בס"ד

פרויקט סיום קורס "רשתות מחשבים ואינטרנט" – משחק "איקס עיגול"

מגיש הפרויקט:

- אביעד שריד
- ת.ז. 208389577

פרטים טכניים:

- מערכת הפעלה: win11
- שפת תכנות: Python
- גרסת פיית'ון: 3.12

תוכן עניינים

בס"ד

אופן הרצת הפרויקט

בתיקייה הראשית של הפרויקט יש 2 קבצי batch וקובץ vbs. כדי להריץ את הפרויקט יש להריץ תחילה את ה-server באמצעות הקובץ runServer.bat. לאחר מכן ניתן להריץ מספר clients כרצונך, כל אחד ע"י פתיחה של הקובץ runClient.bat או ע"י פתיחה של הקובץ שהסיומת שלו היא vbs (ללא חלון שחור).

הערה: אם אין ברשותך פיית'ון בגרסת 3.12, יש להריץ את קבצי py עצמם: תחילה את הקובץ appServer.py בתיקיית Server ואז את הקובץ appClient.py בתיקייה Client.

תיאור האפליקציה, מרכיביה וחלקיה השונים

האפליקציה המוגשת מממשת משחק איקס-עיגול מרובה משתתפים, כאשר הארכיטקטורה בה נקטתי היא משולבת: client-server architecture & layered architecture. נסביר על משמעות כל אחת מהארכיטקטורות:

- ארכיטקטורת לקוח-שרת מחלקת את האפליקציה לשני חלקים מובחנים: חלק השרת וחלק הלקוח. ייתכן ששני החלקים ייושמו באותו host אך על כל פנים תהיה ביניהם הפרדה. הקשר בין החלקים יתבצע באמצעות רשת תקשורת כלשהי (אינטרנט, למשל). על השרת לשרת מספר רב של לקוחות במקביל ללא תלות זה בזה תוך מיקסום חווית הלקוח על אף עומסים שונים בצד השרת.

- ארכיטקטורת השכבות מחלקת את האפליקציה לשכבות לוגיות על פי תכלית כל חלק באפליקציה. כך, צד הלקוח חולק לשתי שכבות: שכבת התצוגה ושכבת התקשורת. צד השרת, לעומת זאת, נדרש לחלוקות נוספות בהתאם לכך שתפקידיו עולים על תפקידי הלקוח: שכבת הלוגיקה, שכבת הגישה למסד הנתונים, שכבת התקשורת ושכבת הישויות. נסביר:

  - שכבת התצוגה אחראית להמרת מידע שמתקבל לכדי אובייקטים גרפיים. לפיכך, יישמתי אותה רק בצד לקוח משום שבשרת אין צורך (לטעמי) בממשק גרפי.

  - שכבת התקשורת אחראית לקבל הודעות מרשת האינטרנט, לפרק אותן ולהעביר את תכולתן ליעדה. כמו כן אחראית השכבה לבנות הודעות בהתאם לדרישת הלקוח/שרת ולשולחן על גבי האינטרנט.

  - שכבת הלוגיקה מומשה רק בצד שרת ותפקידה לנתח את המידע שהתקבל מהלקוחות ולמעשה לנהל מבחינה לוגית את האפליקציה. זו הליבה של הלוגיקה במערכת.

  - שכבת הגישה לנתונים ממסדת גישה מסודרת ומאובטחת לבסיסי הנתונים השונים הקיימים בצד השרת. שכבה זו מממשת את פעולות CRUD ופעולות נוספות הקשורות לבסיסי הנתונים.

  - שכבת הישויות כוללת בתוכה את כלל המחלקות המייצגות אובייקטים שונים באפליקציה, כמו למשל: User, Game, ועוד.

במהלך פיתוח האפליקציה השתמשתי בתבניות עיצוב שונות שמטרתן לאפשר scalability וגם maintainability של האפליקציה. אפרט יותר:

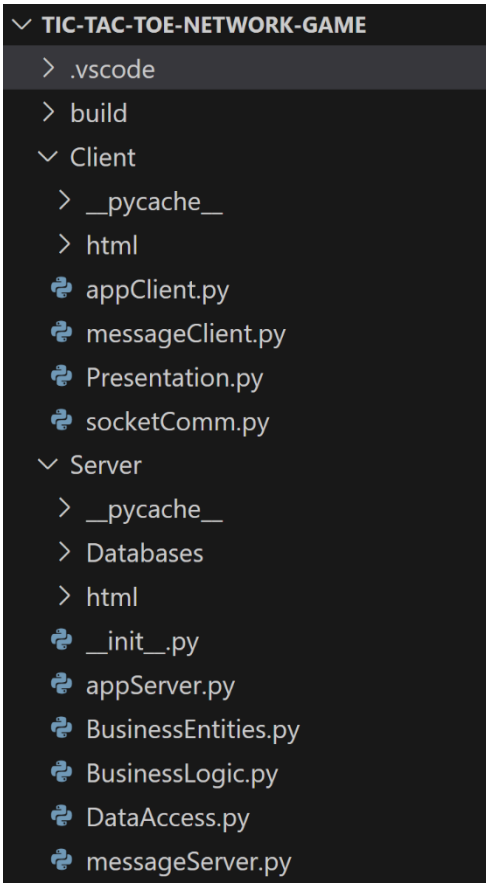
- בשכבת הגישה לנתונים השתמשתי בתבנית עיצוב Factory אשר יודעת להוסיף מימושים חדשים לפונקציונליות של השכבה (בעיקר פעולות CRUD, כאמור) בהתאם לבסיסי נתונים שונים בהם רוצים להשתמש. תבנית זו מממשת את עיקרון OCP בכך שלא צריך לשנות דבר בשכבה מלבד הוספת מימוש חדש (תוך מימוש ה-interface) והוספת שורה אחת בלבד בתוך ה-Factory.

