21.6.19

ספר פרויקט - אלגוריתם יציב ואמין לבחירת מנהיג

מגישות: שירה מנדלבום ואפרת צדוק
מנחה: חנן רוזמרין

חלקי הפרויקט ותוצריו:

- בשלב הראשון, למדנו לעומק את שפת GO.

נעזרנו במקורות הבאים:
- https://www.udemy.com/goprogramming/
- https://www.lynda.com/Go-tutorials/Go-Web-Development/516402-2.html
- https://www.lynda.com/Go-tutorials/Go-Essential-Training/748576-2.html
- https://golang.org/
- https://www.freecodecamp.org/news/learning-go-from-zero-to-hero-d2a3223b3d86/

- בשלב השני, בנינו סימולציית sum – פרוטוקול סכום עם n שחקנים אשר בסימולציה, מנהל המשחק מקבל כפרמטר את מס' השחקנים ויוצר אותם. במהלך המשחק השחקנים שולחים הודעות לchannels של שאר השחקנים ולאחר מכן, מודפס למסך סכום המס' שנשלחו לchannel של כל שחקן. המטרה – המס' יהיו זהים מכיוון שכל ההודעות הגיעו. בשלב זה התחלנו לכתוב טסטים ע"פ הframework המתאים.

נעזרנו במקור הבא: http://onsi.github.io/ginkgo/#getting-ginkgo
קישור לקוד של שלבים 1-2: https://github.com/efi411/final_project1

- בשלב השלישי, יצרנו דימוי של איבוד הודעות ע"פ הסתברות p. מנהל המשחק קיבל פרמטר נוסף עבור ההסתברות. עטפנו את הchannel המובנה של GO במחלקה משלנו ובכל שליחת הודעה משחקן אל הchannel, ביצענו סינון ע"פ הסתברות p של שליחת ההודעה אל הchannel בתוך המחלקה עצמה. במצב זה ההודעה בוודאות נשלחה מן השחקן אך לא בטוח שהגיעה אל יעדה. התוצאה הייתה איבוד הודעות עבור חלק מהchannels והדפסת סכומים שונים. השונות בין הסכומים שהודפסו הייתה תלויה בהסתברות p. כלומר, כאשר ההסתברות הייתה גבוהה, הלכו לאיבוד פחות הודעות והודפסו סכומים כמעט

זהים ולהיפך כאשר ההסתברות p הייתה נמוכה. בשלב זה הרחבנו את הטסטים עבור כל המחלקות והגענו ל100% coverage.

נעזרנו במקור הבא: https://blog.golang.org/cover
קישור לקוד של שלב 3: https://github.com/efi411/final_project2

- בשלב הרביעי, קראנו את המאמר לעומק. הבנו את הצורך בשימוש באלגוריתם של 'בחירת מנהיג' בסביבה מרובת מעבדים. לאחר מכן קראנו על אלגוריתמים שונים המספקים מענה לצורך זה, הבנו את יתרונותיהם וחסרונותיהם ובחנו את האלגוריתם המוצע במאמר שלנו ביחס לאלגוריתמים שמומשו עד היום.
האלגוריתם המתואר במאמר הורכב בשלבים, כאשר כל אחד מהאלגוריתמים שמתואר בכל אחד מהשלבים ממש פונקציונליות נוספת לאלגוריתם שבא לפניו. לכן, החלטנו לאחר התייעצות עם מנחה הפרויקט לממש בתחילה את האלגוריתם הראשון המתואר במאמר.

נעזרנו במקורות הבאים:
- https://en.wikipedia.org/wiki/Leader_election
- https://stackoverflow.com/questions/16055973/distributed-system-leader-election
- https://www.geeksforgeeks.org/election-algorithm-and-distributed-processing/

- בשלב החמישי, מימשנו את האלגוריתם הראשון במאמר והתחלנו ליצור טסטים בהם התייחסנו לתרחישים אותם רצינו לבדוק - כמות משתנה של משתתפים, נפילת משתתפים, תרחיש ללא שחקנים וכו'...

