב-gaps כדי לזהות חבילות שאבדו, ומשתמש במספרים כפולים לאתר חבילות כפולות.
Acknow-ledgment	מאשר חבילות שהתקבלו כהלכה; מכיל מספר סידורי; עשוי להצטבר או להיות אינדיבידואלי.

Negative ack-nowledgment	אומר לשולח שהחבילה לא התקבלה.
Window, pipelining	שליחת מנות רק בטווח מוגבל, מגדילה את השימוש של השולח בהשוואה ל-stop-and-wait.

פרוטוקולים/מנגנונים מוכרים שמספקים אמינות

1. **Stop-and-wait -** כלומר שידרת - תעצור ותחכה, ורק אחרי שקיבלת ACK אז תמשיך.תרצו או לא תרצו החוקים האלה מביאים אותנו לפרוטוקול אמין כי אם המידע לא הגיע לצד השני הוא ישדר את המידע עוד פעם ויתעקש על שידור החבילה עד אשר היא תגיע לצד השני האם זה יעיל? - כמובן שלא, אבל זה עדיין אמין.





2. **Pipelining** - משפר את היעילות הזאת - במקום לשדר חבילה אחת ולחכות, אנו נשדר כמה חבילות יחדיו.

3. **Go-Back-N** -

⏩ מצד השולח:

נותן לשולח חלון,ה-N הוא גודל החלון.

זה אומר שהחלון מאפשר לי לשדר N חבילות אחת אחרי השניה.

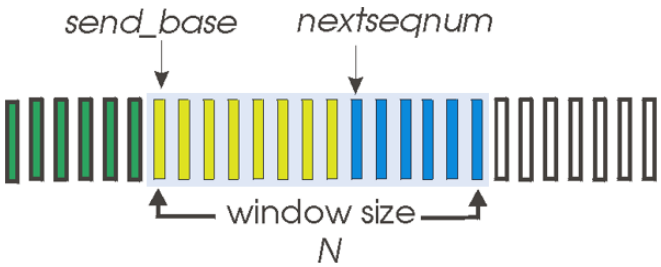
בכל רגע נתון הוא צריך לדעת מה החבילה הראשונה בחלון ומה הבאה שאני יכול לשדר.

ירוק- חבילות ששודרו וקיבלנו עליהם ACK כלומר טיפלנו בהם.

לבן - חבילות שעדיין לא התעסקנו.

צהוב - חבילות שנשלחו אבל לא קיבלתי עליהם עדיין אישור.

כחול - חבילות שאני יכול לשלוח אבל עדיין לא שלחתי אותם.



- המנגנון הזה עובד עם ACK מצטבר כלומר שאני מקבל ערך מסוים ב-ACK וגוזר ממנו שכל החבילות עד אותו ערך כולל - הגיעו.
- ה-ACK לא מאשר לי חבילה ספציפית אלא הוא מאשר לי את כל החבילות ברצף עד נקודה מסוימת בלי קפיצות.
- TIMEOUT of N לאיזשהו חבילה N, שמשדר ממנה ועד סוף החלון ולכן השם של המנגנון.

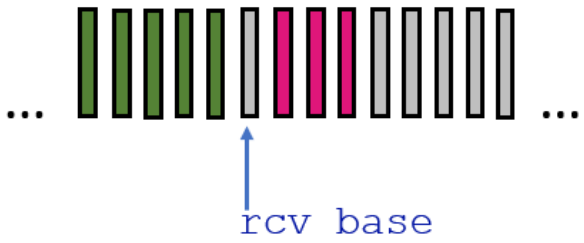
⏪ מצד המקבל:

אם אני מקבל חבילה לא בסדר אני יכול לשמור אותה או אם אין לי מקום לשמור אותה היא נזרקת ואני מוציא ACKs אך ורק על רצף של סדרה כלומר אני תמיד אסתכל על ה-rcv_base וכלומר על ה-ACK האחרון ואם הוא עדיין לא הושלם אז עדיין לא סיימתי את התפקיד שלי.

ירוק - התקבל בהצלחה עם ACK.

ורוד - לא בסדר : התקבל אבל ללא ACK.

אפור - לא התקבל.



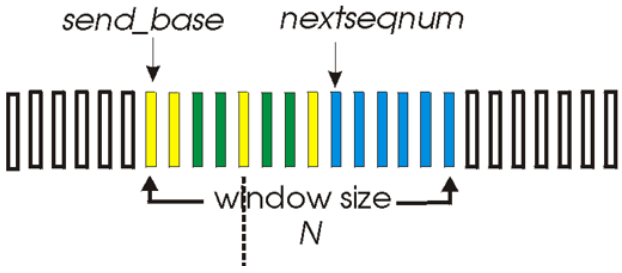
4. **Selective Repeat** -

⏩ מצד השולח:

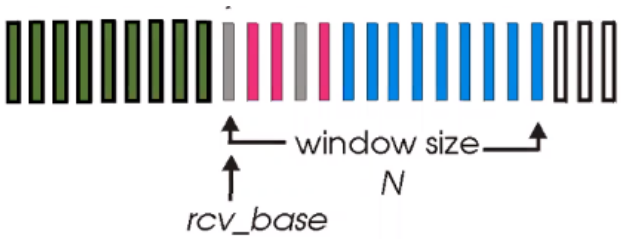
אני יוצא מנקודת הנחה שלצד השני יש חלון קבלה, לכן אני אומר - אני רוצה ומבקש לאשר אינדיבידואלית כל חבילה שאתם מקבלים.

לכן אם יש שידור חוזר אני משדר חבילה אחת שעליה אנו מדברים.

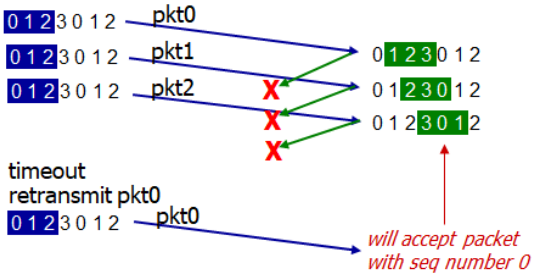
החלון אותו חלון כמו שאנו מכירים הכוונה שהוא באותו התייחסות אבל כאן בגלל שהאישור הוא פרטני יכול להיווצר מצב שבתוך החלון יהיו לי חבילות מאושרת שאישרו אותם (הירוקות) ועדיין יכול להיות חבילות שלא קיבלתי עליהם אישור ולאחר מכן תמיד חבילות שאני יכול לשדר ועדיין לא שידרתי.



⏪ מצד המקבל:
ויש גם חלון קבלה שיכול להיות בסטטוס שונה מהחלון שליחה והוא מתייחס לחבילות שהוא קיבל, חבילות שחסרות לו וחבילות שיכול לקבל אבל עדיין אין שום התייחסות עליהם.



חשוב! - הדילמה של המנגנון הזה, הבעיה היא שלא הגיע ACK על החבילה הראשונה.



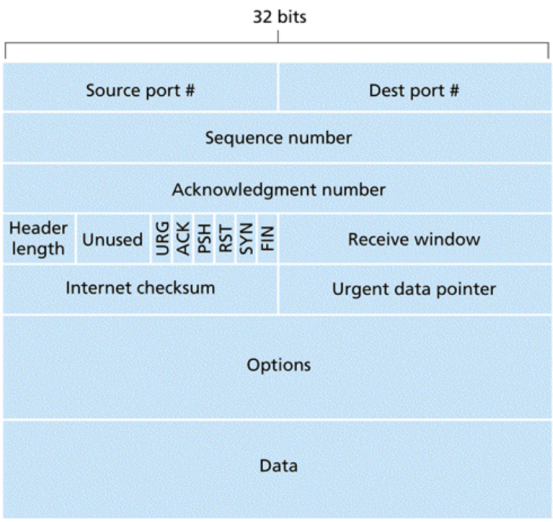
שאלה - איזה יחס נדרש בין גודל הרצף לגודל החלון כדי למנוע את הבעיה ?
תשובה - כדי להתמודד עם המצב שחלון השליחה שונה לחלוטין מחלון הקבלה ולא יהיה לי חזרה על אחד האינדקסים אני צריך שהיחס של גודל חלון השליחה לבין טווח הסדרה יהיה **פי 2**. **כלומר שהסדרה תהיה גדולה פי 2 לפחות**.
זאת אומרת שאם הסדרה הייתה 0,1,2,3,4,5 וגודל החלון הוא 3 אז היה מתקיים כמו שצריך ללא שום בעיה.

TCP: Transmission Control Protocol

- הוא point-to-point הכוונה שיש לו שולח ומקבל והמידע יכול לעבור ב-2 הכיוונים כלומר שאם אני פותח קשר עם מישהו אחר אז אני מדבר איתו והוא מדבר איתי.
- אני מעביר בצורה של בתים ולא בצורה של הודעות.
- מאפשר לי העברת הודעות משכבת האפליקציה ללא הגבלה.
- לרוב אני משתמש ACK מצטבר.
- עובד ב- Pipeline כלומר יכול לשלוח segment אחרי segment כלומר אני לא צריך לחכות לאישור כדי לשלוח את השני.
- פתיחת קשר לפני שליחת המידע.
- משתדל לא להעמיס על הצד השני.

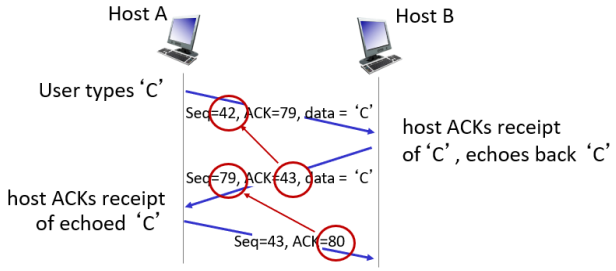
TCP Segment Structure

- source, dest, checksum - כמו שראינו ב-UDP.
- Sequence number and *Acknowledgement number* - משמשים ליישום העברה אמינה.
- Header length - אורך ההאדר.
- Flags field -
  1. URG - מציין כי השולח סימן נתונים מסוימים כדחופים.
  2. ACK - מציין ששדה *Acknowledgement number* בתוקף.
  3. PSH - מציין כי יש להעביר אותו באופן מיידי (PUSHed) .
  4. RST - משמש לאיפוס חיבור, כלומר חיבור מבולבל או סירוב.
  5. SYN - משמש ליצירת חיבור.
  6. FIN - משמש לסיום חיבור.
- Receive window - מזהה כמה שטח זמין עבור נתונים נכנסים.



TCP Sequence numbers and ACKs

- **Sequence numbers**: זרם בתים "מספר" של בית ראשון בנתוני הקטע (לא כמו ספירה רגילה מ1 עד N) אלא לפי.
- **Acknowledgements**: מספר מספר בתים הבאים צפוי מהצד השני, ACK מצטבר



TCP round trip time, timeout

כמה זמן אני מחכה עד שאני מחליט שה-segment לא הגיב?

מה הערך של הtimeout שאני אגדיר?

אמרנו שזה קשור ל-RTT אבל כולנו יודעים ש-RTT הוא לא ערך קבוע.

לכן אנחנו צריכים להעריך את ה-RTT אבל איך נעשה זאת.

SampleRTT: זמן נמדד מהעברת ה-segment ועד קבלת ACK , לא כולל חישוב שידורים חוזרים.

SampleRTT לא ערך קבוע, הוא משתנה כל הזמן, לכן נעשה ממוצע של כמה מדידות אחרונות,

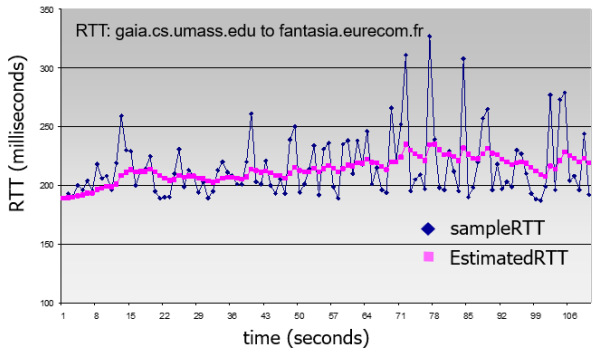
ולא רק של ה- SampleRTT הנוכחי.

פונקצית הממוצע המשוקלל תראה כך:

EstimatedRTT = (1- a)*EstimatedRTT + a*SampleRTT

ערך אופייני: a=0.125 כלומר $a = \frac{1}{8}$.

לכן



את ה-RTT בכחול אפשר לראות ש-segment אחרי segment ה-RTT לא אחיד באמת.

לכן ה-sample לא טוב לנו כי הוא כל הזמן משתנה אני צריך את הממוצע של האחרונים נסתכל על ה-Estimated.