בס"ד

- עשיתי מאמץ להקפיד על עיקרון Separation of Concerns ועל Single Responsibility Principle כך שניפוי התוכנית מבאגים ובדיקת התוכנה נעשו פשוטים יותר, אף שהדבר דרש מחשבה רבה יותר וזמן הפיתוח התארך.

כעת נפרט על חלקי האפליקציה השונים. עץ קבצי הפרויקט נראה כך:

(הפרויקט מחולק לשתי תיקיות עיקריות: Client, Serve)



נתחיל בצד הלקוח. כאמור, צד הלקוח חולק לשתי שכבות: שכבת התקשורת ושכבת התצוגה. ה-entry point של צד הלקוח נמצא בקובץ appClient.py.

מתוך הקובץ appClient.py אנו יוצרים את ה-GUI ושולטים על החלפת ה-pages דרך הפונקציה show_page. כמו כן בקובץ זה יוצרים socket לתקשורת עם ה-server ושם נוצר מופע של המחלקה Message בקובץ socketComm.py.

הקובץ socketComm.py יוצר את ה-socket ומנהל אירועים של קריאה וכתיבה מהשרת.

הקובץ messageClient.py מכיל את המחלקה Message אשר מנהלת חבילה שמגיעה או שנשלחת.

הקובץ Presentation.py מנהל את ה-GUI.

כעת נעבור לצד השרת. ה-entry point של השרת נמצא בקובץ appServer.py. בקובץ זה נוצרים שני דברים:socket לתקשורת עם הלקוחות, כן עבור כל לקוח נוצר מופע של Message שמנהל את התקשורת עימו. בנוסף נוצרת שכבת הלוגיקה של האפליקציה דרך import של הקובץ BusinessLogic.py (ושם נוצר ה-access לבסיס הנתונים).

בס"ד

הקובץ BusinessEntities.py מכיל את כל הישויות שהמערכת זקוקה להן (.user, game etc).

הקובץ BusinessLogic.py מכיל אוסף פונקציות המבטאות את הלוגיקה של המערכת.

הקובץ DataAccess.py מכיל אוסף פונקציות המנהלות את הקשר עם בסיס הנתונים ומסדירות אותו.

הקובץ messageServer.py מכיל את המחלקה Message אשר מנהלת חבילה המגיעה מלקוח או יוצאת לכיוונו.

בסיס הנתונים מורכב מ-3 קבצי JSON:

- games.json – משמש לאחסון רשומות על משחקים שהסתיימו (תאריך יצירה, משך זמן המשחק, מס' משתתפים וכדומה).
- users_tokens.josn – משמש לאחסון מידע בסיסי בלבד על משתמשים (token ושם המשתמש).
- users_full.json – משמש לאחסון מידע מלא על כל משתמש (שם, token, מס' משחקים בו השתתף, מס' נצחונות וכדו).

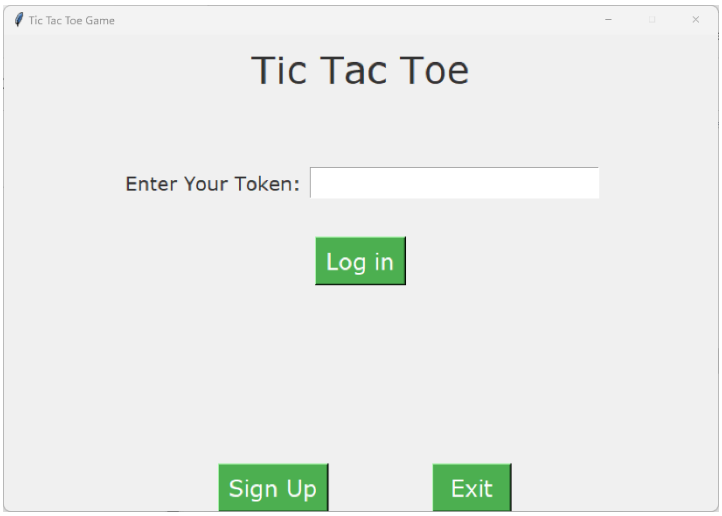
תיעוד והערות

כל קובץ באפליקציה מוער באופן מלא. לצפייה בתיעוד הקבצים ניתן לפתוח את הקובץ Documentation.html בתיקיית הפרויקט, או דרך הקישור הבא: Documentation.html

(התיעוד נוצר באמצעות ה-extenstion שנקרא pdoc3)