פסאודו קוד לאלגוריתם הראשון:

Algorithm 1. Communication-Efficient Self-Stabilizing Leader Election on $\mathcal{S}_5$

```
CODE FOR EACH PROCESS p:
variables:
    Leader_p ∈ {1,...,n}
    SendTimer_p, ReceiveTimer_p: non-negative integers

repeat forever
    for all q ∈ V \ {p} do
        if receive(ALIVE) from q then
            if (Leader_p ≠ p) ∨ (q < p) then
                Leader_p ← q
            end if
            ReceiveTimer_p ← 0
        end if
    end for
    SendTimer_p ← SendTimer_p + 1
    if SendTimer_p ≥ ⌊δ/β⌋ then
        if Leader_p = p then
            send(ALIVE) to every process except p
        end if
        SendTimer_p ← 0
    end if
    ReceiveTimer_p ← ReceiveTimer_p + 1
    if ReceiveTimer_p > 8⌈δ/α⌉ then
        if Leader_p ≠ p then
            Leader_p ← p
        end if
        ReceiveTimer_p ← 0
    end if
end repeat
```

- בשלב השישי, בנינו את האלגוריתם השני במאמר על גבי האלגוריתם הראשון והרכבנו על הפרוטוקול שרת בGO שמתקשר עם לקוח בjavascript.

פסאודו קוד לאלגוריתם השני:

Algorithm 2. Communication-Efficient Pseudo-Stabilizing Leader Election on $\mathcal{S}_4$

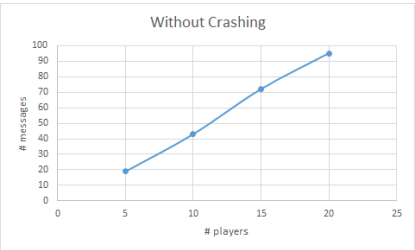
```
CODE FOR EACH PROCESS p:
variables:
    Leader_p ∈ {1,...,n}
    SendTimer_p, ReceiveTimer_p, Round_p: non-negative integers

procedure StartRound(s)
    if p ≠ (s mod n + 1) then
        send(START,s) to s mod n + 1
    end if
    Round_p ← s
    SendTimer_p ← ⌊δ/β⌋
end procedure

repeat forever
    for all q ∈ V \ {p} do
        if receive (ALIVE,k) or (START,k) from q then
            if Round_p > k then
                send(START,Round_p) to q
            else
                if Round_p < k then
                    StartRound(k)
                end if
                ReceiveTimer_p ← 0
            end if
        end if
    end for
    ReceiveTimer_p ← ReceiveTimer_p + 1
    if ReceiveTimer_p > 8⌈δ/α⌉ then
        if p ≠ (Round_p mod n + 1) then
            StartRound(Round_p + 1)
        end if
        ReceiveTimer_p ← 0
    end if
    SendTimer_p ← SendTimer_p + 1
    if SendTimer_p ≥ ⌊δ/β⌋ then
        if p = (Round_p mod n + 1) then
            send(ALIVE,Round_p) to every process except p
        end if
        Leader_p ← (Round_p mod n + 1)
        SendTimer_p ← 0
    end if
end repeat
```

- בשלב השביעי, יצרנו UI בHTML הממחיש את תהליך האלגוריתם למשתמש ע"י הכנסת נתונים מהמשתמש (כגון: מס' שחקנים, נפילות וכו'...) והדפסת ההודעות הנשלחות במהלך האלגוריתם ע"י השחקנים והתוצאה הסופית - המנהיג. בנוסף, הוספנו טסטים לאלגוריתם השני.

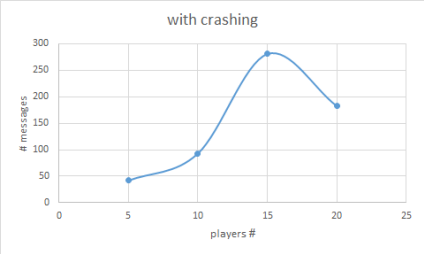
קישור לקוד של שלבים 4-7 (התוצר הסופי):
https://github.com/efi411/final_project3

- בשלב השמיני, בנינו גרפים המשקפים **תחזיות** על מס' הודעות בין שחקנים כאשר יש/אין קריסות במערכת.

גרף תחזית ללא קריסה:



גרף תחזית עם קריסה:



- בשלב התשיעי, בנינו דיאגרמות המשקפות את חלקי המערכת והflow של הריצה.

נעזרנו במקור הבא: https://online.visual-paradigm.com/

- בשלב האחרון, הוספנו פונקציות המריצות את האלגוריתם עם נתוני מס' שחקנים שונים (בין 5 ל120) ומחזירות תוצאות זמן ריצה ומס' ההודעות שנשלחו בין השחקנים בזמן הריצה. בדקנו מה מגמת האלגוריתם ביחס לזמן הריצה וכמות ההודעות הנשלחות בין השחקנים ובנינו מהתוצאות גרפים המשקפים בצורה מדויקת את היחסים.