אבל כשאנחנו משערכים משהו יכול להיות טעויות

ולכן קיים הפתרון שבו נכניס לחישוב שלנו כמה תאים נוספים:

TimeoutInterval = EstimatedRTT + 4*DevRTT

הערכה של RTT

"טווח ביטחון"

כלומר נחשב את הטעות באותו השיטה, כאשר $\beta = \frac{1}{4}$:

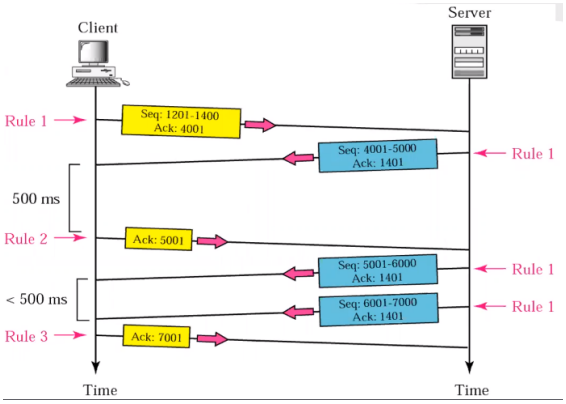
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|

הטעות המצטברת

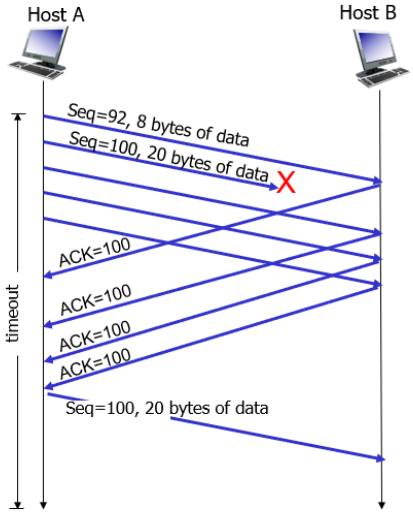
הטעות הנוכחית

TCP ACK generations rules

1. אם אני רוצה לשלוח מידע, אין לי סיבה לשלוח את המידע וה-ACK ביחד זה מיותר.
   אני יכול לשלוח אותם ביחד זה נקרא (piggyback) כלומר אני לוקח את ה-ACK ומכניס אותו לתוך ה-header של ה-data.
2. אם אני מקבל איזשהו segment וקיבלתי אותו לפי הסדר ואין לי מידע לשלוח אולי כדאי לחכות קצת ואז או שהזמן הקצר יעבור או שיגיע לי עוד segment לפי הסדר ואז יש לי שני segments שאני יכול לאשר.
3. אם מגיע לי משהו שראיתי כבר או שהגיע לי לא לפי הסדר או כל דבר אחר אז פשוט נתן לו לשלוח את ה-ACKs כמה שיותר מהר.



TCP Fast Retransmit



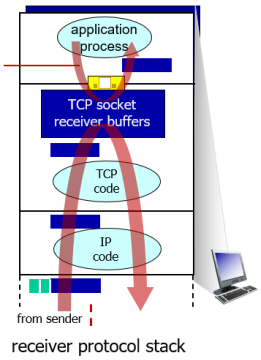
המנגנון עושה אופטימיזציה, אם אני מקבל שכפולים של ACK אז זה אומר שה-segment ששידרתי לא הגיע אל היעד.

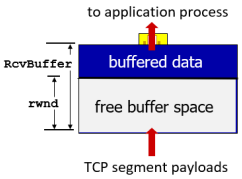
במקום להתעקש ולחכות שה-timeout ייגמר בלי לעשות כלום אז המנגנון ינצל את הזמן היקר הזה וישדר את ה-segment שוב פעם אל היעד כי מן הסתם ידוע לנו שבניסיון הקודם הוא לא הגיע.

ההסכמה היא שאם חזרו אלינו 3 פעמים אותו ACK אז נבצע שידור נוסף של ה-segment שלא הגיע אל היעד.

TCP Flow Control

זה מנגנון, במקרה ויש עומס על המקבל אז ה receive window של המקבל מעדכן את השולח שיש עומס ומודיע לו כמה מקום נשאר לו בשביל כמות החבילות שצריך.
אם יש אפס מקום הוא לא ישדר יותר.





rwnd - זה אומר כמה מקום נשאר לי.
ככה אני יכול להגביל את ה-sender הוא ישלח מידע בצורה כזאת שהיא לא תעמיס את ה-receiver.
המושג "in-flight" זה בעצם כל ה-segments שלא קיבלו ACK הם יכולים להיות segments שכבר שידרתי לא שידרתי.

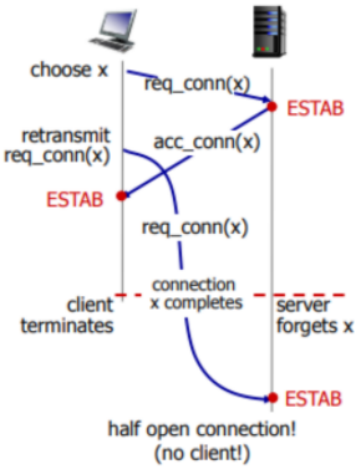
בהם בדרך וגם כאלה שעדיין

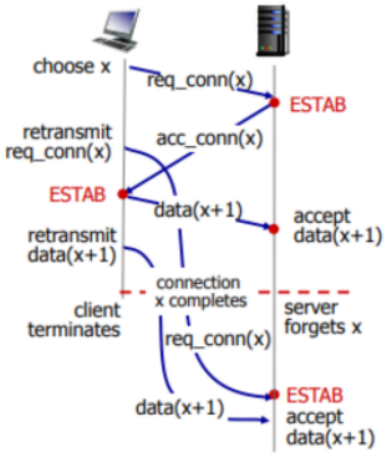
TCP Connection Management

לפני שאני שולח מידע אני חייב לבצע "לחיצת ידיים" כלומר פתיחת קשר בין השולח למקבל.
לחיצת 2 ידיים זה כמו בין שני אנשים שפותחים בשיחה.

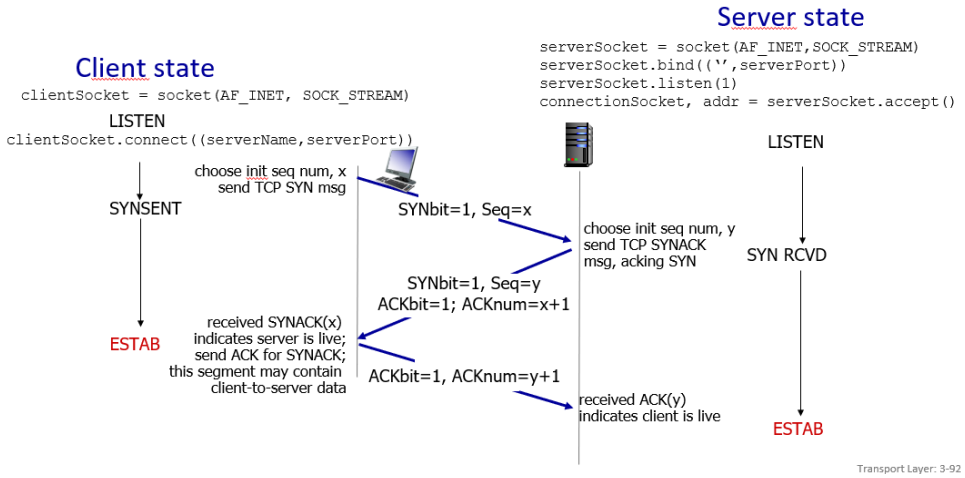
$$Socket\ clientSocket\ =\ newSocket("hostname",\ "port\_number");$$
$$Socket\ connectionSocket\ =\ welcomeSocket.accept();$$

באיור השמאלי הקשר נפתח פעמיים ללא סגירה ולכן יש בזבוז משאבים.
באיור הימני הלקוח מבקש קשר וגם שולח מידע לשרת לפני שקיבל ממנו accept .
אחר כך הלקוח שולח שוב את המידע ואנחנו כבר במצב timeout ורק אז השרת מקבל accept על המידע ולכן נפתח כאן קשר פעמיים כלומר בזבוז משאבים ושליחת מידע פעמיים.





זה לא מספיק לחיצת 2 ידיים כי אני מבזבז משאבים, כי עד שהשולח יעביר מידע למקבל, הקשר כבר ייסגר וכל תהליך לחיצת 2 הידיים יהיה מבוזבז סתם.
נשתמש בלחיצת 3 ידיים- בצורה זו הקליינט/שרת יודע להתעלם מכפילויות.
באיור אפשר לראות שה-Client מניח שה-connection פתוח רק אחרי ה-SYNACK מה- Server.
וה-Server מניח שה-connection פתוח רק אחרי ה-ACK מה-Client כלומר received ACK(y) is alive.
אני שולח הודעה נוספת מצרף את ה-ACK לתוך ה-data של ה-header.
אם יש מספר כלשהו כפול אז השרת ידע להתעלם ממנו.

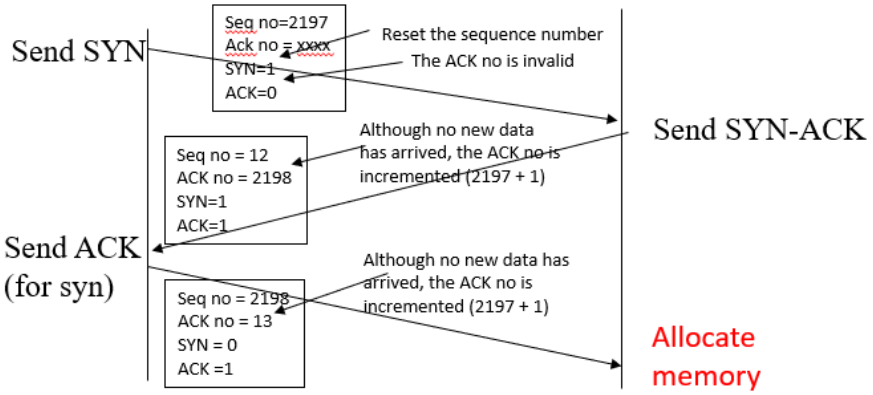


התנתקות מ-TCP

יש אפשרות המידע עובר באופן דו-צדדי (full-duplex) כלומר שמידע ממש בין הקליינט לשרת יכול לעבור בין שני הכיוונים. זה שהקליינט סיים לשדר זה לא אומר שהשרת סיים לשדר ולכן יש הודעת FIN מכיוון הקליינט והשרת יאשר אותה ואחריה יהיה הודעת FIN מצד השרת והודעת ACK מצד הקליינט ורק אז נסגר הקשר.

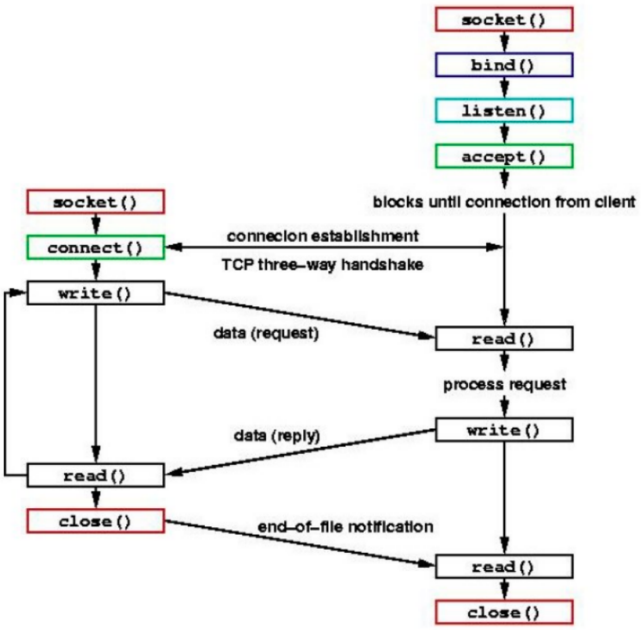
SYN Flooding Attack

נניח ואני תוקף שרת כלשהו, אני מציק לשרת ומוציא אותו משירות בזה שאני פותח מולו הרבה מאוד התחברויות ולוקח לו את כל המשאבים שלו ומציף אותו בבקשות הגורמות לעומס.
אז מצד השרת, אם לקוח מבקש יותר מדי משאבים אז אני אגביל אותו.
כלומר לכל IP יהיה אפשרות להתחבר אלי ב-x התחברויות, אבל זה עדיין לא יעזור לי, כי התוקף לא מעניין אותו לקבל תשובה הוא רוצה להוציא את השרת מאיזון ולכן הוא יכול לזייף את כתובת ה-ip של המקור כלומר לשלוח מלא הודעות מכל מיני ips .
אז מה אפשר לעשות כדי להימנע מתקיפה שכזאת? -
בעזרת **SYN Cookie** -העוגייה מכילה מספר סודי שנוצר בהתבסס על ה-IP Address וגם מפתח סודי.
השרת לא שומר מפתח זה מכיוון שהוא יכול ליצור אותו שוב מתי שהוא רוצה על סמך המידע.
אז אם קליינט שולח SYN ומקבל SYN+ACK מהשרת אז הקליינט יהיה חייב לענות עם ACK שמתאים בדיוק להודעת ה-SYN שנשלחה אליו מהשרת.
אחרי זה, השרת בודק את ה-ACK שאמור להיות ה-"סוד" שהופק מ- previously+1 .