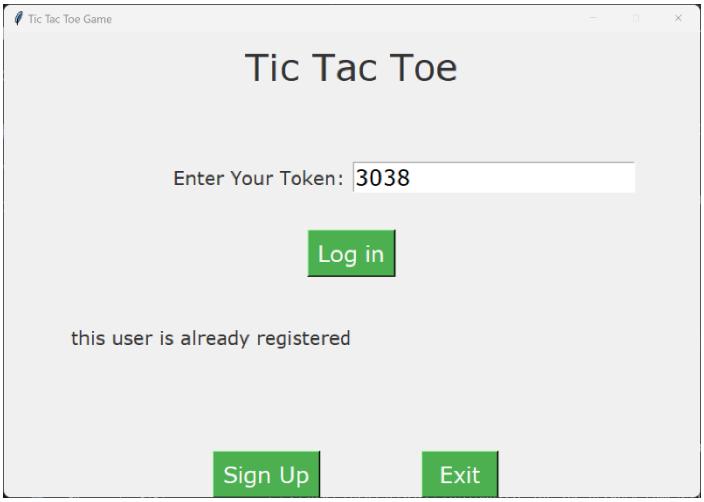
תמונות של חלונות האפליקציה

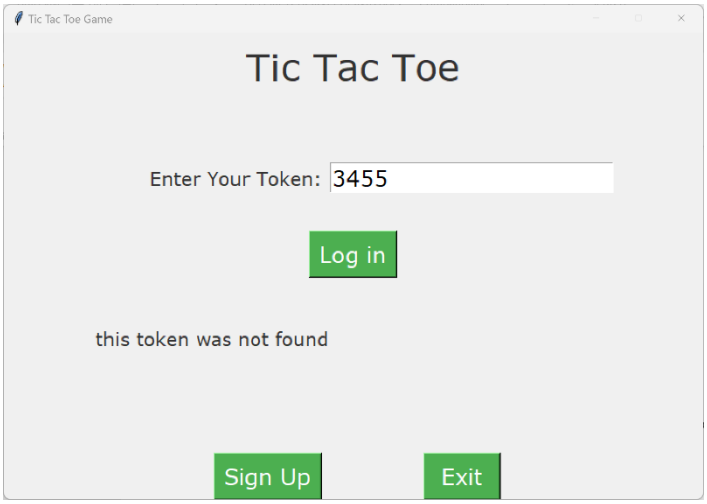
מסך הכניסה נראה כך:



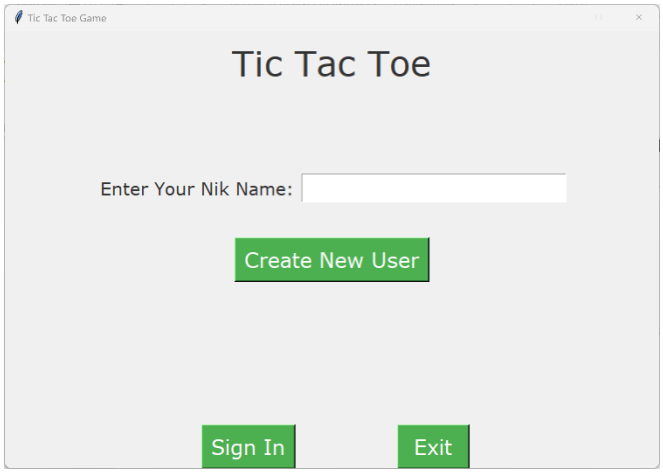
בס"ד

במקרה שה-token אינו קיים במערכת נקבל את המסך הבא מימין, ואם המשתמש כבר מחובר נקבל את המסך המופיע מטה משמאל:



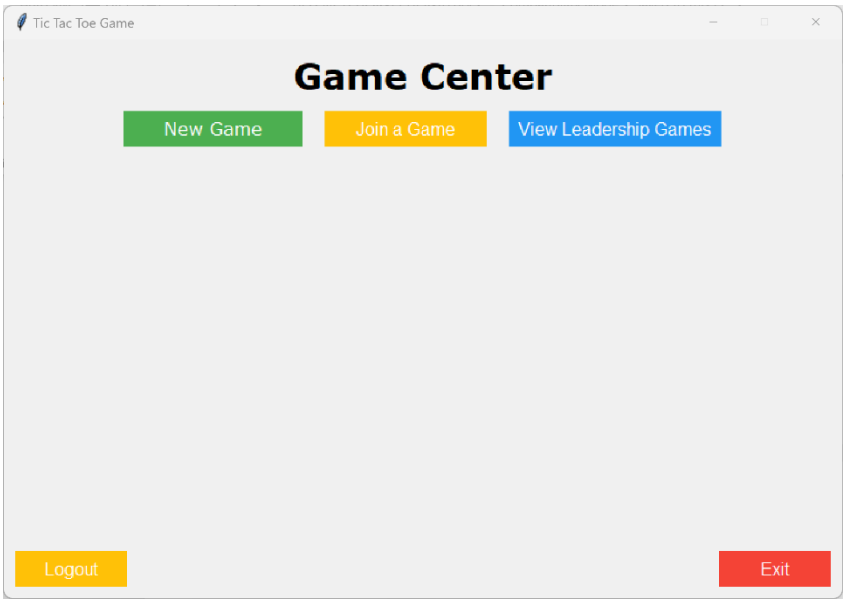


לרישום משתמש חדש נלחץ על הכפתור Sign Up ונקבל את המסך הבא:

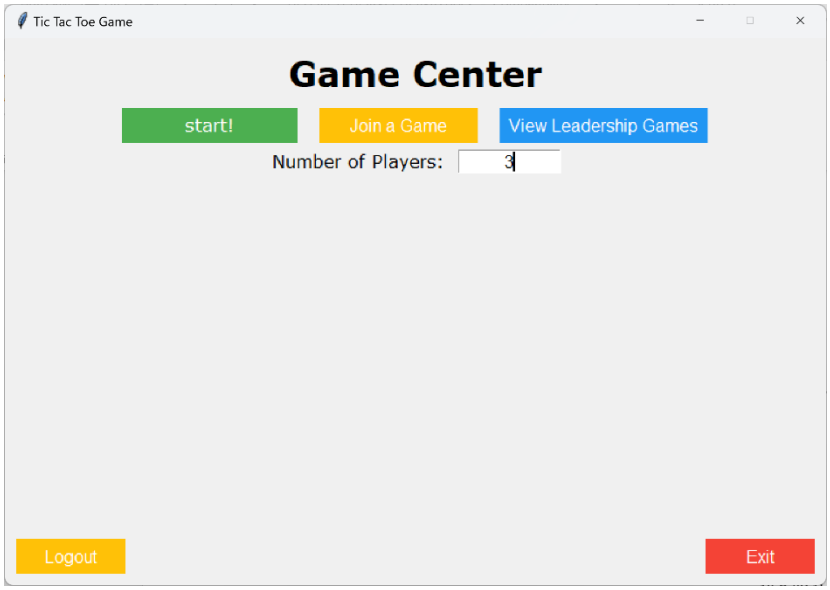


בס"ד

בכניסת משתמש קיים או בכניסת משתמש חדש, נקבל את המסך הראשי:

