רשתות תקשורת מחשבים - תרגיל בית מס 1

מגישים:
ליאור בלנקוב - ת.ז. 319125134
נדב דרורי - ת.ז. 208935882

תיאור הפרוטוקול:
הפרוטוקול שממשנו מבוסס על TCP. התעבורה היא בתורות: לאחר שהתקיים חיבור מוצלח בין הלקוח לשרת, בשלב הראשון השרת מצפה להודעה מהלקוח. לאחר קבלת ההודעה הינו מנתח אותה ובשלב השני מחזיר תגובה ללקוח בהתאם, וכך השלבים חוזרים על עצמם עד שהלקוח מנותק/מתנתק מהשרת.
בעת שליחת הודעה, השרת שולח את גודל ההודעה (אובייקט INT) ב-4 בתים ללקוח ואחרי זה שולח את ההודעה.
בעת קבלת הודעה, השרת מצפה לקבל 4 בתים שייצגו את אורך ההודעה שאמור לקבל. השרת בודק שקיבל הודעה בגודל שנשלח ורק אז מבצע ניתוח שלה ומגיב בהתאם.

אופן עבודה - שרת:

- חשוב לציין: מעבר לקוד שרת ולקוח, יש קובץ פייתון שלישי בשם numbers_service.py שמכיל מתודות ומחלקות שמשמשות גם את תוכנת השרת והלקוח. שני המרכיבים העיקריים:
  - מחלקות הודעה: המחלקה הבסיסית הינה SocketMessage שמכילה שדות של תוכן ההודעה בבתים וגודל התוכן. מעבר יש עוד 2 מחלקות שיורשות ממנה: . הן OutgoingSocketMessage ו-IncomingSocketMessage שכוללות מתודות נוחות למען מימוש המנגנון עבודה בלי חסימות.
  - מתודות שממשות לוגיקת שרת, מנתחות הודעות של משתמש - נתוני המשתמש / פקודות. בפרט execute שמקבלת את ההודעה כאובייקט str, מפרקת אותה לגורמים, מפעילה פונקצייה מתאימה לפי התוכן הפקודה ומחזירה אובייקט str המשמש כ-response למשתמש (או None אם לא מזהה פקודה).

1. **טיפול בחסימות:** כדי למנוע חסימות אנחנו עבדנו עם selector והמנגנון שראינו בכיתה עם מבני נתונים עזר.
   ברגע שהשרת שולח הודעה ל-socket (הינו ברשימת writable ב-selector. בעצם כל socket מתחיל ברשימת writable לאחר חיבור כי השרת שולח הודעת פתיחה) אז הוא שומר במילון אותו כמפתח ואת ההודעה כאובייקט OutgoingSocketMessage. עם כל איטרציה של ה-selector, הוא בוחן האם כל ההודעה נשלחה, אם לא אז הינו לוקח ממנה כמות קבוע של בתים DATA_BANDWIDTH = 4 (זהו משתנה שניתן לשנות בתוכנה) ושולח אותם עם פקודה send של ה-socket. לאחר ששלח את כולה, מוחק את המפתח מהמילון. כעת ה-socket זז לרשימה readabable ב-selector כי לפי הפרוטוקול מצופה לתגובה מהלקוח.
   ברגע שהשרת מקבל אורך הודעה מ-socket. אז הוא שומר במילון אותו כמפתח ואת ההודעה כאובייקט IncomingSocketMessage. עם כל איטרציה של ה-selector, הוא בוחן האם כל ההודעה התקבלה לפי האורך, אם לא אז הינו אוסף כמות קבוע של בתים DATA_BANDWIDTH = 4 ע"י פקודה recv של ה-socket. לאחר שקיבל את כולה, מוחק את המפתח מהמילון ושולף את ההודעה לניתוח. כעת ה-socket זז לרשימה writable ב-selector לפי הפרוטוקול.
   בעצם עם המנגנון הזה מכיוון שעבור כל socket אנחנו שולחים/מקבלים כמות קטנה של בתים ועוברים לטפל בהבא ובכך לא מפספסים אף socket, אנחנו מבטיחים שלא יכול להיווצר מצב של חסימות.