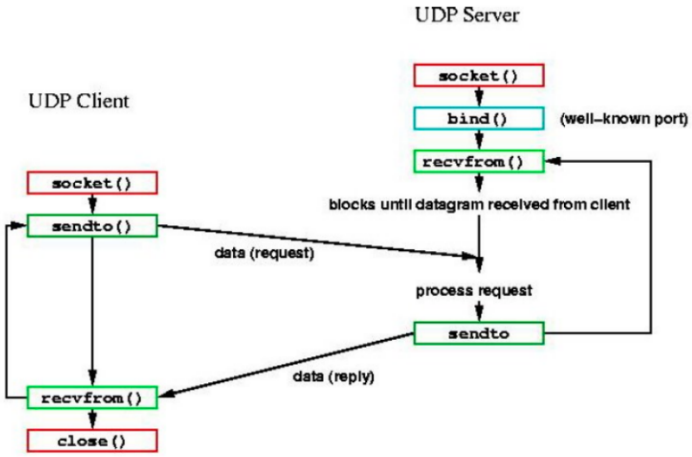


יצירת Socket בשפת C

עבור פרוטוקול ה-TCP נראה את התהליך הבא בדיאגרמה:



ועבור פרוטוקול ה-UDP נראה את התהליך הבא בדיאגרמה:



כדי **להתחיל** את הכל צריך לדרוש את הספריות הבאות מעל מערכת LINUX:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <arpa/inet.h>
```

קריאת מערכת SOCKET הערך חזרה של הסוקט יהיה FILE DESCRIPTOR
הערך של הסוקט יהיה INTEGER שפשוט יצהיר לי איפה לקרוא ולכתוב.

```
int socket(int af, int type, int protocol);
```

- הסבר:
  af – זה פרוטוקול (לדוגמה עבור IPv4 הערך הוא: AF_INET ועבור IPV6 הוא AF_INET6),
  type- סוג ה-socket (ה-SOCK_DGRAM יצור לי סוקט של UDP וה-SOCK_STREAM יצור לי סוקט של TCP).
  protocol– פרוטוקול שכבת תעבורה: עבור TCP: SOCK_STREAM, עבור UDP: SOCK_DGRAM.
  ערך 0- ייתן את ברירת המחדל: בהתאם לסוג ה socket.
- במידה ולא הוקם socket הפונקציה תחזיר את הערך -1.

דוגמה **לפתיחת** סוקט:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

בדוגמה יצרנו socket בפרוטוקול IPv4 עם SOCK_STREAM המאפשר לנו לקיים תקשורת אמינה ועם ערך 0, ברירת המחדל.

מצד הלקוח נפעל בפעולות הבאות:

הקמת חיבור עם שרת (במקרה של פרוטוקול TCP, פרוטוקול UDP עובד ללא הקמת חיבור)

ע"י קריאת הפונקציה הבאה:

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

הפונקציה מחזירה INTEGER, כל פונקציה שנכשלה תחזיר לנו -1.

אני אומר לקליינט "תעשה connect" דרך הסוקט sockfd שפתחנו לפני רגע.

תעשה connect לסרבר שהקונפיגורציה שלו נמצאת בתוך ה-struct sockaddr ואנו מעבירים את הגודל של המבנה addrlen.

- sockfd - זה socket descriptor אשר מוחזר על-ידי פונקציה ()socket מהשלב הקודם.
- serv_addr - מצביע למבנה sockaddr המכיל כתובת ופורט של שרת אליו אנו מתחברים.
  זה מבנה שאותו מעביר לסוקט או מעביר לפונקציה שנשתמש במהלך הקוד.
- addrlen - גודל מבנה sockaddr בבייתים.

מבנה sockaddr מכיל מידע על ה-IP של הסוקט עבור סוגים שונים של סוקטים.

```
struct sockaddr {
unsigned short sa_family; // address family, AF_xxx
char sa_data[14]; // 14 bytes of protocol address
};
```

- sa_family - יכול להיות AF_INET6 או AF_INET שיצהיר לנו איך אני הולך לקרוא את הבייתים הבאים.
- sa_data - מכיל כתובת ופורט של יעד.

```
// (IPv4 only! for IPv6- please see struct sockaddr_in6 )
struct sockaddr_in {
short int sin_family; // Address family, AF_INET unsigned
short int sin_port; // Port number
struct in_addr sin_addr; // Internet address
unsigned char sin_zero[8]; // Same size as struct sockaddr
};
```

בתור מתכנת, אני עובד עם מבנה מסוג sockaddr_in אני ממלא את המבנה בקטע קוד מלמעלה,
אחרי זה אני עושה cast מהמבנה הזה למבנה שיצרנו לפניו שהוא ה- sockaddr .

- sin_zero - צריך לאפס לפני כל שימוש על-ידי פונקציה (memset.
  נניח והייתי צריך להוסיף עוד מידע כלשהו אז אני צריך עוד קצת זיכרון בצד כדי שלא נשנה ארכיטקטורה אחרת (נצביע על ערך זיכרון שלא שלנו).
  המערך הזה הוא כאן לייתר הביטחון שהסברנו אבל לא נשתמש בו לצורך אחר מזה.
- sin_family - אותו הדבר כמו שראינו במבנה sockaddr.
- sin_port - צריך להמיר לייצוג של Network Byte Order באמצעות הפונקציה (htons הכוונה היא לאיזה פורט אנו רוצים להתחבר מהצד השני.
- sin_addr - צריך להמיר IP כתובת בייצוג בינארי (IPv4) ודצימאלי (IPv6) לייצוג רשת על-ידי הפונקציה (inet_pton. (אני פשוט ממלא בערך זה כתובת IP).

הפונקציה **memset**:

```
void memset(void *str,int c, size_t n)
```

אותה הפונקציה שעוזרת לנו לאפס את sin_zero שנמצא במבנה sockaddr_in.
היא מוחקת n תווים ראשונים של מחרוזת str אשר המצביע אליה מתקבל כפרמטר ובמקומם רושמת את c.

לדוגמה:

```
struct sockaddr_in serverAddress;
memset(&serverAddress,0, sizeof(serverAddress));
```

אני מעביר ל-memset את הכתובות, אומר לה לאפס (הארגומנט השני) באורך של הכתובת ששלחתי לה (הארגומנט השלישי).

Byte Order - בגלל שמייצרי המעבדים של המחשבים שלנו לא עובדים באופן אחיד,
כמו למשל שיש מעבדים שעובדים עם Big Endian ויש כאלה שעובדים עם Little Endian
אז נקבע שהרשת עובדת בסדר של Big Endian כלומר כל דבר שאני מעלה לרשת, שולח או מקבל אלי נתייחס אליו באופן של Big Endian.

- במחשבים בייתים נשמרים ב-Host Byte Order.

כעת נציג פונקציות להמרה (אחד מהם נמצא עוזר לפונקציה sin_port שבמבנה sockaddr_in):

```
htons() // host to network short
htonl() // host to network long
ntohs() // network to host short
ntohl() // network to host long
```

לדוגמה:

```
serverAddress.sin_port = htons(5000); // short, network byte order
```

אם נרצה להמיר כתובת IP (כתובת זו היא לא INTEGER) לכן יש לנו פונקציה בשם inet_pton():

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6
inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

כעקרון אנו שולחים לה את סוג הכתובת למשל IPv4 , נעביר לה את הכתובת ב-String כלומר בצורת char* והפונקציה לבד תעשה העבודה של לחלק את הכתובת לארבע כתובות וכו' ואחרי כל זה היא תהפוך

אותו לייצוג בינארי ב-Big Endian .
הארגומנט השלישי אומר לפונקציה שתשמור על השינוי שלה בכתובת ששלחתי לה.
הפונקציה מחזירה 1- אם יש שגיאה או 0 אם ההמרה לא עברה בהצלחה.
יש לוודא שערך המוחזר על-ידי פונקציה inet_pton גדול מ-0.
נשים לב שנעזרנו בפונקציה זו ב-sin_addr שבמבנה sockaddr_in.

נניח שאנו מקבלים הודעה "מבחוץ", קיבלנו חבילה מסוימת שהיא בייצוג בינארי ב-Big Endian.
אז איך נהפוך את הכתובת IP למשהו שרלוונטי לקריאה בשבילנו?
נשתמש בפונקציה inet_ntop.

```
// IPv4:
char ip4[INET_ADDRSTRLEN]; // space to hold the IPv4 string
struct sockaddr_in sa; // pretend this is loaded with something
inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
printf("The IPv4 address is: %s\n", ip4);

// IPv6:
char ip6[INET6_ADDRSTRLEN]; // space to hold the IPv6 string
struct sockaddr_in6 sa6; // pretend this is loaded with something
inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);
printf("The address is: %s\n", ip6);
```

מצד השרת נפעל כך:
כאשר אני פותח סוקט יש לי הרשאות, אני צריך לקשר את הסוקט הזה לאיזשהו **פורט** ספציפי שאני אחראי עליו ואוכל לקרוא ולכתוב עליו.
לכן יש לו את הפונקציה ()**bind** כלומר, קריאת מערכת.

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- sockfd - זה socket descriptor אשר מוחזר על-ידי פונקציה socket שיצרנו בשלב הראשון.
- my_addr - מצביע למבנה sockaddr המכיל כתובת ופורט של שרת אליו מתחברים.
- namelen - גודל המבנה של sockaddr בביתים.
- הפונקציה מחזירה 1- אם יש שגיאה. אחרת, היא תחזיר 0.

הפונקציה setsockopt תביא לנו שליטה על ההגדרות של הסוקט.
לפעמים מנסים להפעיל שרת אך קישור ה-bind נופל עם הודעה "Address already in use" .
הבעיה נובעת מכך שביט קטן של סוקט שהיה מחובר עדיין בשימוש וזה תופס פורט.

על מנת לאפשר שימוש חוזר של פורט, יש לנו משהו שנקרא SO_REUSEADDR ויש עוד הרבה FLAGS שנוכל להיעזר בהם. ניתן להוסיף את הקוד הבא:

```
int yes=1;
if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes) == -1){
        perror("setsockopt");
        exit(1);
}
```

אחרי פעולת bind השרת עובר למצב המתנה (מדובר על TCP בלבד).
מאחר ומדובר עכשיו בצד השרת, הוא חייב להאזין במקום מסוים, הוא **מאזין** ב-IP שלו עם פורט מסוים עם ה-bind.

```
int listen(int sockfd, int backlog);
```

- sockfd - זה socket descriptor שמוחזר על-ידי הסוקט.
- backlog - גודל מקסימלי של תור הבקשות לחיבור.
  תור הבקשות לחיבור מוגבל על-ידי כמות החיבורים שניתן להקים.
- הפונקציה מחזירה 1- אם יש שגיאה. אחרת, היא תחזיר 0.

הוצאת בקשה לחיבור מתוך תור הבקשות (מעובר על TCP בלבד).
ה-listen מחכה לחיבורים ומי שיקבל את החיבורים ז פונקציית **ה-accept**.
היא תקבל את החיבור ותחזיר לנו סוקט ששייך לחיבור הזה ספציפית (לקליינט).
אני מספק לו sockaddr (מבנה) שבעצם אם אני בתור קליינט בא להתחבר לשרת הזה אז השרת הזה יאחסן את ה-IP שלי ואת הפורט שלי וכו בארגומנט addr* .
ע"י קריאת פונקציה:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- הפונקציה מחזירה socket descriptor של סוקט חדש שנוצר ומכניסה למבנה sockaddr מידע אודות הלקוח שמתחבר (IP ו- Port).
- addlen - משתנה שלם מקומי אשר מגדיר את גודל המבנה sockaddr.
- הפונקציה מחזירה ערך 1- אם יש שגיאה ואחרת תחזיר 0.

אם נרצה **לשלוח** מידע לפי פרוטוקול TCP נשתמש בפונקצית **send**() .
חשוב לציין שאם אני ב-TCP אז לפני ה-send עשינו connect ולכן יש לנו חיבור ויש לנו לאן לשלוח.