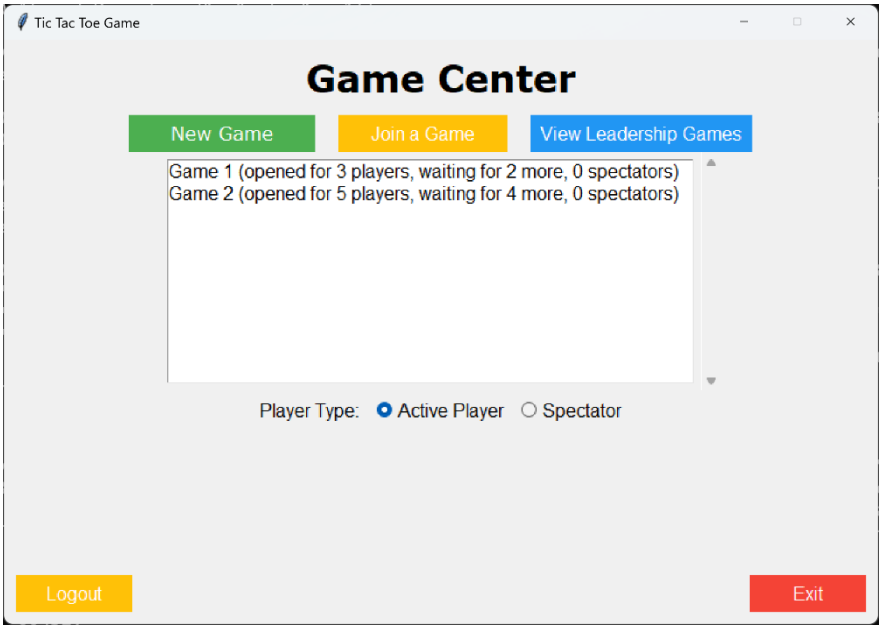


בלחיצה על New Game תצוץ תיבת טקסט להזנת מס' משתתפים (בין 2 ל-8 שחקנים):

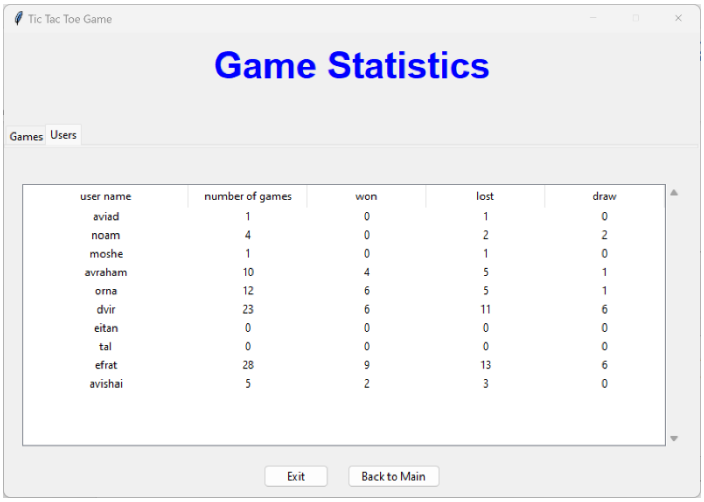


בלחיצה על Join a Game תופיע רשימת משחקים אליהם ניתן להצטרף (כשחקן ניתן להצטרף רק למשחקים שעוד לא התחילו, כצופה ניתן להצטרף גם למשחקים שכבר החלו). הצטרפות למשחק ע"י double click עליו.

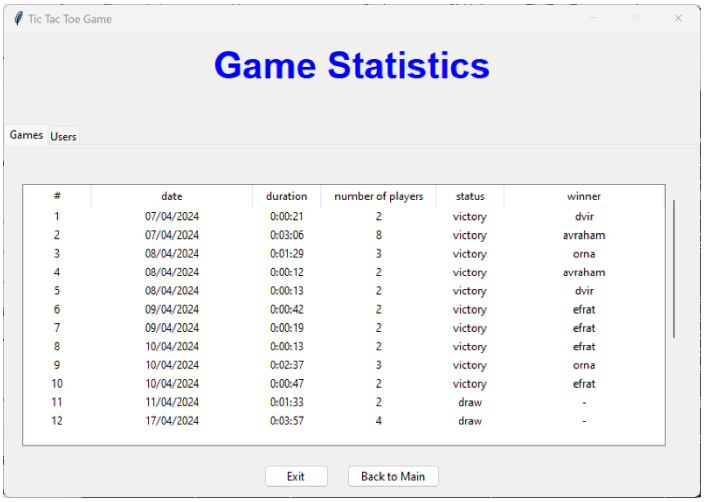
בס"ד



ניתן אף לצפות במשחקים האחרונים או בסטטיסטיקות על משתמשים, בלחיצה על הכפתור הכחול:



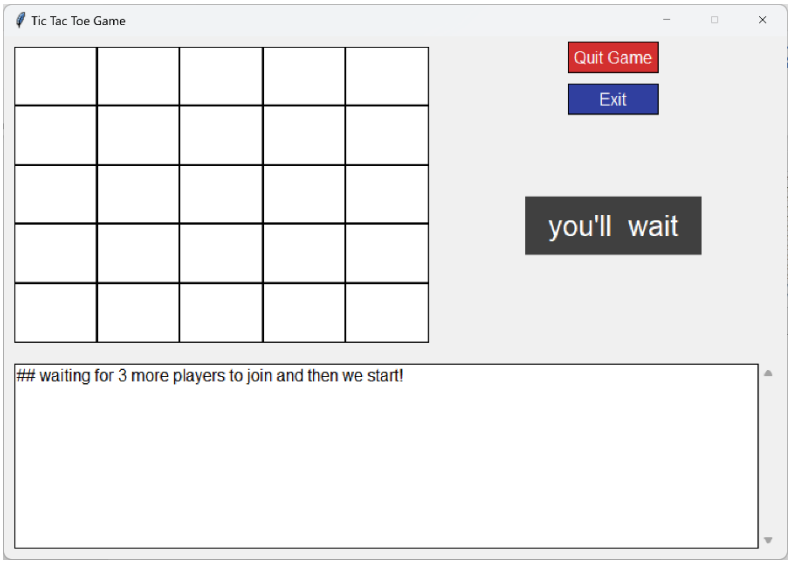
user name	number of games	won	lost	draw
aviad	1	0	1	0
noam	4	0	2	2
moshe	1	0	1	0
avraham	10	4	5	1
orna	12	6	5	1
dvir	23	6	11	6
eitan	0	0	0	0
tal	0	0	0	0
efrat	28	9	13	6
avishai	5	2	3	0



#	date	duration	number of players	status	winner
1	07/04/2024	0:00:21	2	victory	dvir
2	07/04/2024	0:03:06	8	victory	avraham
3	08/04/2024	0:01:29	3	victory	orna
4	08/04/2024	0:00:12	2	victory	avraham
5	08/04/2024	0:00:13	2	victory	dvir
6	09/04/2024	0:00:42	2	victory	efrat
7	09/04/2024	0:00:19	2	victory	efrat
8	10/04/2024	0:00:13	2	victory	efrat
9	10/04/2024	0:02:37	3	victory	orna
10	10/04/2024	0:00:47	2	victory	efrat
11	11/04/2024	0:01:33	2	draw	-
12	17/04/2024	0:03:57	4	draw	-

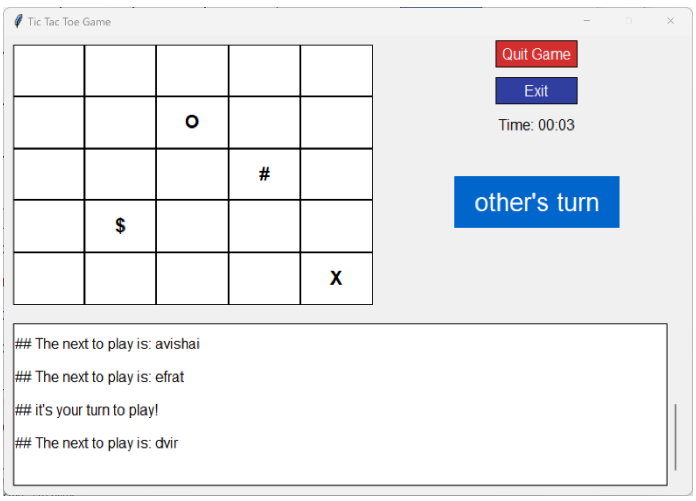
בס"ד

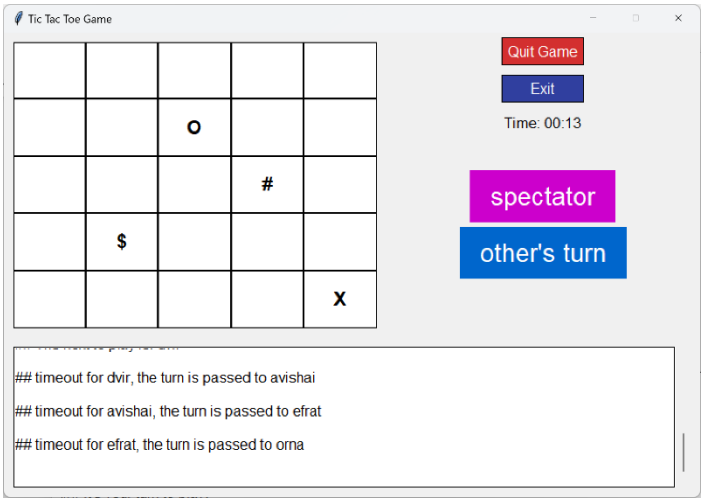
בלחיצה על New Game כאמור תופיע תיבת טקסט להזנת מספר משתתפים, ולאחר מכן ניתן ללחות על אותו כפתור שוב (יהיה כתוב עליו !Start) ולהמתין לשחקנים נוספים שיצטרפו עד להגעה למס' המשתתפים שהוצהר עליהם בתחילת המשחק. נראה את המסך הבא:



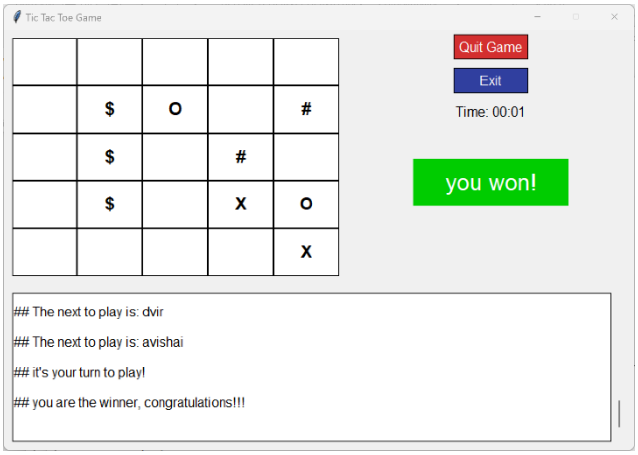
כאשר מספיק משתתפים מצטרפים, המשחק מתחיל וזה נראה כך (הסמל של כל משתתף נשלח אליו בהודעה בתיבת ההודעות למטה, כמו גם כל מיני הודעות נוספות):

(חלון צופה מימין, חלון שחקן משמאל)





במקרה של ניצחון או תיקו השעון ייעצר ויראה החלון הבא (במקרה זה מדובר בניצחון):



תשובות לשאלות תיאורטיות

1. security

a. במצב הנוכחי של האפליקציה, הזדהות מתבצעת באמצעות הקשת קוד סודי אשר מנופק לכל משתמש בעת הירשמו למערכת. קוד זה נשמר בקובץ JSON פשוט אליו ניתן לגשת דרך מנהל הקבצים. לפיכך, דרך פשוטה אחת להשיג קוד סודי של משתמש אחר הינה קריאה של אחד מקבצי JSON בתיקיית הפרויקט, וכך ניתן למעשה "לפרוץ" למערכת. בנוסף, במהלך ההזדהות איני מבצע בדיקת תקינות על קלט הסיסמא וכך ניתן להשתמש ב-code injection ולגרום למערכת לחשוב כי הפורץ הינו משתמש רשום.

ניתן לשדרג את ההגנה ע"י הצפנת קבצי הטקסט המהווים את מסד הנתונים של היישום, ובנוסף יש לבצע בדיקת תקינות על הקלט וכן ערבול שלו ע"י hashing מה שיוצר הפרדה בין המשתמש ובין הגישה למסד הנתונים.

b. כאשר שולחים מידע ברחבי הרשת כ-plain text ישנם סיכונים לא מעטים. תחילה, באופן שכזה השולח והמקבל עלולים לחשוף עצמם למתקפה המוכרת בשם "man in the middle", מתקפה בה מחשב אחד ברשת מזדהה כשרת אף שהוא איננו כזה ובתוך כך מנתב דרכו את כל ההודעות ליעדן. אם ההודעות אינן מוצפנות הרי שהתוקף יהיה מסוגל לקרוא את תוכנן ללא כל קושי. ישנן מתקפות נוספות להן יהיה נתון השולח את הודעותיו ללא כל הצפנה.

ניתן לשפר מעט את המצב באמצעות הצפנת המידע. ישנן שיטות שונות להצפנה, באפליקציה שלי יישמתי הצפנה סימטרית.

2. scalability

a. כדי לטפל במספר גדול של משתמשים במקביל הייתי עושה שימוש בספרייה asyncio אשר נותנת כלים לתכנות מסוג Asynchronous Programming. תכנות מסוג זה מאפשר פעולות non-blocking I/O, ובכך מאפשר לשרת לטפל במספר גדול של משתמשים ללא צורך לחכות לכל פעולה שתסתיים כדי לעבור למשתמש הבא. בכך גם יוצרים responsive feeling אצל המשתמשים. עם זאת, השימוש בכלים שמציעה הספרייה הנ"ל דורשת תשומת לב מרובה באשר לגישה למשאבים משותפים בכל הנוגע לסינכרון תהליכים וכדו'. אשר ל-bottlenecks בפיתוח אפליקציות מרובות משתמשים ממוקבלים, ניתן למנות את ה-I/O operations וכן פעולות CPU מורכבות (לא קיימות כל כך ביישום שלנו). בשתי הדוגמאות הנ"ל תהליכים ימתינו זמן רב יחסית לסיום הפעולה ואף שישנה תחלופה בין התהליכים וכל אחד מקבל slot משלו, התהליכים עלולים להצטבר בתור המתנה לסיום הפעולות וכך ליצור צוואר בקבוק באפליקציה.

b. ניתן ליישם אסטרטגיות שונות לטיפול במקרים של צוואר בקבוק. למשל, ניתן לעקוב אחר מס' התהליכים הממתינים לסיום פעולה מסוימת ובמקרה של התהוות צוואר בקבוק יש להשהות פעולות קצרות אחרות ולהפנות יותר משאבי CPU או streams של data למתן שירות מהיר יותר וחיסול צוואר הבקבוק, על חשבון המתנה מעט ארוכה יותר עבור פעולות אחרות.

