**מושגים בסיסיים:**

* **מחשב** : נשתמש בשם זה לישות כללית המחוברת לרשת האינטרנט (יכול להיות טלפון חכם, טבלט, שרת, ראוטר, ועוד..)
* **לקוח** : המחשב ממנו יוצאת בקשה לקבלת מידע משרת
* **שרת**: מחשב המקבל בקשות של לקוחות ומשרת אותן, כלומר מחזיר את המידע המבוקש.
* **World Wide Web** : רשת האינטרנט המורכבת מאינספור דפי אינטרנט המאוחסנים על שרתים. המידע של דפי האינטרנט עובר בין לקוחות ושרתים באמצעות מארג רשתי עצום של ראוטרים ומחשבים.
* **פרוטוקול** : שפה בה מתקשרים באינטרנט על מנת שכל משתמשי האינטרנט יוכלו להבין את המידע העובר ברשת.

בעולם האינטרנט נהוג לחלק את המידע המועבר בפרוטוקול לשני חלקים:

1. Data – תוכן המידע שאנו רוצים להעביר, למשל תוכן המכתב או תוכן השיחה
2. Header – מידע עוטף הבנוי מאוסף של שדות, למשל למי מיועד המידע, מי שלח אותו ומה אורכו.

פרוטוקולים שונים יתנו מענה ליצירת התקשורת ברמות שונות. כדי לאחד את המענים הללו למודל יחיד, פותח מודל עבודה אשר מחלק את רשת האינטרנט לשכבות או למשימות שונות, עבור כל שכבה ישנם פרוטוקולים המגדירים את אופן העבודה בשכבה.

כל שכבה מקבלת שירותים מהשכבה שמתחתיה ונותנת שירות לשכבה שמעליה. השיטה של חלוקה לשכבות מאפשרת ליצרנים לפתח ציוד תקשורת עבור שכבה מסוימת ולדעת שהציוד להם יתחבר או יתקשר היטב עם ציוד של יצרנים אחרים שעובדים על שכבות אחרות.

אחד המודלים המקובלים באינטרנט הוא **מודל חמש השכבות**, המכיל את השכבות הבאות:

1. השכבה הפיזית
2. שכבת הקו
3. שכבת הרשת
4. שכבת התעבורה
5. שכבת האפליקציה

**שכבת האפליקציה Application Layer :**

כל מידע שמגיע מהרשת עובר מהאפליקציה של השרת לאפליקציה של הלקוח. ואילו אינטראקציה שהלקוח עושה עם המידע הזה (לייק, העלאת תמונה וכו') עובר מהלקוח לאפליקציה של השרת.

**שכבת התעבורה :**

תפקידה של שכבת התעבורה הוא לבצע חיבור לוגי בין אפליקציות במחשבים שונים, במטרה להעביר את המידע לא רק למחשב הנכון אלא גם לאפליקציה הנכונה. שכבת התעבורה "מדוורת" את המידע לכל אפליקציה במחשב, המידע עובר לאפליקציה הספציפית באמצעות מספרי פורט (port) שונים.

מידע עובר ברשת בחבילות קטנות שנקראות פקטות, העוברות במסלולים שונים ובזמנים שונים ממכשיר למכשיר. שכבת התעבורה מטפלת במידע המגיע משירותים שונים ומיועד לאפליקציות שונות, וגם יכולה (אך לא חייבת) לשמור על הסדר והאמינות שלו.

שכבת התעבורה יכולה לבצע תפקיד נוסף, והוא לוודא שהמידע שנשלח לשרת בקצה השני של העולם יגיע ליעד שלו. השכבה מתבססת על השכבה שמתחתיה, שכבת הרשת, שיודעת להעביר חבילות מידע ברחבי הרשת. שכבת התעבורה לוקחת את המידע שמגיע משכבת האפליקציה ומוסיפה לו שדות שונים, למשל הוספה של מספר הודעה תאפשר לצד שמקבל את חבילות המידע משכבת הרשת לוודא שהן מגיעות לפי הסדר הנכון ואף אחת אינה חסרה.

קיימים שני פרוטוקולים נפוצים של שכבת התעבורה שאיתם נעבוד בקורס זה:

**פרוטוקול TCP – העברה אמינה של נתונים:**

כפי שאמרנו שכבת הרשת שמתחת לשכבת התעבורה מעבירה חבילות מידע ברשת, אך חלקן לא יגיעו ליעדן, יגיעו לא לפי הסדר שנשלחו או יגיעו משובשות, כיצד נוכל לוודא שהודעה הגיעה בשלמותה ולפי הסדר כך שהמשתמש יבין אותה?