```
int send(int sockfd, const void *msg, int len, int flags);
```

- sockfd - זה socket descriptor דרכו אנו שולחים את המידע.
- msg - מצביע על המידע שאנו שולחים.
- len - גודל המידע בביתים.
- flags - ערך שלו הוא 0.
- הפונקציה מחזירה את כמות הביתים שנשלחה בפועל. כמות היכולה להיות פחות ממה שרצינו לשלוח.
- יש לוודא כמה נשלח בפועל ולהשלים את מה שחסר.
- הפונקציה מחזירה 1- אם יש שגיאה.

דוגמא לשליחת הודעה בפרוטוקול TCP:

```
char *msg = “Haim was here!";
int len, bytes_sent;
. . .
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
```

- הפונקציה send תחזיר לי כמה ביטים נשלחו בפועל והם ישמרו ב-bytes_sent .

כשנרצה **לקבל** מידע לפי פרוטוקול TCP :

```
int recv(int sockfd, void *buf, int len, int flags);
```

- sockfd - זה socket descriptor דרכו אנו קוראים את המידע המתקבל.
- buf - מציע ל-buffer אשר קוראים מתוכו.
- len - הגודל המקסימלי של ה-buffer בביתים.
- flags - הערך שלו הוא 0.
- הפונקציה מחזירה את כמות הביתים שנקראו בתוך ה-buffer.
- הפונקציה מחזירה 1- אם יש שגיאה.
- הפונקציה מחזירה 0 אם החיבור נסגר.

אם נרצה **לשלוח** מידע מעל UDP חשוב לציין שאין לנו חיבור קודם ולכן נשתמש בפונקציה הבאה:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
        const struct sockaddr *to, socklen_t tolen);
```

- **sockfd** - זה socket descriptor דרכו שולחים מידע.
- **msg** – מצביע למידע אשר שולחים
- **len** – גודל המידע בביתים
- **flags** – ערך שלו 0
- **to** – מצביע למבנה sockaddr המכיל IP ו-Port של יעד
- **tolen** – גודל המבנה sockaddr.
- הפונקציה מחזירה את כמות הביתים שנשלחה בפועל. כמות היכולה להיות פחות ממה שרצינו לשלוח.
- יש לוודא כמה נשלח בפועל ולהשלים את מה שחסר.
- הפונקציה מחזירה ערך -1 אם יש שגיאה.

אם נרצה **לקבל** מידע מעל UDP חשוב לציין שאין לנו חיבור קודם ולכן נשתמש בפונקציה הבאה:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
        struct sockaddr *from, int *fromlen);
```

- **sockfd** - זה socket descriptor דרכו קוראים מידע
- **buf** – מצביע לבאפר אשר קוראים בתוכו
- **len** – גודל מקסימלי של באפר בביתים
- **flags** – ערך שלו 0
- **from** – מצביע למבנה המכיל IP ו-Port של השולח
- **fromlen** –מצביע למשתנה מקומי אשר יכיל גודל המבנה sockaddr .
- הפונקציה מחזירה את כמות הביתים שהתקבלו.
- אם הפונקציה מחזירה ערך -1 אז יש שגיאה.
- אם פונקציה מחזירה 0 זה משקף כי חיבור נסגר.

סגירת חיבורים וסגירת סוקט

אחרי שסיימתי את השיחה שלי אז אני צריך לעשות לסגור את הסוקט כדי שאוכל לשחרר את הפורט.

```
int close(int sockfd);
```

- sockfd - זה socket file descriptor

אם לא נעשה close זה יאפשר פרצת אבטחה כאשר אני משאיר פורט פתוח לציבור ולכן צריך לסגור.

יש עוד פונקציה שסוגרת הגדרות ספציפיות בסוקט מסוים:

```
int shutdown(int sockfd, int how);
```

- ערך של ארגומנט how קובע אופן סגירת חיבור:
  1. SD_RECEIVE הערך 0 - סגירת חיבור לקבלת מידע.
  2. SD_SEND הערך 1 - סגירת חיבור לשליחת מידע.
  3. SD_BOTH הערך 2 - סגירת שני כיוונים – דומה לפונקציה ()close.
- הפונקציה מחזירה 0 אם הפעולה עברה בהצלחה, אחרת, -1.

Raw Sockets

- מאפשר גישה למידע לפרוטוקולי רשת מלבד TCP ו-UDP כמו למשל ICMP,IGMP .

למשל ICMP זה פרוטוקול בקרה, העבודה שלו זה נטו שליחת הודעה בסיסית, בין היתר האחריות שלו זה לבדוק את הרשת בהודעת ping. עוזר בצורך להבין אם הצד השני קיים.

הפרוטוקולים האלה לא משתמשות בפורטים והם לא נמצאות בשימוש בשכבת התעבורה.

- לממש פרוטוקולים חדשים על IPv4.
- לבנות חבילות משלך (בזהירות).

למה Raw Sockets?

- גישה לדברים אחרים.
- מאפשר ברמת המתכנת אופציות לשחק עם החבילות.
- מאפשר לממש פרוטוקולים חדשים.
- נוכל לבנות ולמלא את ה-IP Header איך שאנחנו רוצים (טוב לבדיקה ושליטה).

מגבלות?

- אין אמינות (רק ביני לבין כרטיס הרשת).
- אין פורטים.
- אין סוג תקשורת מסוים (יכול לעשות מה שבא לי).
- אין ICMP אוטומטי - אני צריך לבנות אותה בעצמי ולהגיד לה מה לעשות.
- מביא לי גישה ישירה ל-raw data (המידע שלא עבר עיבוד עדיין).
- דורש גישה מנהל.

תפעול ה-Raw Sockets כ-ICMP

- אתחול סוקט:

```
socket (PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

- אין לנו ()bind בגלל שאין לנו פורט.
- אין לנו ()connection.

- שולחים מיידית בצורה:

```
sendto (s, buf, strlen(buf), 0, addr, &len);
```

יצירת Raw Socket

- יצירת IP Header משלנו בצורה:

```
setsockopt (s, IPPROTO_IP, IP_HDRINCL, &on, sizeof (on));
```

- יכול לבצע bind and connect.

Principles of Congestion Control

מהו עומס? - הרשת היא עמוסה אם יש יותר מדי תעבורה של מקורות ומידע ברמה שהרשת לא יכולה להתמודד איתה .

התופעה - השהיות ארוכות וחבילות שהולכות לאיבוד בגלל נניח buffer overflow (הכוונה לראוטרים שה-buffer שלהם מלא מדי).

יש לשים לב שלא מדובר כאן ב-Flow Control (כי הרבה יש שולחים לא רוצים להעמיס את הרשת).

מה גורם לעומס?

תרחיש 1

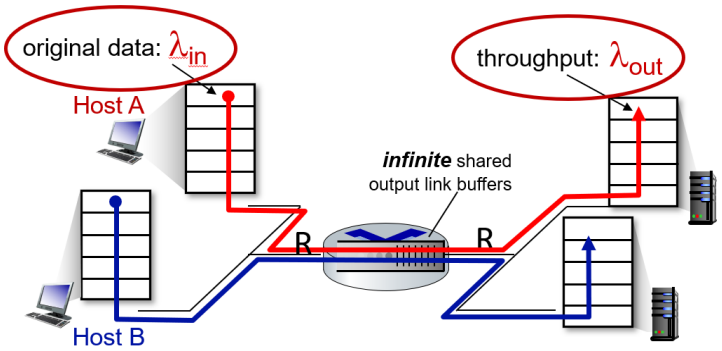
נתבונן באיור משמאל.

יש לנו 2 הוסטים: הוסט A משדר לשרת העליון ויש את הוסט B שמשדר לשרת התחתון.

קיים ראוטר אחד ביניהם שיש לו תור אינסופי.

קצב הכניסה של הנתב = קצב היציאה (R=).

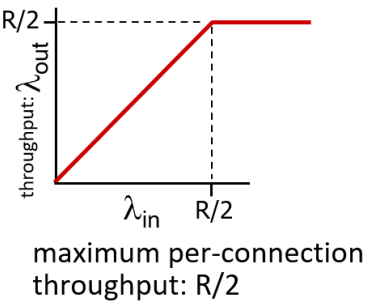
את התכולה המקסימלית הוא R מכל אחד מהכיוונים באיור.



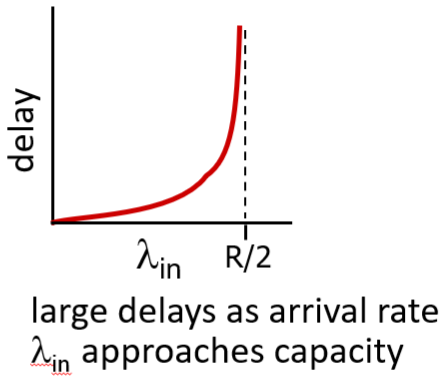
כאשר λ_{in}(כלומר הקצב שבו האפליקציה מייצרת מידע לתעבורה שצריך לשדר) ו- λout הם קצב של מידע שגם קיים בשידור הכחול בציור.

אין צורך בשידורים חוזרים כי ה-buffer (התור) הוא אינסופי (מדובר בתור שמשתף ביחד את שני השידורים). אז מה קורה אם קצב של λ_{in} גדל?

אז במצב האופטימלי וההוגן אם ה-λ_{in} גיעלה אז ה-λout יעלה עד איזשהו $R/2$ כי 2 ה-λ וכל אחד מקבל חצי אז אני יכול לעבור את ה-capacity שלי.



אבל ככל שאנו מגדילים את קצבי התעבורה אנחנו יכולים ליצור תורים בנתב ואז השהייה גדלה מאוד וזה לא מצב אופטימלי כי זה אומר שעד החבילה מגיעה לצד השני זה יקח הרבה מאוד זמן.



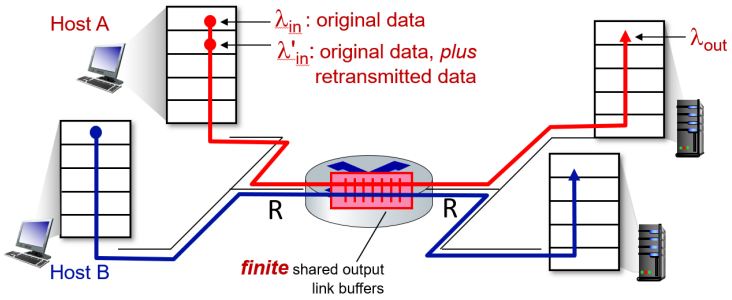
זאת אומרת שזה נחמד להעלות את הקצב אבל באיזה נקודת זמן העלאה הזאת פוגעת בנו.

תרחיש 2

אותו המצב כמו מקודם רק שהפעם השרת עם buffer סופי.

כמובן שייתכן שהשולח צריך לעשות שידורים חוזרים או בגלל timeout ולכן מבחינתנו יש לנו את ה-λשל האפליקציה כלומר $\lambda_{in} = \lambda_{out}$

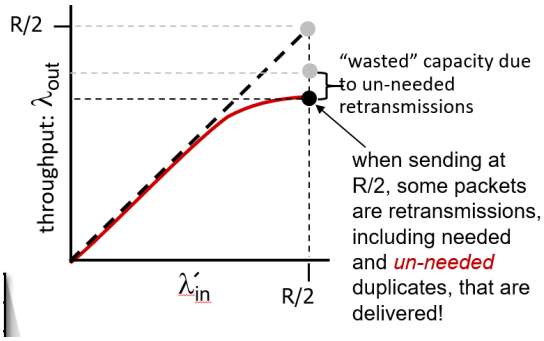
ויש את ה- 'λשל שכבת התעבורה המכיל בתוכו גם את מה שהאפליקציה רוצה שאשדר וגם את השידורים החוזרים.



אין לנו אפשרות לדעת מה המצב של הנתב ומאחר והתור של הנתב סופי, יש לתת תשומת לב למצב התור לפני שנשלח הודעה כלשהי מההוסטים כי אם נשלח הודעה בתור מלא אז ההודעות שלנו ילכו לאיבוד.

ככל שנעלה את ה-'λ שהיא מכילה בתוכה 2 דברים (המידע האמיתי וגם את ה- retransmit) אז יהיה לי בעיה כי ה-output לא מגיע לאן שנרצה וזה כי אני מבזבז חלק מהרוחב פס שלי כדי לשדר הודעות חוזרים וזה פוגע ברוחב הפס.

יותר מזה, חלק מה-capacity הוא בשביל שידורים חוזרים.