3. Reliability and Fault Tolerance

a. תחילה, על מנת לשפר את ה-reliability של האפליקציה עשיתי שימוש בספריות סטנדרטיות בלבד אשר הסיכוי כי הקוד שקיים בהן יקריס את האפליקציה שלי הינו נמוך מראש. בנוסף, שגיאה כלשהי בצד שרת לא תעיב על הביצועים בצד לקוח במידה ניכרת משום שהשרת רץ בלולאה אינסופית וכל תקלה אצלו "נתפסת" בחכת ה-Exception וכך הוא ממשיך לרוץ. במקרים חמורים בהם השרת נופל לחלוטין, הלקוח מקבל הודעה מתאימה המתריעה בפניו על אובדן הקשר עם השרת ועל כך שעליו ליצור את הקשר מחדש בשנית. זאת ועוד, המידע נשמר בצד השרת בצורה שאינה נדיפה כך ששגיאה או נפילה של השרת לא גורמת לאובדן מידע חמור.

b. ניתן לממש תצורה של message persistence ע"י הכנסה של ההודעות הנכנסות ל-persistent buffer, באמצעות ענן או קובץ מידע ב-cache אשר קריאתו תהיה זמינה ומהירה. כך, אפילו אם השרת נופל ומאתחל עצמו מחדש, ההודעות אינן אובדות.

4. User Authentication

a. החשיבות הנודעת ל-user authentication באפליקציות מהסוג שאני בניתי ניכרת במידע הנשמר עבור כל משתמש ומשתמש. לכל משתמש רשומה על שמו במסד הנתונים אשר שומרת נתונים אישיים עליו ועל אופן תפקודו במהלך המשחקים בהם השתתף. אילולי מערכת ההזדהות, היה כל אחד יכול להיכנס ולפגוע ברישומים אודות שחקנים אחרים על מנת לקדם את עצמו או חמור מכך, לשאוב נתונים אישיים הזמינים לכל אחד בחשבון האישי שלו.

על מנת לזהות משתמש באופן בטוח ניתן לנקוט במספר שיטות:

1. שם משתמש וסיסמא.
2. הזדהות מרובת שלבים (תוך שימוש בקוד זמני).
3. שימוש ב-digital certificate
4. הזדהות מבוססת טוקנים (הנפקת קוד לאחר הכנסת פרטי ההזדהות בפעם הראשונה אשר משמש להזדהות בפעמים הבאות ללא צורך בהכנסת הפרטים המזהים בשנית).

b. באופן אישי הייתי שומר בסיס נתונים מוצפן של שמות משתמש וסיסמאות. סיסמא שמשתמש יזין למערכת תעבור תהליך בדיקת תקינות (וידוא כי לא מתחבאים תווים חשודים או שאורך המחרוזת לא חריג) ומשם יבוצע hashing למחרוזת שם המשתמש והסיסמא ותתנהל השוואה למול בסיס הנתונים. כל התהליך יתבצע כמובן בצד השרת ולא יישלחו נתונים רגישים לצד לקוח.

5. Concurrency and Synchronization

a. ב-server מרובה תהליכים המפתח עשוי להיתקל בקשיים מסוגים שונים. תחילה, ללא קשר ללקוחות בצד השני של הקשר, יש לתת את הדעת על כל נושא סינכרון תהליכים אשר כולל בתוכו בעיות קטע קריטי ומשאבים משותפים, ועוד. בנוסף, כיוון שהיישום מרובה התהליכים הוא server, הודעות צריכות להישלח ללקוחות ומכיוון שיש מספר תהליכים במערכת תיתכן דריסה של הודעות.

b. ניתן להתגבר על הבעיות המנויות לעיל באמצעות שימוש זהיר במשאבים המשותפים (הגנה ויישום mutual exclusion בקטע הקריטי) ובאמצעות buffer של הודעות במקום משתנה יחיד להודעות אשר עלול להידרס.

6. Protocol Design

a. הפרוטוקול בו בחרתי להשתמש במסגרת הפרויקט הינו TCP ולא UDP. בחרתי בפרוטוקול זה משום שהוא מבטיח את שלמות המידע המגיע ללקוח ואת הסדר הנכון שלו, מאפיין חשוב כאשר מדובר ביישום שבו הפרטים חשובים, בשונה למשל מיישום של שידור וידאו וכדומה אשר איבוד של frame אחד פעם בכמה פקטות לא יסב תשומת לב רבה מדי.

b. אסביר כעת כיצד מתבטא ה-socket communication אצלי בפרויקט.

שורה ראשונה:

```
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

בפקודה זו יצרתי socket מהספרייה socket והפרמטרים שנתתי לו הם: socket.AF_INET, socket.SOCK_STREAM. הפרמטר הראשון מנחה את ה-socket להשתמש בפרוטוקול IPv4 (ניתן היה

להשתמש בפרוטוקול IPv6 אך לא היה בכך צורך). הפרמטר השני שסופק ל-socket הנחה אותו להשתמש בפרוטוקול TCP. כלומר, נבחרו שני פרוטוקולים עבור שתי השכבות Network & Transport.

הפקודה הבאה היא:

```
lsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

בפקודה זו רציתי להימנע מ-Exception שעולה במקרה של סגירת connection ופתיחתו מחדש לפני תום פרק זמן מסוים שקבעה מערכת ההפעלה.

לאחר מכן:

```
lsock.bind(addr)
lsock.listen()
```

קשירת ה-socket לכתובת localhost 127.0.0.1 בפורט 65432 שנקבע מראש ותחילת האזנה לפורט הזה.

הפקודה הבאה:

```
conn, addr = lsock.accept()  # Should be ready to read (blocking command)
```

פקודה זו היא מסוג blocking command כאשר המשמעות היא שהקוד לא ממשיך לרוץ אא"כ התקבל חיבור חדש או שאירעה תקלה. זו בדיוק ההתנהגות שאנו רוצים: ללא חיבור חדש, ה-server לא יבצע פעילות כלשהי (בתהליך הראשי).

לאחר מכן (משמע, התקבל חיבור חדש), החיבור החדש שהתקבל נקבע להיות מסוג שאיננו חוסם כדי שהשרת ימשיך לתפקד במקביל לכל התהליכים שרצים בנפרד.