לשמחתנו קיים פרוטוקול TCP, או Transmission Control Protocol. זהו פרוטוקול של שכבת התעבורה המשמש להעברה אמינה של מידע. פרוטוקול אמין מבטיח שכל המידע של האפליקציה יגיע בצורה מלאה ומדויקת, ללא אובדן או שיבוש. יש לו שני מנגנונים שמאפשרים העברה אמינה של נתונים:

1. מספר סידורי Sequence Number – כל פקטה מקבלת מספר סידורי המאפשר למקבל המידע לסדר אותו בסדר הנכון ולבדוק אם יש מידע שהושמט בדרך
2. מנגנון אישור Acknowledgment – הצד שמקבל את המידע משתמש במנגנון אישור על מנת להודיע לצד השולח שהמידע שהגיע אליו הגיע באופן תקין. אם השולח אינו מקבל אישור על העברה תקינה, הוא ישלח את המידע שוב ושוב עד שיתקבל אישור.

כל אפליקציה שחייבת שהמידע שהיא מעבירה יעבור בצורה אמינה חייבת לעבוד בפרוטוקול TCP!

**פרוטוקול UDP – העברה לא אמינה (ומהירה) של נתונים:**

החסרון של פרוטוקול אמין הוא מהירות העברת המידע. בפרוטוקול אמין הצד השולח והצד המקבל צריכים לבצע פעולות נוספות (סידור הפקטום, בדיקה, שליחה חוזרת וכו'..) מה שמאט את התהליך.

אם לא חייב שהמידע יגיע בצורה מדויקת (נניח שיחת וידאו, משחקי וידאו, שיחות קוליות) כדאי להעביר את המידע מהר יותר ולבצע שידור streaming (שידור בזמן אמת) בו אם נאבד חלק מהמידע, כמו אות או מילה פה ושם, לרוב לא זה לא יפגום בהבנת השיחה במלואה.

פרוטוקול UDP (User Datagram Protocol) הוא פרוטוקול של שכבת התעבורה והוא דומה לפרוטוקול שאינו אמין. כאשר שולחים מידע באמצעות פרוטוקול UDP, בנוסף למידע של שכבת האפליקציה נוספים רק עוד 4 שדות. החסכון בשדות ביחד לTDP והעובדה שלא מתבצע מאמץ כדי לשלוח מחדש את הפקטות שלא הגיעו, מאפשרים לUDP להעביר מידע במהירות רבה יותר לעומת TCP האמין.

**סוקט**

צינור תקשורת וירטואלי שבו שני פתחים, פתח אחד בצד השרת ופתח אחד בצד הלקוח, דרכו הם מתקשרים כשהם נמצאים על מחשבים מרוחקים או על אותו מחשב.

כדי ליצור סוקט אנו צריכים לתת לו כתובת מדויקת על מנת שההודעות הממוענות אל השרת או אל הלקוח ידעו לאן להגיע. החלק הראשון המשמש לזיהוי הוא כתובת הIP של המחשב. הפרט השני הוא מספר הפורט בסוקט. כמו כן, כל הודעה שנשלחת, מועברת גם עם כתובת הIP והפורט של המקור ממנה היא הגיעה, ככה יהיה ניתן להחזיר תגובה.

כשהמחשב מתחבר לאינטרנט, הוא בדרך כלל מקבל כתובת IP באמצעות פרוטוקול סטנדרטי. בכל פעם שמתכנת כותב שרת הוא מגדיר את מספר הפורט שאליו יפנו הלקוחות. כדי שהלקוחות ידעו לאיזה מספר פורט לפנות, צריך פעולה שמתאמת בין מתכנתי השרתים למתכנתי הלקוחות. בהרבה מקרים ניתן לחסוך פעולה זו בעזרת כמה מספרי פורטים מפורסמים שנמצאים בטווח שבין 0-1023 ושמורים לפרוטוקולים חשובים ושכיחים. הפורטים האלה נקראים בשפה המקצועית well known ports. פורט מספר 80 לדוגמה הוא מפורסם במיוחד והוא משמש לגלישת אינטרנט בפרוטוקול HTTP.