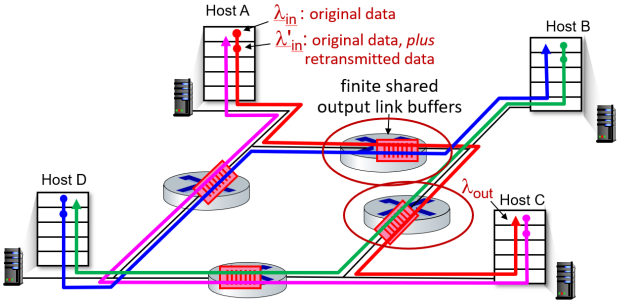


יכול להיות שיבוצע שידור חוזר שלא נדרש באמת כי הטיימר שלי היה קצר מדי.
כלומר, עד שההודעה תגיע לשרת ההוסט ישלח הודעה נוספת כי מבחינתו הוא לא קיבל ACK שההודעה התקבלה בהצלחה בזמן שהוא הקצה ולכן מבחינתו נשלח שידור נוסף של הודעה זו.

אז מה המחיר לעומס?

- שידורים חוזרים.
- שידורים חוזרים שלא נצרכים.
- לינקים שמעבירים מידע שמשוכפל (כלומר מקטינים לנו את ה-throughput ולכן עומס הוא לא טוב בשבילנו).
- שידורים חוזרים כפולים.

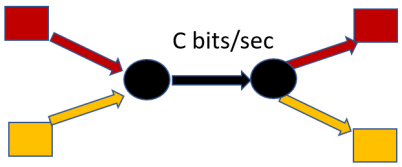
תרחיש 3



כאשר יש לי 4 הוסטים שמדברים ביניהם מכל מיני מסלולים אז אם אצל הוסט מסוים הקצב יעלה עם השידורים החוזרים זה יכול לפגוע בקצב של הוסט אחר.
כלומר שיש כאן עוד מחיר,
אם החבילות שלי נזרקות באיזושהי נקודה אז כל מי שהתאמץ לפני להביא אותי לנתב הזה בזבז את ה-capacity שלו.

לכן התובנות שלנו מתרחישים אלה הם:

- ה-throughput לא יכול לעבור את ה-capacity הכללי.
- השהיות גדולות ככל שאני מגיע ל-capacity (בעיה - תורים יותר גדולים ודיליי).
- שידורים חוזרים בגלל אבדות שמורידים לנו את התפוקה האפקטיבית.
- אם יש לנו שידורים נדרשים שלא נדרשים זה עוד יותר פוגע ברשת.
- אם חבילה נזרקה אז כל מי שהתאמץ להביא אותה לנקודה מסוימת בזבז משאבים.



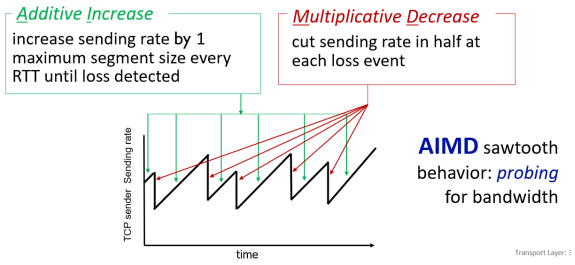
יהיו 2 flows אמינים (אדום וצהוב) עם לינק משותף (שחור) בעובד בקצב C.
הבעיה כאן שאם התורים בנתבים כמעט ריקים אז זה אומר שאני לא מנצל את הרשת אבל אם התורים מלאים אז הדיליי הוא גבוהה וזה גם לא טוב.
נרצה לנצל את הרשת טוב יותר אבל להימנע מעומס ברשת.

מה הייתי רוצה לשפר? -

- לצמצם ככל האפשר את השידורים החוזרים = (Efficiency) Goodput
- איך לחלק את רוחב הפס בצורה ההוגנת ביותר בין כולם = Fairness
- שהחבילות לא ימתינו עד אינסוף = Low delay
- הקצאות מהירות = Fast

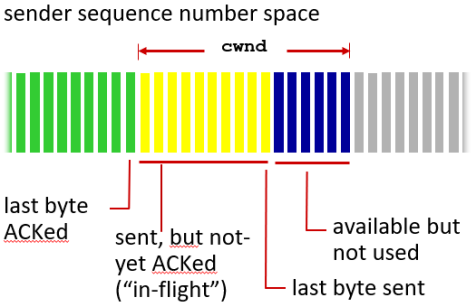
TCP Congestion Control

AIMD

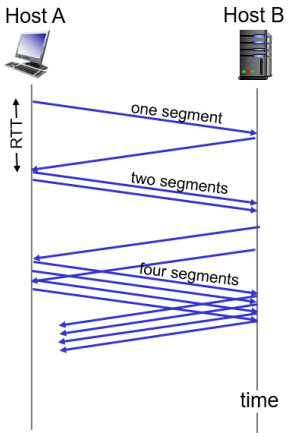


הכוונה שאני מגדיל את הקצב עד שאני רואה שיש עומס
ואז מקטין את הקצב.
עליה לינארית וירידה אקספוננציאלית.

יש לנו חלון ואותו אני מגדיל ומקטין לפי השינוי כלומר $TCP\ rate \approx \frac{cwnd}{RTT}\ (bytes/sec)$
נרצה להתחיל לשדר ואנו יודעים מה גודל חלון הקבלה של הצד השני,לכן נרצה מצד אחד לנצל במקסימום את חלון הקבלה של הצד השני ובמקביל לא להעמיס על הרשת ובנוסף לשדר כמה שיותר.

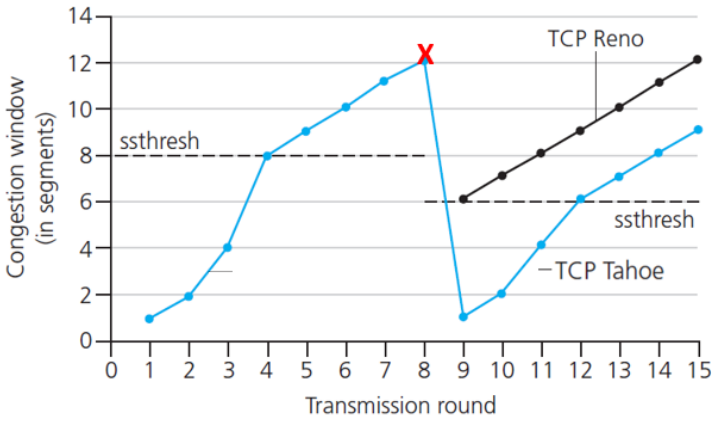


Slow Start



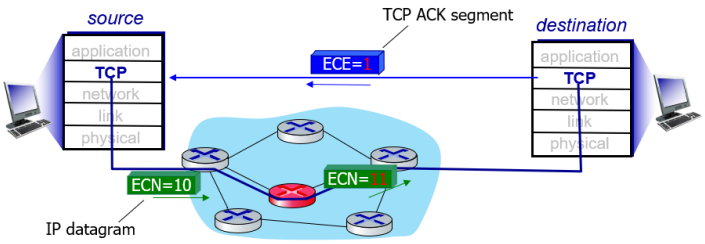
בתחילת השידור אני צריך לעשות תהליך שהוא אקספוננציאלי כלומר כל RTT אני מכפיל את כמות החלונות שלי (איור משמאל).

ברגע שאגיע ל-ssthresh אני אעבור למצב של עליה לינארית.
כאשר אגיע למצב של timeout אני יורד ל-1 אבל ה-ssthresh נחתך בחצי.
אין משמעות יותר ל-ssthresh הקודם אלא יש משמעות ליצור אחד חדש שיורד בחצי מהמצב הקודם וקל לראות את הסיבה בעזרת האיור למטה.



ECN

מאחר והנתבים לא יכולים להגיע ל-header של TCP אז הם מסמנים ב-header של IP, הם משנים ביטים לאלמנט כשיש עומס ואז בצד השני (dest) הוא מעתיק את הסימון מתוך ה-header של ה-IP datagram לתוך ה- TCP ACK segment ואז ה-source מקבל ב-ACK segment ויכול באמצעותו להבין אם אותו segment שאושר עבר באיזשהו נתב שהוא עמוס.



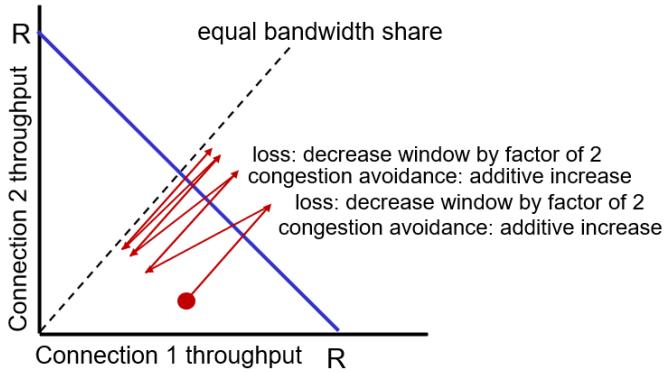
TCP Fairness

אם יש כמה חיבורי TCP על אותו לינק היינו רוצים שהם יתחלקו ביניהם בצורה שווה.

באיור הבא נראה את הקצב של 2 חיבורי TCP , הקו המקווקו זה החלוקה ביניהם.

אם נעלה לינארית אז באיזושהי נקודה אנחנו נעמיס יותר מדי ואז נרד בחצי ככה עד שנשים לב שאנחנו מתכנסים לאלמנט שהרוחב פס כביכול מחולק בצורה שווה בין כולם.

TCP מאפשר לי ליצור הוגנות בין הרבה חיבורים (לעומת UDP).



שכבת הרשת

שירותים ופרוטוקולים

השכבות ב-endsystem לא מדברות בינם לבין עצמם כמו בשכבת התעבורה והאפליקציה.
שכבת הרשת של הנתב מתערב בתהליך בניגוד לשכבות שנלמדו וגם ההוסטים לוקחים חלק.
מצד השולח: עוטף סגמנטים בתוך datagram ועובר לשכבת הקישור (link layer).
מצד המקבל: מקבל את ה-datagram ומוסר סגמנטים לפרוטוקול בשכבת התעבורה.

- פורטים בשכבה זו הם פיזיים בניגוד לשכבות הקודמות שהם וירטואלים.

פונקציות עיקריות בשכבה

- Forwarding : להעביר פאקטה מ-input link ל-output link.
- Routing: להבין מה המסלול בין המקור אל היעד (בעזרת אלגוריתמים).

נעזר ב-waze כדוגמה שתעזור לנו להבין את ההבדל בין 2 הפונקציות האלו:
ההבדל ביניהם הוא שה-routing זה כמו לדעת מה המסלול ממקור ליעד וה-forwarding זה שבכל צומת שנגיע הוא אומר לנו לפנות ימינה או שמאלה.

Data Plane

הוא לוקאלי, זה פונקציה שהראוטר עצמה מבצעת.
הפעולה של להעביר את ה-datagram מ-input port אחד לאחר.

Control Plane

בין הנתבים עצמם, הקונטרול פליין עוזר לו לדבר עם נתבים אחרים כדי להבין מהם המסלולים האפשריים מהוסט להוסט.

- Traditional Routing Algorithms : ממומש בתוך הנתבים וביניהם לעצמם יוצרים מעין פרוטוקולי ניתוב למסלולים האפשריים והקצרים ביותר.
- SDN : יותר מתקדם, לוקחים מכל נתב את ה-control plane שלו ומאחדים את כולם למקום אחד מרכזי ואז יש לנו ראיה מרכזית על הזרימה (יש לנו שליטה נרחבת יותר על הרשת).

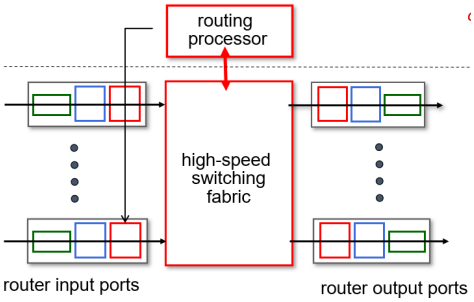
Best Effort Service

האינטרנט עובד בשיטת השירות בשם "**Best Effort**" אשר מתאר רמת שירות שלא מבטיחה שהמסר אכן יגיע אל יעדו, לא מחייבת העברת datagram מוצלחת, לא מחייבת זמן העברה וסדר העברה ולא מבטיחה שהעברת המסר תעמוד בסטנדרטים מסוימים (כגון השהיה ועיכובים).