לבסוף,

Thread / selector

כעת ניגש להרכב ההודעות הנשלחות ול-headers שנקבע שייעשה בהם שימוש. הוחלט כי כל הודעה תהיה מורכבת מ-3 חלקים: אורך ה-header, header, message.

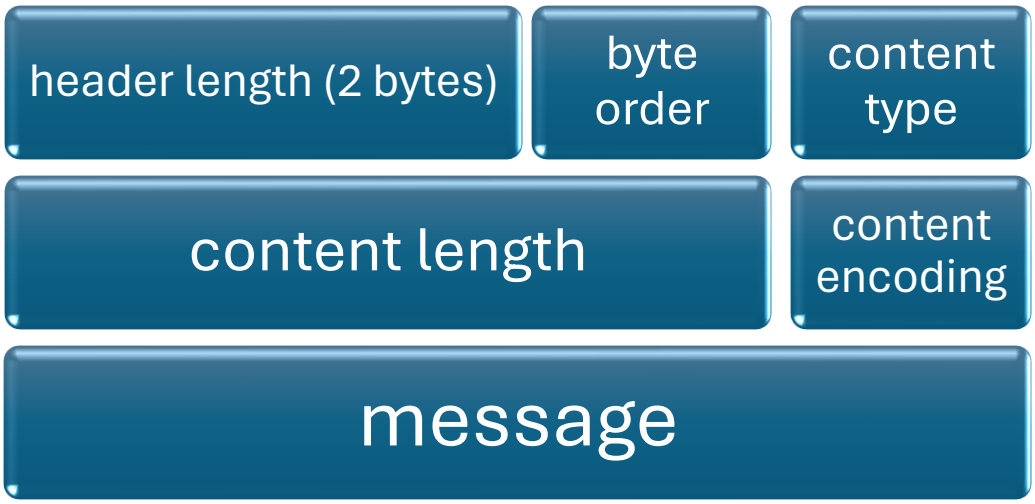
החלק הראשון אורכו קבוע – 2 בתים. לאחר שהוא נקרא, ניתן לדעת כמה צריך עוד לקרוא בשביל לקבל את כל ה-header. בתוך ה-header יש 4 שדות:

- byte order (little endian or big endian)
- content type (type of content, either binary or text/json)
- content encoding (utf-8 or other)

- content length (how many bytes the message without the header is).

השדות האלו משמשים לפענוח הודעה בצד המקבל, ובפרט השדה שנקרא content length חשוב משום שבאמצעותו ניתן לדעת כמה ביטים לקרוא מתוך buffer הקבלה.

(הדיאגרמה למטה מייצגת הודעה שנשלחת על כל חלקיה, אף שהיחס בין גודל כל חלק בדיאגרמה אינו מייצג נאמנה את המציאות).



ההודעה עצמה (message) אף היא מורכבת משני חלקים: כותרת ותוכן. עפ"י הכותרת ניתן לדעת באופן כללי באיזה סוג הודעה מדובר, וכך לטפל בתוכן באופן מתאים.

7. Message Ordering and Delivery

a. באפליקציה שלי הכנתי buffer של הודעות לשליחה. Buffer דומה לקבלה לא היה צריך, כפי שאסביר מיד. הצורך ב-buffer של הודעות לשליחה עלה כאשר צץ הצורך לשלוח מספר הודעות בזו אחר זו לאותו לקוח, או לחילופין מספר הודעות תכופות לשרת. על מנת שההודעות יישלחו בסדר הרצוי, הן נכנסו ל-buffer אשר ממנו נמשכה כל פעם הודעה אחת בלבד, ורק כאשר היא טופלה נמשכה הודעה חדשה. Buffer דומה עבור קבלה לא צריך משום שבמקרה זה ממילא מושכים מס' ביטים בגודל של ההודעה הראשונה ורק אז מתפנים למשוך הודעה נוספת מזרם הביטים של ה-socket.

b. כאמור, שימוש ב-buffer באופן זהיר וכן שמירה על כללי השימוש במשאב משותף בין כמה תהליכים עשוי להבטיח זרם הודעות אמין ומסודר.

8. Persistent Storage

c. הצורך באמצעי אחסון שאיננו נדיף אלא persistent מובן למדי. במקרה של תקלה כמו למשל אם השרת נופל, היינו רוצים להבטיח שימור של המידע שנצבר בשרת כל שכאשר השרת יועלה מחד, כל המידע יחזור ויהיה ניתן להשתמש בו. באפליקציה שלי השתמשתי בקבצי JSON. זהו אמצעי שאיננו נדיף ואף די פשוט לשימוש.

d. ישנם סוגי פתרונות שונים לסוגיית אחסון המידע. ניתן לשמור מידע בקבצים או לחילופין במסד נתונים מסודר. את שני הפתרונות האלו ניתן ליישם בזיכרון מקומי או בשירותי ענן. היתרונות בשימוש בזיכרון מקומי הינם קרבת המידע אל האפליקציה (לא צריך לעבור באינטרנט בשביל לקבל את המידע) וחינמיות השימוש, בעוד שבשירותי ענן המידע די רחוק והשימוש בהם עלול לעלות כסף. מאידך, שימוש בשירותי ענן מבטיח גישה מכל מקום בעולם (בו יש גישה אל הענן). שימוש בקבצים עשוי להיות פשוט וזול אך לעיתים יותר מסורבל מאשר שימוש במסדי נתונים עליהם ניתן לבצע שאילתות מורכבות וכך לחלץ מידע מורכב במהירות וביעילות.