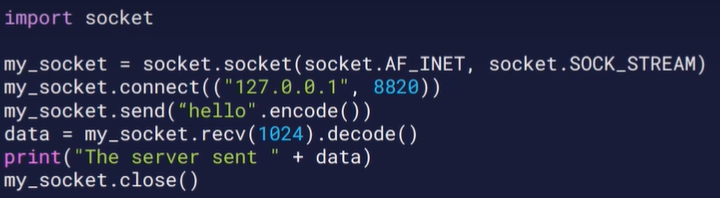
בנוסף, לכל סוקט יש גודל, לפיו נדע כמה מידע יכול לעבור בצינור ברגע נתון. ה"קוטר" של הצינור שלנו הוא בית (byte) אחד, כלומר 8 ביטים (bits). בשיטת הייצוג של תווים טקסטואליים שבה נשתמש, ASCII, גודלו של כל תו טקסט הוא בית אחד.

השרת והלקוח מחזיקים ביניהם בשני קצוות של ערוץ התקשורת, כל נקודת קצה היא סוקט, שיש לו שני מזהים – כתובת IP ומספר פורט.

הצד השולח יעביר את ההודעה אל כתובת הIP של הידע ואל הפורט של היעד, ועם ההודעה יעביר גם את כתובת הIP והפורט שלו, זאת על מנת שיוכל לקבל תגובה.

השרת מאזין לסוקט ספציפי, ונותן מענה רק לפניות המיועדות לIP שלו ולפורט אליו הוא מאזין. שולחים הודעות, מקבלים תגובות ובסוף סוגרים את הסוקט. אנחנו יכולים לשלוח כמה הודעות שנרצה.

איך לממש סוקט? (זהו סוקט צד לקוח, צד שרת יובא בהמשך)

* נפתח קובץ פייתון חדש ונשמור אותו (חשוב לא לשמור את הקובץ בשם של ספריה קיימת בפיתון, כמו במקרה זה בשם socket.py)
* נייבא את הספרייה על ידי import socket
* ניצור אובייקט מסוג socket ונקרא לו my\_socket, הארגומנטים המועברים אליו הם:
  + socket.AF\_INTER שמגדיר שאנו משתמשים בפרוטוקול IP ולכן הוא יצפה לקשר בין שתי כתובות IP
  + Socket.SOCK\_STREAM מגדיר שימוש בפרוטוקול TCP
* נחבר את הסוקט שייצרנו לשרת על ידי שימוש במתודה connect שמקבלת כתובת IP ומספר פורט. הכתובת היא כתובת הבית של השרת ולכן היא נקראת local host, אם השרת שאנו מנסים להתחבר אליו מרוחק, נעביר למתודה את כתובת הIP של השרת המרוחק.
* כדי לשלוח מידע לשרת, נשתמש במתודה בשם send ובגלל שמה שאנו שולחים הוא רצף של מספרים בינאריים, נשתמש במתודה encode לקודד את המחרוזת שאנו רוצים לשלוח.
* כדי לקבל את התגובה שתשלח מהשרת, נשתמש במתודה recv על ידי שמירה של המידע המוחזר בתוך משתנה בדם data שאותו נדפיס למסך. למתודה recv אנו מעבירים אלמנט מספרי שאומר מהו מספר הבתים המקסימלי שאנו מבקשים להוציא מהסוקט. אם יש פחות בתים זה גם בסדר.
* בשלב האחרון, נשתמש במתודה decode כדי לתרגם את רצף הבתים הבינארי שמתקבל מהשרת למחרוזת שאפשר להבין.
* בסיום, לאחר ששלחנו וקיבלנו את כל מה שרצינו, נסגור את הסוקט על ידי שימוש במתודה close.

כדי לבדוק שמה שכתבנו עובד טוב, ניתן להשתמש בשרת הדים Echo Server, שמחזיר ללקוח בדיוק את המידע שהוא שלח.

כעת נעבור לכתוב את צד השרת. אנו נכתוב שרת שמהדהד את ההודעה ללקוח:

* נייבא את הספרייה socket
* ניצור אובייקט מסוג socket ונקרא לו server\_socket, הארגומנטים המועברים אליו הם:
  + socket.AF\_INTER שמגדיר שאנו משתמשים בפרוטוקול IP ולכן הוא יצפה לקשר בין שתי כתובות IP
  + Socket.SOCK\_STREAM מגדיר שימוש בפרוטוקול TCP
* נבצע קישור של הסוקט שיצרנו לכתובת מקומית, על ידי שימוש במתודה bind. מתודה זו דומה למתודה connect שפגשנו קודם, והיא גם מקבלת כתובת IP ומספר פורט, אבל ההבדל הוא שהיא משמשת את השרת כדי לקבוע לאיזה צירוף כתובת IP ופורט הוא יאזין, בעוד שconnect משמשת את הלקוח כדי לבחור לאיזה צירוף של IP ופורט להתחבר. אנו נשתמש בכתובת IP 0.0.0.0 שאומרת "האזינו לכל מי שפונה לכתובת הIP החיצונית שלכם, או לכל מי שפונה מתוך המחשב אל הLocal Host.
* יצירת הקישור לבדה אינה מספיקה כדי שנוכל לקבל חיבורים מלקוחות, לכן נשתמש במתודה listen. ואז אנו מחכים לחיבורים נכנסים.
* בקשות ההתחברות של הלקוחות מחכות בתור לאישורן, רק לאחר שהשרת יאשר הבקשה יתבצע חיבור. כדי לאשר חיבור, משתמשים במתודה accept שמשמעותה blocking, כלומר הרצת הקוד "תקפא" והתוכנית לא תמשיך לרוץ עד אשר יתקבל בשרת חיבור חדש.

המתודה מחזירה שני משתנים:

* + Client\_socket – אובייקט סוקט המכיל את כל המידע שהשרת צריך בשביל לפנות אל הלקוח. כיוון שהוא אובייקט, הוא גם יאפשר לנו להשתמש בכל המתודות של סוקט שנזדקק להן.
  + Client\_address – טאפל המכיל את כתובת הIP ומספר הפורט של הלקוח שהתחבר.
* כעת ניתן לקבל מידע מלקוח, נשתמש במתודה בשם recv כמו קודם, עבור הסוקט שהתקבל, ונשתמש במתודה decode על מנת לתרגם מה הלקוח שלח. ואז נהדנד את המידע חזרה ללקוח על ידי שליחה חוזרת שלו ללקוח על ידי send וencode.
* כשנסיים את התקשורת עם הלקוח, עלינו לסגור את החיבור איתו. עומדות בפנינו שתי אפשרויות:
  + לתת שירות ללקוח אחר – כדי לעשות זאת נצטרך להשאיר את סוקט השרת ולחזור למצב האזנה לפניות של לקוחות
  + לסיים את פעולת השרת ולסגור את סוקט השרת, כך שהוא לא יקבל פניות של לקוחות נוספים

נקודה שצריך לשים לב אליה, שכל לקוח משתמש במספר פורט אחר בשרת ולכן יכול להיות שנקבל הודעת שגיאה אם יהיו לנו יותר מלקוח אחד שפונה לסוקט על אותו הפורט. אם נתקלנו בזה, נסגור את הסוקטים של הלקוחות ונשנה את מספר הפורט למספר שנבחר גם אצל השרת וגם אצל הלקוח. לעיתים ניתקל במצב קיצוני שבו למרות שווידאנו שאין שרת שעדיין פעיל, בניסיון להריץ שרת חדש עדיין תופיע שגיאת פורט תפוס. זה עלול להתרחש אם תוך כדי ריצה קוד הלקוח קרס, ואז מערכת ההפעלה עדיין מחזיקה את הפורט של השרת במצב תפוס עוד פרק זמן קצר לפני שהיא משחררת. אפשרות נוספת היא שהפורט של השרת כבר תפוס על ידי אפליקציה אחרת שמשתמשת בו. הפתרון בשני המקרים הוא להחליף את מספר הפור אליו מאזין השרת.



ראינו איך להריץ שרת ולקוח מקומית על אותו מחשב, אבל אם אנו רוצים להריץ על שני מחשבים שונים אלו שלבי הפעולה:

1. וודאו שבשני המחשבים מותקנת סביבת העבודה של פייתון
2. בחרו מחשב כלשהו שיקרא מעתה "שרת" ומצאו את כתובת הIP שלו (על ידי הרצת ipconfig בcmd, כתובת הIP מופיעה בשורה המתחילה במילים IPV4 address ומתחילה במספר 10 או 192).
3. ודאו כי שני המחשבים מחוברים לאותה רשת ביתית (על ידי שליחת ping בcmd של מחשב הלקוח אל מחשב השרת וקבלת reply עם אותה כתובת)
4. במחשב הלקוח היכנסו לקוד הלקוח ובמקום הכתובת 127.0.0.1 הזינו את כתובת הIP של השרת.