יתרונות של שירות זה:

- השירות שלו פשוט, הרשת עצומה ומורכבת וכדי לבקש ממנה שינויים זה יכול ליצור בעיות, מנגנון זה מאפשר פריסה נרחבת של האינטרנט.
- אספקה מספקת של רוחב הפס מאפשרת ביצועים של יישומים בזמן אמת (למשל, קול אינטראקטיבי, וידאו) להיות "טובים מספיק" במשך "רוב הזמן".
- (מרכזי נתונים, רשתות הפצת תוכן) המתחברים קרוב לרשתות הלקוחות, מאפשרים לספק שירותים ממקומות רבים. לשמחתנו יש מספיק משאבים כדי לתת לנו את היכולות האלה דרך שכבת הרשת ישנם שיפורים כמו CDN וגם Data Centers שמקרבים הכל לשירות הלקוח וככה אפשר לאזן גם עומסים.
- שירותים "אלסטיים" כמו כמובן עניין ה-TCP עם Congestion Control.

ארכיטקטורת הנתב



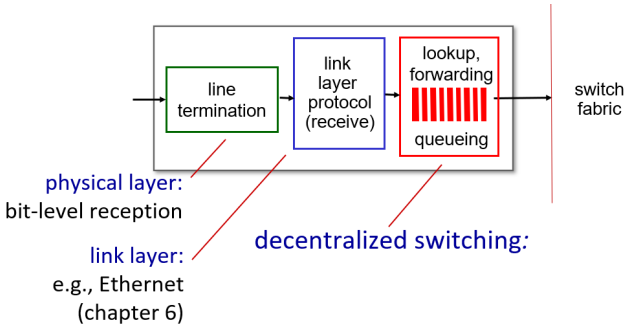
Input port שנכנס אל הנתב ו- Outport port שיוצא מהנתב ויש ביניהם תהליך (switching fabric).

בנוסף יש את ה-routing processor שמקבל הודעות שקשורת לנתב ודואג להחלפת ההודעות בין הנתב הזה לבין שאר הנתבים.

הוא גם בונה את ה-forwarding table שנמצא בכל input port (החץ שיוצא מה-routing processor אל ה-input port בריבוע האדום).

ה-Data Plane באיור זה כל הגופים מתחת לקו המקווקו.

Input port functions



בשלב ב-Decentralized Switching אני אסתכל על איזשהם שדות בתוך ה-header ואני אעשה איזשהו ביצוע ב- lookup כדי לקבל איזשהו match ואז אוכל לעשות כל מיני פעולות.

לדוגמה, אני אקח מתוך ה-header את ה- IP destination, אעשה lookup בטבלת ה-forwarding וברגע שיש match הפעולה שלי תהיה להעביר ל-outport port שכתוב בטבלה.

Forwarding Table

Destination-Based Forwarding

forwarding table

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

כתוב IP נבנה מ-32 ביטים כלומר 4 קבוצות של 8 ביטים כל קבוצה. בטבלה משמאל כל שורת ביטים היא כתובת IP.
טווח כתובות ה-IP שכתוב בשורה הראשונה הוא 2^{11} וזה כי כל הביטים זהים (מימין לשמאל) עד השלב המודגש (ה-10 גם חלק מהזהים).

Longest Prefix Matching

זאת אחת השיטות הכי טובות להתאמה בין כתובות.
אם כל הכתובות משותפות ליציאה אחת (Link Interface) אז למה לא לנסות לחפש ביניהם משהו משותף? - במקום לעשות הרבה בדיקות על מלא תווים כמו בשיטה הקודמת אני יכול לעשות שלוש בדיקות בלבד.
איפה שיש match אני יוצא דרך ה-interface שלו ואם יש לנו כמה matches אז נחפש את ההתאמה עם התחילית הארוכה ביותר.

Destination Address Range	Link interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

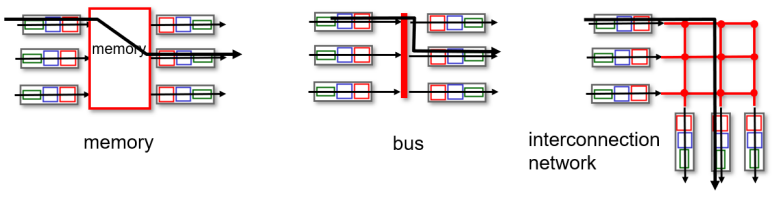
מבחינה לוגית אם הייתי יכול למיין את התחיליות מהגדול לקטן אז ברגע שיש התאמה אחת נצא מיידית מהתחילית הארוכה ביותר ולא נעשה בדיקות נוספות.

Switching Fabrics

מנגנון המעביר חבילות מה-input link אל ה-output link הנכון.

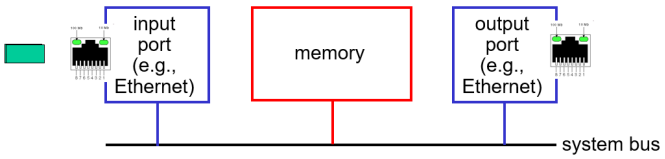
- קצב המיתוג: הקצב שבו אפשר להעביר חבילות מהכניסות אל היציאות.
  לפעמים הקצב נמדד בכמה חיבורי כניסות/יציאות.

ישנם 3 סוגים עיקריים של switching fabrics :



שיטת ה-memory

שיטה שפעלה בדור הראשון של הראוטרים ועובדת תחת שליטה ישירה על המעבד.
החבילות נעתקות למערכת הזיכרון והמהירות מוגבלת על ידי הזיכרון.



שיטת ה-bus

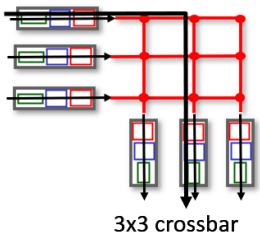
ה-input port מעבירה את ה-datagram ישירות ליציאת הפלט ב-bus משותף, ללא התערבות בעיבוד הראוטר. מכיוון שה-bus משותף מועברת רק חבילות אחת בכל פעם ב-bus. אם ה-bus תפוס החבילות הנכנסות צריכות להמתין בתור. מהירות המיתוג מוגבלת ברוחב הפס של ה-bus.

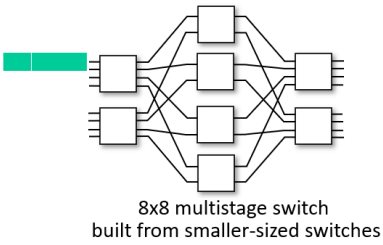
שיטת ה- interconnection network

Crossbar, Clos networks ועוד כמה interconnection nets פותחו בתחילה לחיבור מעבדים במעבדים מרובי-מעבדים.

- Multistage switch: הוא מתג במידה nxn מכמה עמדות של מתגים קטנים.

חבילה המגיעה ל-port input נעה לאורך האפיק המאוזן שמחובר אליו עד שהיא פוגשת אפיק אנכי שמוביל אותה ל-port output הרצוי. רק אם האפיק האנכי לא פנוי, החבילה מחכה בתור ב- port input.





Input Port Queuing

אם ה- switch fabric לא מהיר מדי (קרוב מאוד למהירות הקו) כדי להעביר את כל החבילות המגיעות ללא עיכוב, אז יווצר תור ב-input ports.

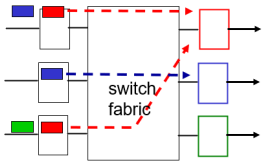
ננית שיש לנו crossbar switching fabric שמקיים את התכונות הבאות:

- כל המהירויות של הלינקים דומות.
- חבילה אחת יכולה לעבור מכל input ports ל-output port כלשהו באותו זמן שלוקח לקבל את החבילה ב- input link.
- ניהול התור הוא FIFO.

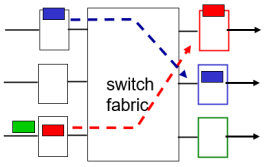
במצב זה, מספר חבילות יכולות לעבור במקביל, כל עוד ה-output ports שלהן שונה.

אם יש שתי חבילות שנמצאות ראשונות (בשני תורים שונים) ורוצות להגיע לאותו תור ב- Output, אז חבילה אחת תיחסם ותחכה. הרי ה- switching fabric יכול להעביר רק חבילה אחת ל-output port ביחידת זמן אחת.

בציור הבא יש דוגמא שבא 2 חבילות (אדומות) נמצאות בראשם של שני תורים ב- input והן מיועדות לעבור לאותו output port . לתופעה הזו קוראים blocking (**HOL**) head of the line.



output port contention: only one red datagram can be transferred. lower red packet is *blocked*



one packet time later: green packet experiences HOL blocking

Output Port Queuing

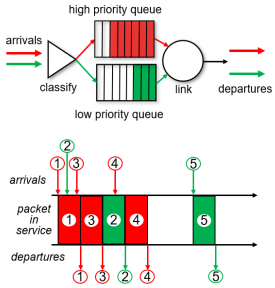
בעיות

- buffering נדרשת כאשר datagrams מגיעות מה-fabric מהר יותר מקצב העברת הקישור.
  ה-buffer מתמלא עד אפס מקום ואז datagrams יכולים ללכת לאיבוד בגלל העומס.
- איזה סדר עדיפות עלינו לשדר?

פתרון

1. **ניהול ה-buffer**
   drop: איזו חבילה להוסיף וזרוק כאשר ה-buffers מלאים.
   אם אין לנו מקום, נעיף את זנב התור.
   עדיפות: זריקה על בסיס עדיפות
2. **סימון**
   נסמן את החבילות בנקודת זמן כלשהי (בערך באמצע תהליך ההעברה).
3. **FIFO**

4. סדר עדיפות



לתת עדיפויות כי אולי יש flow מסוים או חבילות שיותר חשובות מהאחרות. נניח שנרצה להעביר חבילות מסוג קול ווידאו יותר מחבילות מסוג אחר, ולכן נבחר עדיפות מסוימת לסוג חבילות הנכנסות. הבעיה של עדיפות היא שאנחנו יוצרים מצב שאנחנו משרתים את התור העדיף ורק אחרי שאני מסיים את התור הזה נעבור לתור הבא בתור וזה יכול ליצור לנו בעיות כי התור עם העדיפות הנמוכה לא מקבל שירות בזמן ולכן יש פתרון שמעביר חבילה אחת כל פעם מכל תור באופן יחסית הוגן (זה האופציה הבאה).

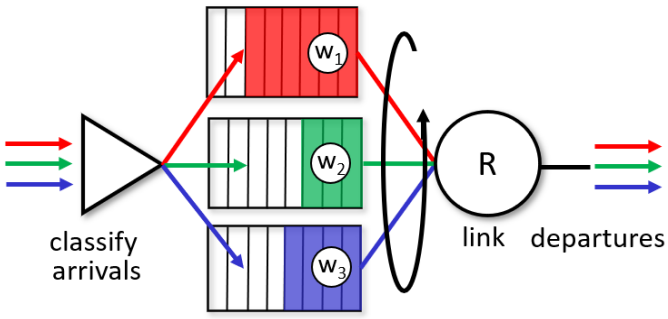
5. Round Robin - RR

נותן פתרון לבעיה באפשרות מספר 4, נעבור כל פעם על חבילה אחת מכל תור באופן יחסית הוגן. זה טוב, אבל החסרון שלו הוא התפעול שלו כי הוא מבטל את העדיפות. ולכן באפשרות הבאה נאזן את עניין העדיפות לבין RR.

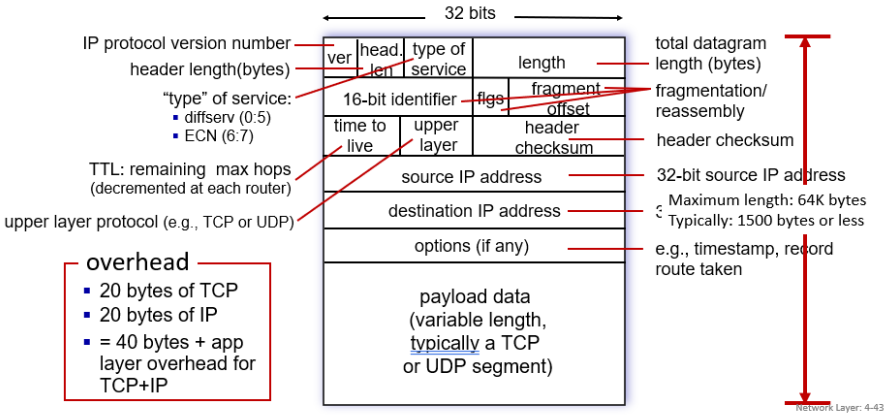
6. משקל הוגן בתור (WFQ)

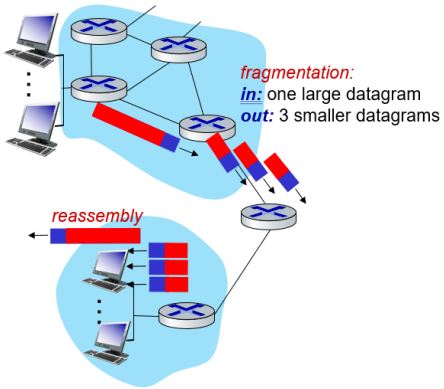
שילוב בין העדיפות לבין ה-"מעבר בין כולם" פשוט נותנים משקל לכל תור (ניתן לתרגם את זה למעין מינימום שירות שכל תור מקבל).