2. **טיפול בהודעות לקוח (workflow השרת):** נסתכל על השלבים שבהם השרת מגיב להודעות לקוח:

a. שלב login: אחרי הודעת הפתיחה השרת מצפה להודעה מהמשתמש בפורמט הבא:
User: [user_name]\nPassword: [user_password]
התגובה נקבעת לפי ניתוח ההודעה, יש 3 אופציות:
   i. הפורמט נכון, ונתוני המשתמש תואמים לקובץ שממנו קורא השרת. במקרה זה השרת מעביר את הלקוח לשלב הבא שבו יכול להזין פקודות (יש 2 מבני נתונים set זרים ששומרים socket של לקוח: אחד של לקוחות שמנסים להתחבר והשני לכאלו שכבר מחוברים - כך השרת מצליח לזכור ולהבדיל בינהם).
   ii. הפורמט נכון, אך נתוני המשתמש לא תואמים לקובץ. השרת שולח הודעה "Login failed" ונותן למשתמש נסיון נוסף לשלוח לו נתונים.
   iii. הפורמט לא נכון. לפי הנחיות התרגיל, נשלח הודעה "Disconnected from server" וננתק את המשתמש מהשרת.
b. שלב פקודות: אחר הודעת ה"ברוך הבא משתמש", השרת מצפה פקודה מהמשתמש. מכיוון שלפי הנחת התרגיל הפקודה נשלחה בפורמט נכון אז יש רק 2 תגובות אפשריות:
   i. השרת מזהה את הפקודה ולכן ינתח אותו עם execute וישלח את תוצאת הפונקציה ללקוח בפורמט:
   response : [result].
   ויצפה לפקודה הבאה.
   ii. השרת לא מזהה את הפקודה, לפי הנחיות התרגיל, נשלח הודעה "Disconnected from server" וננתק את המשתמש מהשרת.

3. **טיפול בשגיאות Socket:** כל פעולה של socket עטופה בבלוק של try/except(OSError) as exception, והטיפול עבורם שונה:
   a. recv, send: פעולות שקורות במקרים שונים לאורך הלולאה האינסופית של workflow השרת ולכן נרצה לטפל בשגיאות של פעולות אלו שלא יפריעו לעבודה של השרת ולשאר ה-sockets. במקרה של שגיאה ננתק את ה-socket ונמחק אותו מכל מבני הנתונים של השרת (מילוני הודעות ורשימות writable/readable). אם exception.errno == errno.ECONNRESET אז משמע התנתקות מצד לקוח ולא נדפיס את השגיאה, בכל שגיאה אחרת כן נדפיס exception.strerror.
   b. accept: פעולות שקורות רק כאשר לקוח מנסה להתחבר. במקרה זה נדפיס את השגיאה ופשוט לא נחבר אותו ולא נכניס אותו למבני נתונים.
   c. bind, listen: פעולות שקורות רק לפני setup של השרת. לכן אין צורך לדאוג ל-workflow, במקרה של שגיאה נדפיס הודעת שגיאה ונפסיק את התוכנה.

<u>אופן עבודה - לקוח:</u>

- חשוב לציין: אחרי קבלת הודעת הפתיחה מהשרת, בקוד יש שני פקודות input ומצופה כי המשתמש בשורה הראשונה יזין User: user_name ואז בשורה השניה Password: user_password, כדי שהפורמט יהיה תקין (לפי הנחיה בפורום).

1. **טיפול בפקודות הלקוח:** עקב ההנחיה הבאה:

מולו. כמו כן לשם פשטות ניתן להניח כי אם מתקבל קלט לא בפורמט צפוי מהמשתמש, התוכנה תדפיס הודעת שגיאה ותתנתק מהשרת.

נרצה לבדוק שהפקודה בפורמט הבא:
[command_name]: [cmd_arg_1] … [cmd_arg_n]

אם אין arguments אז פשוט [command_name]. וגם ש-arguments תקינים. למען כך לפני שליחה של כל פקודה הלקוח מריץ שני מתודות עזר:

a. check_cmd_argument_amount - בודקת את פורמט הפקודה. אם זו פקודה מוכרת לשרת אז גם בודקת שכמות ה-arguments תקינה. בכל מקרה של אי-תקינות תקפוץ שגיאה.
b. execute - בעצם הלקוח מריץ אותה פקודה שהשרת מריץ אך אינו שומר את התוצאה שלה בשום מקום, הסיבה היחידה להרצה היא כדי לזהות אינפוט בעייתי אם תקפוץ שגיאה בהרצת הפקודה (למשל חילוק ב-0).

שני המתודות עזר נמצאים בבלוק try/except ואם קופצת שגיאה בהם אז כפי שהונחה נדפיס הודעת שגיאה, ננתק את המשתמש מהשרת (הוא ישלח פקודת quit במקום) ואחרי שיסיים לשלוח תסתיים התוכנית.

2. **טיפול בשגיאות Socket:** קוד הלקוח עטוף בבלוק try/except(OSError) as exception. בכל מקרה נדפיס את שגיאת socket ונסיים את ריצת תוכנה. מכיוון שהשרת יודע להתמודד עם התנתקות מצד לקוח, אז אין מה לדאוג מצד הלקוח במקרה והתוכנה מסיימת ריצה בפתאומיות בגלל שגיאה.