עד כה למדנו לתכנת סוקטים מבוססי פרוטוקול TCP של שכבת התעבורה. לעיתים הפרוטוקול גורם להעברה איטית מידי של מידע ואנו מעדיפים שהמידע יהיה לא אמין אך יעבור מהר. למשל כשאנו עושים שיחות וידאו, מידע שלא הגיע ליעדו ואנו מנסים לשחזר אותו, עד שנצליח הוא יהיה מידע לא רלוונטי ולכן עדיף כבר לשדר את המידע כמה שיותר מהר ולא להתעכב.

פרוטוקול UDP (User Datagram Protocol) הוא פרוטוקול של שכבת התעבורה, והוא דומה לפרוטוקול לא אמין. בפרוטוקול UDP נוספים רק 4 שדות למידע של שכבת האפליקציה:

1. פורט המקור, שממנו נשלחת ההודעה
2. פורט היעד, שאליו נשלחת ההודעה
3. אורך ההודעה
4. בדיקת תקינות של המידע הנשלח (אם המידע בפקטה הגיע משובש הפקטה תיזרק, בניגוד לTCP שישלח אותה שוב).

בשל הכמות הקטנה של המידע המתווסף לפרוטוקול UDP, הוא חסכוני יותר מTCP ולכן מהיר יותר, ומכאן גם מתאים יותר לתוכניות שלא דורשות העברה אמינה של מידע, כמו בשיחת וידאו, משחקי וידאו ושיחות קוליות.

דוגמה נוספת להעברת מידע הממחישה מדוע TCP אינו תמיד מושלם היא הודעות קצרות מאוד שכל אחת מהן עומדת בפני עצמה ואינה חלק משיחה, לדוגמה בדיקת כתובת הIP של אתר אינטרנט. זוהי בקשה קצרה מאוד הנשלחת לשרת שמחזיק את כתובת הIP של אתרי אינטרנט. אין סיבה לשלוח הודעה כזו בפרוטוקול כמו TCP שידאג לקבל אישור מהשרת על כך שהבקשה שלנו התקבלה, שהרי אם הבקשה הגיעה לשרת, נקבל בכל מקרה תגובה הכוללת את כתובת הIP המבוקשת.

בשורה התחתונה, יש מקרים בהם העברת המידע חייבת להיות מדויקת לחלוטין, לדוגמה שליחה של קובץ – אם הקובץ יגיע פגום הוא לא יהיה שימושי. לעומת זאת, במקרים אחרים, למשל כשמתבצע שידור streaming (שידור בזמן אמת), איבוד של חלק מהמידע לרוב לא יפגום בשיחה.

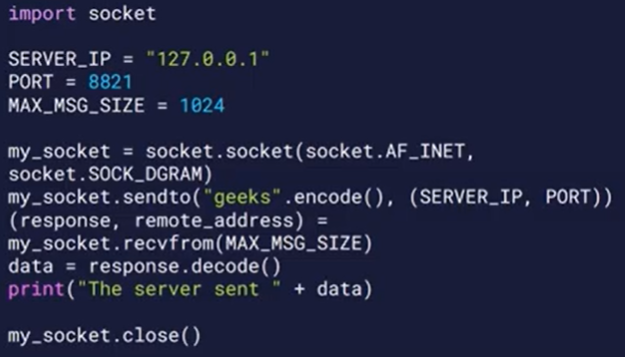


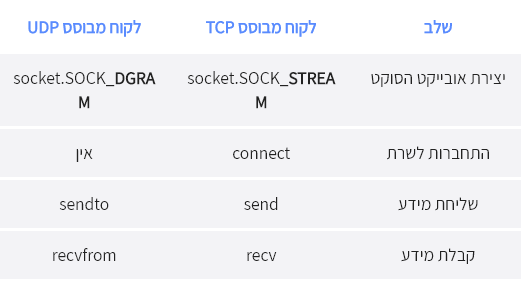
נתחיל מכתיבת לקוח פשוט באמצעות פיתון:

1. יצירת סוקט – נייבא את הספרייה socket ונייצר אובייקט מסוג סוקט עם הפרמטרים:
   * socket.AF\_INET
   * socket.SOCK\_DGRAM – יוצר סוקט עם פרוטוקול UDP
2. בפרוטוקול UDP אין צורך להשתמש במתודה connect ששולחת בקשת התחברות מהלקוח לשרת, אלא לשליחת מידע נשתמש במתודה sentto המקבלת כפרמטרים:
   * את המידע שברצוננו לשלוח אחרי שמפעילים עליו את הפונקציה encode()
   * טאפל המתאר את התוכנה המרוחקת ומכיל כתובת IP ומספר פורט
3. כדי לקבל מידע עלינו להשתמש במתודה recvfrom, המאפשרת לנו לדעת ממי קיבלנו את המידע. היא מחזירה טאפל שמורכב משני איברים:
   * Response שהוא ההודעה
   * Remote address שהוא כתובת הIP של השולח

הסיבה לכך היא שבניגוד לפרוטוקול TCP, בפרוטוקול UDP אין קישור קבוע בין שני הצדדים לשיחה, לכן בכל הודעה שאנו מקבלילם צריך לדעת מי המקור שלה.

1. לבסוף נסגור את אובייקט הסוקט שיצרנו



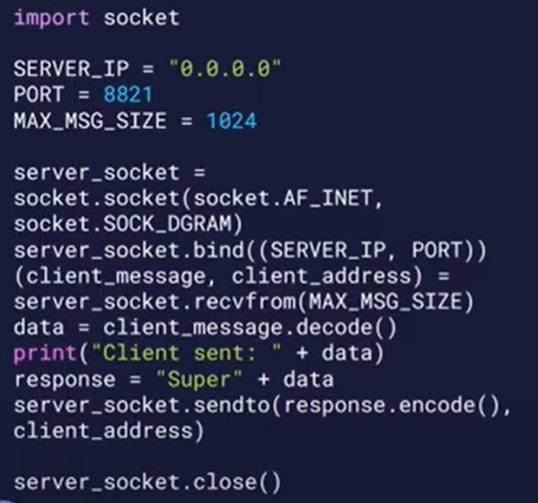
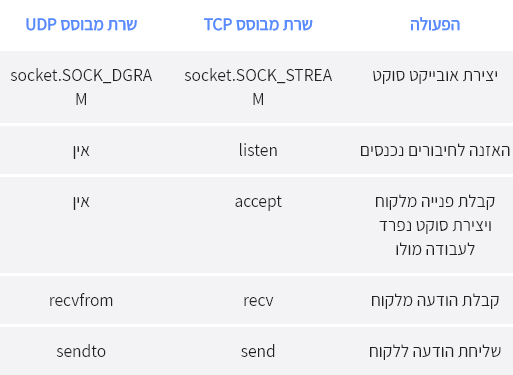


כעת ניצור סוקט לשרת:

1. נייבא את המודול של סוקט
2. נגדיר את הקבועים שישמשו אותנו בהמשך:
   * כתובת הIP של השרת - 0.0.0.0
   * הפורט – 8821
   * האורך המקסימלי של ההודעה התואם לאורך שקבענו בקוד הלקוח – 1024
3. נייצר אובייקט מסוג סוקט עם הפרמטרים:
   * socket.AF\_INET
   * socket.SOCK\_DGRAM – יוצר סוקט עם פרוטוקול UDP
4. נבצע קישור של אובייקט הסוקט שיצרנו לכתובת מקומית שהגדרנו על ידי המתודה bind
5. בניגוד לשרת הTCP שמשתמש במתודה listen, ובמתודה acceptף, התקשורת בין השרת ללקוח עוברת ישירות דרך סוקט שםתח השרת, הוא עונה ללקוחות באותו סוקט שהוא מאזין אליו ולכן אין צורך ביצירת סוקט חדש, השרת לא צריך להאזין לחיבור נכנס או להסכים לקבל חיבור חדש. כיוון שאנו לא מייצרים סוקט חדש עם קישור קבוע בין השרת לבין הלקוח שפונה אליו. במקרה של UDP, על גברי אותו הסוקט שהשרת מאזין אליו הוא גם עונה ישירות לכל לקוח.