לכל שלב i , יש משקל w_i המקבל את המשקל של כמות השירות בכל סיבוב הביצוע: $\frac{w_i}{\Sigma_j w_j}$



IP Datagram format

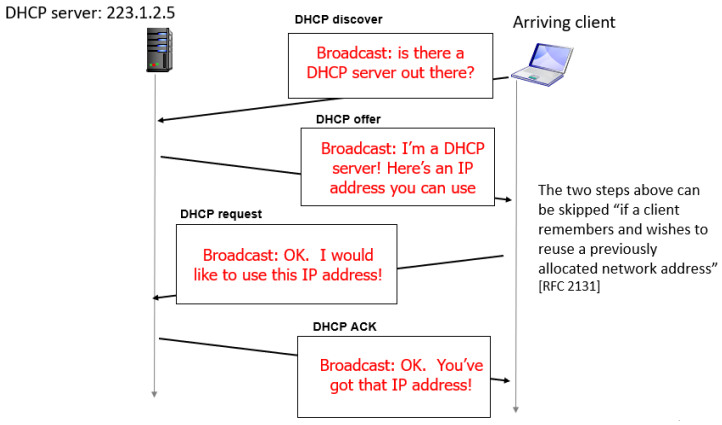




נניח ונרצה לשלוח איזשהו דאטאגרם בגודל מסוים, יכול להיות שבאחד כבלי הרשת בדרך אל היעד יש כבל שלא יכול לספק את הגודל של הדאטה-גרם הזה בגלל שהוא גדול מדי בשביל הכבל.
לכל לינק יש MTU (גודל תעבורה מקסימלי עבור כל רמת לינק).
יש כל מיני סוגים של לינקים ו- MTU השונים זה מזה.
IP-Datagram גדול מתחלק לחלקים בתוך הרשת, כלומר דאטאגרם אחד הופך לכמה דאטאגרמים, והם מורכבים מחדש בחזרה לדאטאגרם אחד כשהם מגיעים אל היעד, ככה אפשר למנוע מצב של חסימה בגלל MTU כלשהו. במציאות שיטת הפתרון הזאת לא אמינה כי יכול להאבד מידע בדרך בגלל יותר מדי חלוקות...

DHCP

איך מארח מקבל כתובת IP? - ההוסט מקבל באופן דינאמי כתובת IP מהשרת כאשר הוא "מצטרף" לרשת.
הוא יכול לחדש את הכתובת האחרונה שהשתמש בה, מאפשר שימוש חוזר בכתובות (אלא אם בוצע התנתקות). ה-DHCP תומך בעיקר למשתמשים בנייד שמתחברים ומתנתקים מהרשת באופן מאוד דינמי.



התהליך של DHCP:

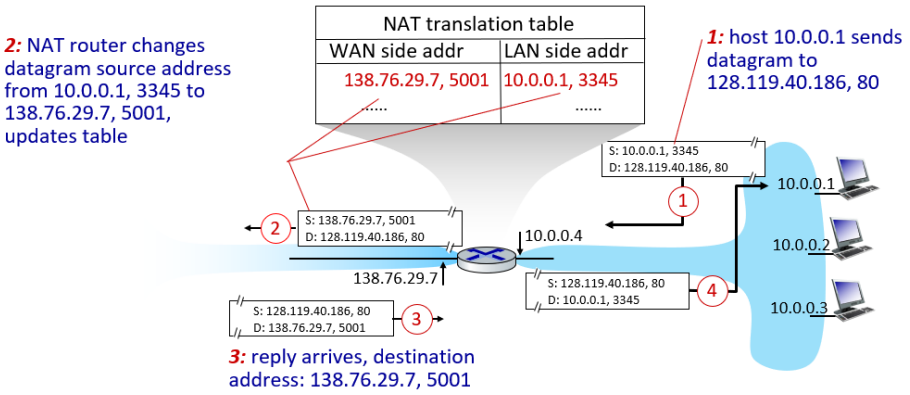
בהתחלה אני מחפש לגלות האם יש לי פה שרת DHCP כלומר DHCP discover.
ברגע שהוא עונה לי בין השורות הוא גם מציע לי איזשהו קונפיגורציה (DHCP offer) ,
אני בוחר אותה כלומר (DHCP request)
ואז הוא מאשר (DHCP ACK) .
ה-DHCP יכול להציע לנו יותר מלהחזיר כתובת IP בתת הרשת.
הוא יכול לספק לנו את השם וכתובת IP של שרת DNS כלשהו.

ICANN

את כתובת ה-IP של הרשת מקבלים מספק השירות שיש לו מרחב כתובות שקנה. ספק שירות יכול לאחד את תתי הרשתות שמתחתיו לכתובת אחת וכל פנייה ללקוח תנותב ישירות מהספק. שיטה זו טובה בכך שהיא מפחיתה את מספר הכתובות שהראוטרים צריכים לזכור.
נניח שאחד מהארגונים שנמצא מאחורי כתובת של ספק שירות מסוים רוצה לעבור לשרת אחר והוא רוצה לשמור על מרחב הכתובות שלו. על מנת לאפשר זאת הראוטר של ספק השירות הישן מוסיף כלל בנוסף לכללים הקיימים, שכל חבילה שנשלחת לכתובת הספציפית הישנה של הארגון תנותב לספק החדש. ספקי השירות מקבלים את מרחב הכתובות שלהם מארגון שנקרא ICANN.

NAT

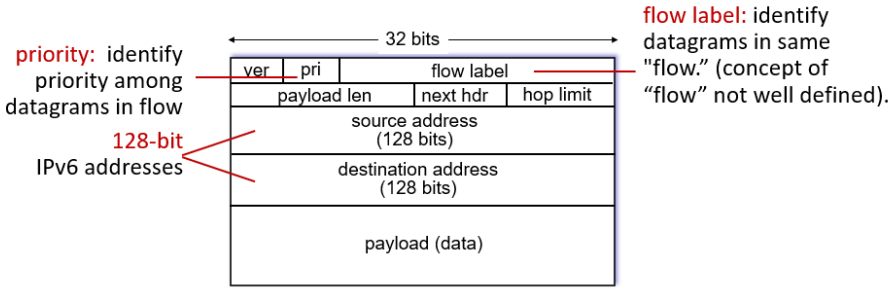
ראוטר NAT ממיר כתובת פנימית לכתובת חיצונית.
הוא גם נותן יתרונות אחרים לדוגמה אנשים מבחוץ לא בקלות יכולים להכנס לרשת הפנימית מבלי שהרשת הפנימית יצרה איתם קשר.ככה הם גם לא יכלו לראות מה המבנה של הרשת הפנימית וכו'
זה נותן לנו גם את האפשרות להבין מי מדבר עם מי בנושאים שאני יוצא מהרשת.
הוא גם יכול להמיר את הפורטים.



IPv6

המוטיבציה הראשונית לפתח גרסה חדשה הייתה בשל העובדה שמרחב הכתובות בנות 32 ביטים הולך ונגמר.
את רוב השינויים בין הגרסאות ניתן לראות בפורמט החבילה:
הגדלת מרחב הכתובות: אורך הכתובת גדל מ-32 ל-128 ביט (לכל גרגר חול בעולם יכולה להיות כתובת IP !!).

- Header קצר יותר המאפשר מעבר מהיר יותר של חבילות.
- Flow Labeling ועדיפויות.



פירוט:

- version -כמובן שכאן הגרסה היא 6.
- Priority שדה זה דומה לשדה TOS שבגרסה 4 .
  מספרים בין 0 ל-7 משמשים למתן עדיפות כשיש תנועה צפופה,מספרים בין 8 ל-15 משמשים למתן עדיפות כשיש תנועה לא צפופה.
- Flow Label שדה זה מיועד לאפיין "flow "של חבילות.
- Payloads Length מספר הבתים בחבילה מעבר לאורך הקבוע שהוא 40 בתים (אורך ה- header).
- Next header מזהה פרוטוקול בשכבה העליונה (TCP או UDP) כמו השדה protocol בגרסה 4.
- Hop limit כמו TTL•
- כתובת המקור והיעד.

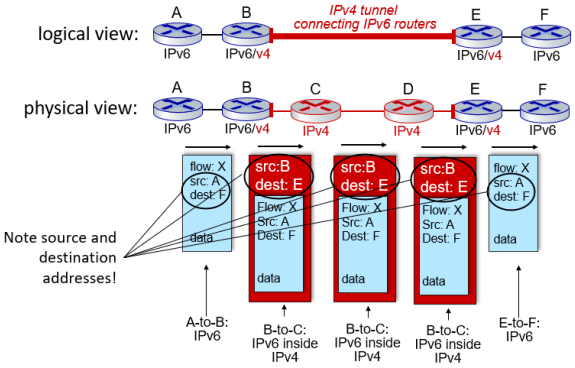
מה אין ב-IPv6?

- checksum
- fragmentation - כלומר אם מגיעה לראוטר חבילה גדולה יותר מהלינק היוצא,
  הראוטר פשוט זורק את החבילה ושולח הודעת שגיאה לשולח.
  במקרה זה השולח יכול לשלוח שוב את המידע בחבילות קטנות יותר.
- Options.

מעבר מ-IPv4 ל-IPv6

איך כל האינטרנט שמבוסס על IPv4 יוחלף ל-IPv6?

הבעיה היא שמערכות עם הגרסה החדשה יכולות לתקשר עם מערכות מהגרסה הישנה, אולם ההפך זה בלתי אפשרי.



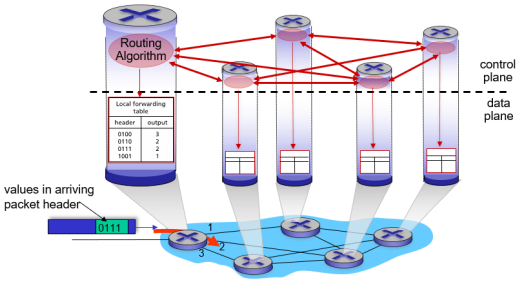
הפתרון:Tunneling.

נניח ששני קודקודים IPv6 רוצים להעביר ביניהם חבילות IPv6 יש ביניהם מספר קודקודים של IPv4 (לקודקודים אלו קוראים tunnel).

לפי שיטת ה-tunnetling, קודקוד B שזה קודקוד IPv6 שנמצא בצד אחד של המנהרה, לוקח את כל החבילה שלו ושם אותה בשדה data של חבילה IPv4.כתובת היעד של חבילה זו היא הכתובת של קודקוד IPv6 שנמצא בצד השני של המנהרה (קודקוד E).אז החבילה עוברת בין קודקודים IPv4 מבלי שידעו בכלל שהם מכילים את כל תוכן החבילה מגרסה 6.כשהחבילה מגיעה לקודקוד E הוא מחלץ את תוכן החבילה שאותה צריך להמשיך להעביר ושולח אותה כחבילה IPv6 .

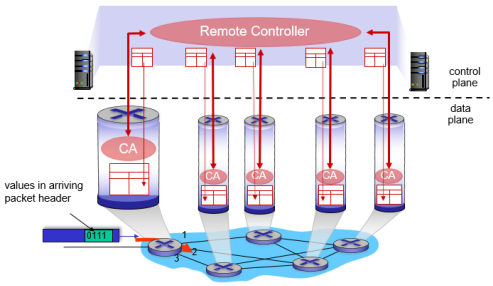
עד עכשיו עברנו על הפונקציה Data Plane וכעת נעבור ללמידה על הפונקציה השניה בשכבת הרשת שנקראת ה-Control Plane.

ישנם שתי גישות לבניית Control Plane ברשת:



Per-router control plane

מערכת מבוזרת, הנתבים ביניהם מחליפים הודעות תחת פרוטוקול routing,
מתוך התוצר של הפרוטוקול הזה הם מעדכנים את טבלאות ה-forwarding שלהם.

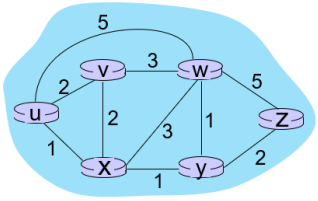


SDN control plane

מערכת מרכזית, אני לוקח את לנתב כביכול את ה-control plane ומרכז במקום אחד ושם מחליט על הטבלאות forwarding ואז אני מוריד את זה לנתבים.