כיוון שלא הקמנו קישור, המתודה recvfrom מחזירה:

* + את המידע שהלקוח שלח (אותו שמרנו אל המשתנה client\_message)
  + את הכתובת של הלקוח (אותה שמרנו במשתנה client\_address) כתובת זו תשמש אותנו כשנרצה לשלוח הודעה חזרה אל הלקוח

1. כדי לשלוח הודעות, נייצר תחילה את התשובה שלנו ואז נשלח את המידע ללקוח על ידי שימוש במתודה setdto המקבלת את המחרוזת ואת פרטי הלקוח.
2. לבסוף נסגור את הסוקט

**ריבוי משתמשים בשרת TCP**

בשני הפרקים האחרונים למדנו לממש שרת אשר יודע לקבל ההודעה מלקוח ולענות לו. לאחר שהשיחה מסתיימת, יכול השרת להתפנות ולשרת לקוח נוסף. מה קורה אם רוצים שהשרת ידע לשרת יותר מלקוח אחד? במקרים מסוימים שרת UDP יכול לשמש כשרת מרובה משתמשים, למשל עבור שיחות וידאו מרובות משתמשים כמו זום, פרוטוקול וUDP ייתן מענה טוב. אבל שימוש בפרוטוקול UDP לא מספיק טוב לנו לכל שרת מרובה משתתפים.

שרת מסוג TCP מחכה לחיבורים חדשים עד שהוא מקבל התחברות מלקוח חדש. בשלב זה הוא נקשר ללקוח שהתקשר איתו הוא מטפל רק בו עד שאחד הצדדים יחליט לסיים את השיחה. אבל אם השיחה תימשך זמן רב, יווצר "לחץ" ותור ארוך של לקוחות שמחכים להתחבר לשרת.

כדי לכתוב שרת TCP שיתמוך בכמה משתמשים בו זמנית, עלינו להכיר טכניקה חדשה לניהול סוקטים, בעזרת פונקציה הנקראת select.

שרת מרובה לקוחות צריך לסרוק כל הזמן את הלקוחות המחוברים אליו למערכת ולראות אם הם צריכים לקבל שירות. הפונקציה select מבצעת מעקב אחרי הסוקטים שמבקשים להתחבר ומדווחת לנו ברגע שאחד מהסוקטים מוכן לקריאה. הפונקציה מוגדרת באופן הבא:

ready\_to\_read, ready\_to\_write, in\_error = select.select(read\_list,

write\_list, error\_list)

הפונקציה מקבלת שלוש רשימות ומחזירה שלוש רשימות. הפרמטרים שמקבלת הפונקציה הן הרשימות הבאות:

* Read\_list – רשימת סוקטים אשר נרצה לקרוא מהם
* Write\_list – רשימת סוקטים אשר נרצה לכתוב להם
* Error\_list – רשימת סוקטים אשר נרצה לדעת אם התרחשה בהם שגיאה

הפונקציה select תסרוק את הסוקטים שברשימה הראשונה, read\_list ותחכה שאחד מהם יסמן לה שיש לו מידע שאפשר לקרוא. ברגע שזה יקרה, הפונקציה תסתיים ותחזיר לנו את ערכי ההחזרה. חשוב לציין שselect היא פונקציה שעוצרת את המשך הריצה של התוכנית שלנו, עד שהיא מסתיימת. פונקציות כאלה נקראות blocking, כיוון שהן חוסמות את ריצת התוכנית לזמן מסוים.

ברגע שהפונקציה select תסתיים, לתוך המשתנה ready\_to\_read תוחזר רשימה הכוללת רק את הסוקטים שאפשר לקרוא מהם, כלומר רק את הסוקטים שיש להם מידע חדש.

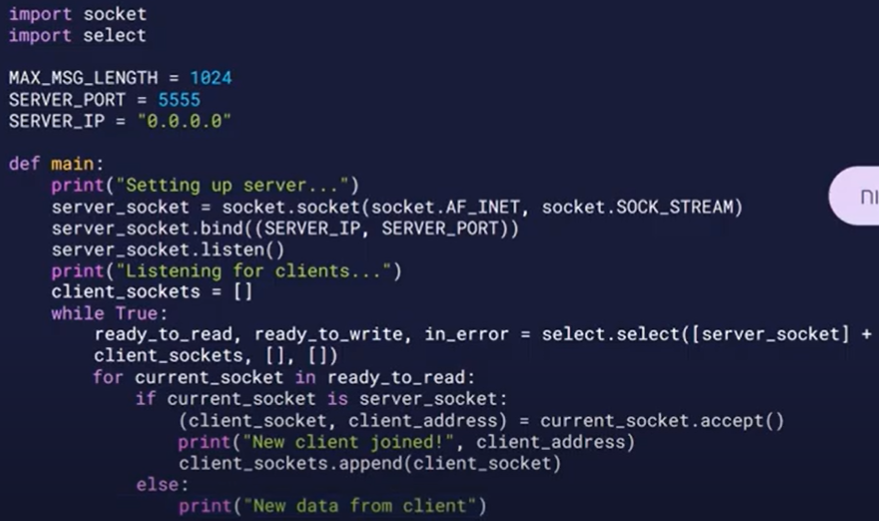
אנו לא נדון בשני הערכים הנוספים שהפונקציה מחזירה.

כדי להשתמש בselect בצורה הבסיסית ביותר, הדבר שמעניין אותנו הוא בעיקר הפרמטר הראשון, read\_list, שהוא רשימת כל הסוקטים שאנו רוצים לסרוק. רשימה זו מורכבת מסוקט השרת, אותו אחד שאנו מכירים מכל שרת TCP שראינו עד כה, וזה לשם לקוחות חדשים שיכולים להגיע, ורשימת הלקוחות הקיימים במערכת, כלומר כל הסוקטים שהתחברו כבר לשרת. כלומר לread\_list נעביר את client\_sockets + [server\_socket]

במקרה שלנו, נממש בצורה פשוטה ולא נעביר כלום לשני הפרמטרים השניים (כלומר נעביר רשימה ריקה []).

ברגע שהפונקציה תסתיים, נתמקד בערך שחזר, הרשימה ready\_to\_read. רשימה זו כוללת את כל הסוקטים שהשרת צריך לטפל בהם, כלומר אלו שסימנו לנו שיש בהם מידע. בתוכם יכולים להופיע סוקטים של לקוחות קיימים שקורה איתם משהו חדש, או לקוחות חדשים שרוצים להתחבר לשרת. כדי לטפל בלקוחות אלה נוכל ליצור לולאה פשוטה שעוברת על רשימת הלקוחות באופן הבא: for current\_socket in ready\_to\_read:

#Do something for every client



כעת נשדרג את הקוד שכתבנו של שרת TCP על ידי הוספת הפונקציה select.

השרת שנממש יהיה שרת הדים מסוג TCP שמטפל בלקוחות רבים בו זמנית – כמה לקוחות יוכלו לשלוח לשרת בו זמנית הודעה וכל לקוח יקבל בחזרה מהשרת בדיוק את אותה הודעה ששלח, בלי להמתין שהלקוחות הקודמים יסגרו את הסוקט שלהם מול השרת.

1. הרמת חיבורים מול לקוחות חדשים והוספתם לרשימה – בשלב זה נממש שרת שלא מבצע דבר מלבד לקבל חיבורים מלקוחות והדפסת הודעה קבועה כאשר לקוח מתחבר. תחילה נקבע קבועים שימושיים ונייבא את הספרייה select. ניצור אובייקט מסוג סוקט ונקרא למתודה bind כדי לקשר אותו לכתובת מקומית. לבסוף נתחיל לבצע האזנה לשרת.
2. קבלת לקוחות חדשים – ניצור רשימה בשם client\_sockets שתכיל את כל אובייקטי הסוקט של הלקוחות שמחוברים לשרת, כשהתוכנית מתחילה לרוץ אין אף לקוח מחובר ולכן הרשימה הזו תתחיל ריקה.
3. בשלב זה עלינו לטפל בפניות שמגיעות לשרת, לשם כך ניצור לולאת while מרכזית שבה בכל פעם נקרא לפונקציה select פעולת הלולאה הזו אומרת שכל עוד השרת פועל, הוא סורק שוב ושוב את המצב בשטח.

במצב התחלתי עוד לא התחברו לקוחות אך הרשימה לא ריקה, כי היא מכילה את server\_socket שממנו אפשר לקבל חיבורים חדשים. אם אפשר לקרוא מסוקט זה, המשמעות היא שיש פניה של לקוח חדש ובמקרה זה נקרא לaccept ונקים חיבור עם הלקוח.

ברגע שיסתיים הselect נקבל את הרשימה של הסוקטים שאפשר לקרוא מהם ready\_to\_read. עבור כל סוקט קריא עלינו להבין אם הוא סוקט של השרת או של לקוח קיים ולתפעל בהתאם.

* + סוקט שרת – פניה מלקוח חדש – השרת צריך לפתוח מולו חיבור באמצעות accept
  + סוקט של לקוח קיים – השרת צריך לקרוא את ההודעה שהגיעה (באמצעות recv)

בכל ריצה של הלולאה מתבצעת סריקה עם select (שתימשך כמה זמן שצריך, עד שיהיו סוקטים שלהם מידע המוכן לקריאה) ולאחר מכן יתבצע טיפול בקריאת המידע מהלקוחות, וחוזר חלילה.