התחילו את הרשת באלמנט מבוזר והרשת הולכת כרגע לאלמנט של ריכוז כמו ה-SDN שמאפשר לנו יותר יכולות מאשר הניתוב ה-per router.

Routing-Protocols



אנחנו רוצים להגיע למצב שאנו רוצים להבין מהו מסלול טוב.
מסלול זה אוסף של נתבים שאני יכול להעביר את החבילות מההתחלה לסוף.מסלול טוב זה אומר מבחינת מחיר, מהירות ופחות עומס.
ניתן לחשוב על זה כגרף.

מבחינת אפשרויות היינו רוצים למצוא איזשהו אלגוריתם שהוא:

- מתמודד מהר עם שינויים או איטי.
  נראה "***distance vector***" שהוא בעזרת אלגוריתם בלמן-פורד לגבי זה.
- בעל הסתכלות גלובלית כלומר שהוא רואה את כל הרשת בנתב או לא גלובלית כלומר שיכול להתמודד עם ידע מהשכנים שלו בלבד.
  נראה "***link state***" המבוסס על אלגוריתם דייקסטרה.

DIJKSTRA Algorithm

כדי להתחיל את האלגוריתם אני צריך להבין ברשת שלי את כל הטופולוגיה ורק אחרי זה אני מוצא את המסלול ה-"זול" ביותר מהמקור אל השאר.
ברגע שאחד הקודקודים רואה את כולם הוא יכול להריץ את המסלולים הזולים ואז לבנות לעצמו את ה-forwarding table.
כמובן שברגע שיש עדכון כלשהו, (משקל שהשתנה) אז צריך לעדכן את כולם ואז להריץ עוד פעם את הכל עם דייקסטרה.

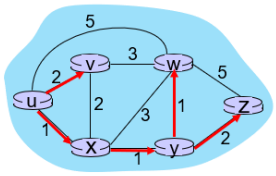
נסמן

- $C_{X,Y}$עלות המשקל בין 2 קודקודים שכנים X ל-Y. הוא יהיה ∞אם אין שכנות.
- $D(v)$ אומדן שוטף של עלות נתיב בעלות נמוכה ביותר ממקור ליעד v.
- $p(v)$צומת קודמת לאורך הדרך ממקור ל- v.
- N'קבוצה של צמתים שהדרך הנמוכה ביותר שלהם ידועה.

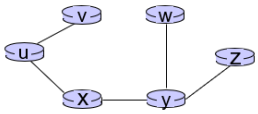
```
Initialization:
N' = {u}
for all nodes v
  if v adjacent to u
    then D(v) = C_{u,v}
  else D(v) = ∞
```

Loop:
find w not in N' such that D(w) is a minimum
add w to N'
update D(v) for all v adjacent to w and not in N' :
$D(v) = min(D(v), D(w) + C_{w,v})$
until all nodes in N'

דוגמה



resulting least-cost-path tree from u:



resulting forwarding table in u:

destination	outgoing link
v	(u,v)
x	(u,x)
y	(u,x)
w	(u,x)
x	(u,x)

route from *u* to *v* directly

route from u to all other destinations via *x*

סיבוכיות האלגוריתם

ישנם N קודקודים. לכל אחד מה-N איטרציות צריך לבדוק את כל הקודקודים.

לכן מבחינת מספר ההשוואות נקבל $O(n^2)$.

אך ישנו מימוש הרבה יותר יעיל המבצע $O(n \cdot logn)$.

הסיבוכיות של ההודעה

אחד הבעיות הכי קשות שלו זה העניין שכל נתב צריך לעשות broadcast על הלינקים שלו לכל שאר הראוטרים. כלומר צריכים ליצור לעצמנו איזשהו תהליך/אלגוריתם שבו אני מספר על המצב שלי לכולם וכולם מספרים לכל השאר כדי שנוכל בכלל להתחיל (כי ההתחלה היא שאני יודע מהי הטופולוגיה). לכן הסיבוכיות היא $O(n^2)$.

Bellman-Ford Algorithm

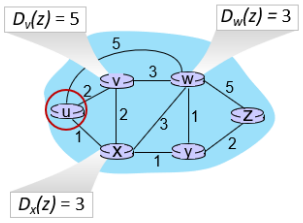
יהא $D_x(y)$: עלות המינימלית עבור מסלול מ-x ל-y.

אזי $D_x(y) = min_v\{C_{x,v} + D_v(y)\}$ כאשר $C_{x,v}$ זה העלות של השכן v של x .

בשונה מאלגוריתם דייקסטרה, אין לו ראייה טופולוגית כלומר הראייה שלו היא רק עבור שכניו כך שאם נרצה להעביר הודעה מ-x ל-y נראה זאת כמו "טיפה מתנפצת על שפת המים" כלומר x יעביר לשכן שלו u ו-u יעביר הלאה לשכניו עד שנגיע ל-y.

בסופו של דבר $D_x(y)$ מחזיר לנו את המסלול הזול ביותר בין x ל-y, אז לא את המסלול עצמו, כלומר בעזרת אלגוריתם זה x לעולם לא ידע את המסלול אל y אלא רק את המשקל המינימלי ביותר המצטבר ביניהם.

Suppose that *u*'s neighboring nodes, *x,v,w,* know that for destination *z*:



Bellman-Ford equation says:

$D_u(z) = \min \{ c_{u,v} + D_v(z),$
$c_{u,x} + D_x(z),$
$c_{u,w} + D_w(z) \}$
$= \min \{2 + 5,$
$1 + 3,$
$5 + 3\} = 4$

node achieving minimum (x) is next hop on estimated least-cost path to destination (z)

Distance Vector Algorithm

אלגוריתם זה מתבסס על אלגוריתם בלמן-פורד.
כל קודקוד שולח את המרחק הוקטורי שלו אל שכניו.
כאשר x מקבל v מרחק חדש זה, הוא מעדכן את המרחק שלו בעזרת אלגוריתם בלמן-פורד, וכך גם שכניו וכן הלאה...
אלגוריתם זה עובד באופן איטרטיבי ובו יש תהליך קבוע ומוסכם עבורו כל קודקוד מחכה להודעת שינוי מרחק משכניו, ברגע שהוא מקבל הודעה זאת, הוא מחשב מחדש את המרחק שלו משכניו ואז מעדכן לשאר היעדים שלו וכך גם הם יחשבו מחדש את מרחקם וכן הלאה...

אז מה ההבדלים בין Link State ל-Distance Vector?
נזכיר כי *Link State* מתבסס על אלגוריתם Dijkstra.

סיבוכיות זמן ההודעה (עדכון) -

- עבור LS הסיבוכיות היא $O(n^2)$.
- עבור DV זמן ההתכנסות משתנה.

מהירות ההתכנסות -

- עבור LS : האלגוריתם הוא $O(n^2)$וגם ההודעות $O(n^2)$, יכול להיות שינויים בעמדות.
- עבור DV: זמן ההתכנסות משתנה, יכול להיות לנו סיבובים אינסופיים.

מה קורה כאשר הראוטר מתקלקל? -

- עבור LS: ראוטר לפרסם עלות קישור שגויה, כל נתב מחשב רק את הטבלה שלו.
- עבור DV: יכול לפרסם עלות שגויה של נתיב ("יש לי נתיב בעלות נמוכה באמת לכל מקום"). בנוסף, כל טבלה של ראוטר מנוצלת ע"י ראוטרים אחרים: התפשטות שגיאות דרך הרשת.

Autonomous Systems (AS)

נתבים המצטברים לאזורים מכונים AS כלומר Autonomous Systems.

ישנם 2 סוגים של AS:

intra-AS: ניתובים בתוך אותו AS (רשת).

- כל הנתבים ב-AS חייבים להריץ אותו פרוטוקול ניתוב.
- נתבים מ-AS אחרים יכולים לרוץ בפרוטוקולי ניתוב אחרים.
- gateway router: הצלע המחברת בין ה-AS לבין AS-ים אחרים.

inter-AS: ניתובים בין AS-ים.

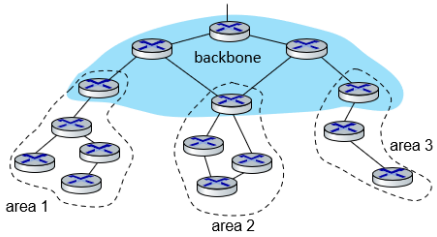
פרוטוקולי ניתוב הנפוצים ביותר של **intra-AS** הם:

- **RIP**: Routing Information Protocol , מבצע שינוי DVs בכל 30 שניות, פחות שמיש.
- **EIGRP**: Enhanced Interior Gateway Routing Protocol המתבסס על DV.
- **OSPF**: Open Shortest Path First הוא משתמש באלגוריתם ניתוב של LS ונכנס לקבוצת פרוטוקולי השער הפנימיים, הפועלים בתוך מערכת אוטונומית (AS) אחת.
  כל נתב מציף הודעת OSPF LS ישירות מה-IP לכל שאר הנתבים בכל ה-AS.
  לכל נתב יש טופולוגיה מלאה ולכן משתמש ב-Dijkstra כדי ליצור forwarding-table.
  בנוסף, כל הודעות ה- OSPF מאומתות (למניעת חדירה זדונית).

היררכיית ה-OSPF

יש 2 שלבים של היררכיות: האזור המקומי וה-backbone.

- הודעת ה-LS מוצפת אך ורק באיזור מקומי כלשהו, או ב-backbone.
- כל נתב בעל ראייה טופולוגית על האיזור שבו הוא נמצא, הוא יודע לגשת רק ליעדים מוגדרים באזור זה.



BGP : Border Gateway Protocol

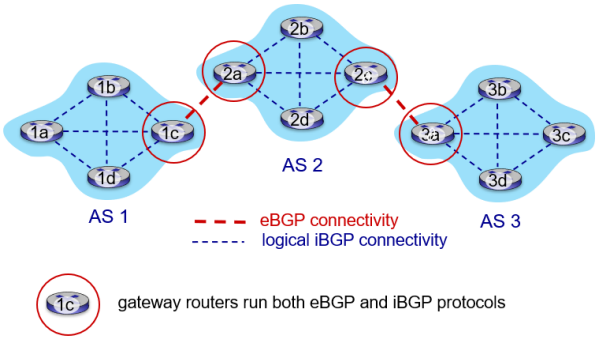
פרוטוקול הניתוב של ה-inter-AS הוא Border Gateway Protocol : **BGP**.

הוא ה-"דבק" שמחזיק את כל האינטרנט ביחד.

הוא מאפשר ל-subnets להודיע על קיומם ולידעים שאליהם הם יכולים להגיע לשאר האינטרנט.

פרוטוקול ה-BGP מספק לכל AS התייחסות ל eBGP ו-iBGP.

- **eBGP**: משיג מידע על הנגישות של ה-subnet מ- ASes שכנים.



- **iBGP**: להפיץ מידע נגיש לכל הנתבים הפנימיים של AS. לקבוע מסלולים "טובים" לרשתות אחרות על בסיס מידע ומדיניות נגישות (**Policy**).

ניתוב ה-BGP בעל 2 תכונות חשובות:

- **AS-PATH**: רשימה של AS-ים דרכה עברה הודעת ה-prefix.
- **NEXT-HOP**: מציין נתב פנימי AS ספציפי ל- AS הבא-הופ.

ניתוב מבוסס על מדיניות

gateway מקבל הודעת ניתוב שמשתמשת במדיניות מסוימת כדי לדעת אם לקבל את המסלול או לא.
מדיניות AS קובע אם להודיע על נתיב ל-AS-ים שכנים אחרים.

שני נתבי BGP ("עמיתים") מחליפים הודעות BGP באמצעות חיבור **TCP** קבוע למחצה.

הודעות BGP

- **OPEN -** פותח חיבור TCP לעמית BGP מרוחק ומאמת שליחת עמית BGP כלומר peer.
- **UPDATE** - מפרסם מסלול חדש (או מסלול ישן אם צריך).
- **KEEPALIVE** - משאיר את ההתחברות פתוחה למקרה של עדכונים ו-ACKs OPEN.
- **NOTIFICATION** - מדווח על שגיאות בהודעות קודמות, גם מיועדת כדי לסגור התחברות.

Hot Potato Routing - בוחר gateway מקומי בעל העלות הנמוכה ביותר עבור intra-domain.

ICMP: Internet Control Message Protocol

ICMP בשימוש על-ידי hosts ונתבים כדי להעביר מידע ברמת הרשת.

- בעיקר בשימוש כדי לדווח על שגיאות כמו מארח שלא ניתן להגיע אליו וכו'..
- יכול לדעת מי הנתבים בדרך עד ליעד.
- משתמש ב-ping לבקשות ותשובות.
- הודעות ICMP מחזיקה איתה IP Datagrams.

חלאס.