נעזרנו במקורות הבאים:
- https://www.canva.com/graphs/
- https://www.rapidtables.com/tools/line-graph.html

על האלגוריתם:

מערכות מבוזרות הן מערכות שבהן יש הרבה תהליכים שמתקשרים זה עם זה דרך חיבורים. כל תהליך במערכת זו רץ בסבבים, כאשר בכל סבב הוא יכול לשלוח/לקבל הודעה ולשנות את הסבב שבו הוא נמצא.

במערכות אלו, יש שגיאות שנובעות ממוות של תהליך (crash failure או שגיאת קריסה) ויש כאלו שנובעות מהפרעה זמנית (transient failure או שגיאות חולפות). במאמר מחפשים אלגוריתמים לבחירת מנהיג (leader election) באותן מערכות. המאמר משלב בין שתי גישות- יציבות וחוזק. ע"פ הכתוב במאמר, האלגוריתמים הקיימים לבחירת מנהיג, עד כה, לא שילבו בין שתי הגישות- אלה שהיו יציבים לא היו חזקים ולהיפך. לכן, האלגוריתמים המתוארים במאמר הם כאלה שהם גם יציבים (כלומר, מתכנסים למצב טוב, בו, מעבר במצבים "לא-טובים" אינו משפיע על המצב הטוב של האלגוריתם, כלומר, על הפעילות התקינה של המערכת) ולכן טובים בשגיאות חולפות וגם חזקים (robust), ולכן טובים בשגיאות קריסה.

האלגוריתם השני המתואר במאמר, הוא האלגוריתם אותו אנחנו מימשנו.
כפי שמוזכר לעיל, כל תהליך רץ בסבב מסוים. ע"פ הגדרות האלגוריתם במאמר, כל סבב, המנהיג הוא זה שהid שלו שווה לתוצאת המשוואה הבאה- $s \bmod n + 1$, כאשר s הוא מציין את מספר הסבב וn מציין את מספר השחקנים (בהינתן שהם מתחילים מ1).
האלגוריתם מתרחש בתוך לולאה שרצה תמידית:

1. בדוק אם אתה מקבל הודעה מאחד התהליכים האחרים.
2. אם כן,
   - במידה והסבב בו השולח נמצא קטן מהסבב בו אתה נמצא, שלח הודעת START (בצירוף מספר הסבב שלך) לתהליך ממנו קיבלת את ההודעה.
   - במידה והסבב בו השולח נמצא גדול מהסבב בו אתה נמצא:
     אם אתה לא המנהיג של אותו הסבב, שלח הודעת START (בצירוף מספר הסבב שקיבל מהשולח) למנהיג של אותו הסבב, עדכן את הסבב שלך למספר אותו קיבלת ואתחל את טיימר שליחת ההודעה לחסם הקבוע.
   - אפס את טיימר קבלת ההודעות.
3. הגדל את טיימר קבלת ההודעות ב1.
4. אם לא קיבלת הודעה יותר מזמן, החסם הקבוע פי שמונה:
   - אם אתה לא המנהיג של הסבב הנוכחי, התחל סבב חדש עם המספר העוקב לסבב הנוכחי, שלח הודעת START (בצירוף מספר הסבב החדש) למנהיג של הסבב החדש (במידה וזה לא אתה), אתחל

את הסבב שלך למספר החדש ואפס את טיימר שליחת ההודעה לחסם הקבוע.

- אפס את טיימר קבלת ההודעות.

5. אם לא שלחת הודעות יותר מזמן החסם הקבוע:

- אם אתה המנהיג של אותו הסבב, שלח את המספר שלח הודעת ALIVE לכל שאר התהליכים.
- אתחל את המשתנה בו נשמר המנהיג למנהיג של הסבב הנוכחי.
- אפס את הטיימר של שליחת ההודעות.

כיצד אלגוריתם זה מביא לידי מימוש את שתי הגישות אותן הזכרנו בתחילה?

בשל קבועי הזמן שאותם מגדירים באלגוריתם, כלל התהליכים מקבלים באופן עקבי, סימני חיים מהתהליך שנבחר להיות המנהיג. במידה ובתוך פרק זמן מסוים, הם לא קיבלו סימן חיים מאותו המנהיג, הם מתקדמים לסבב הבא ו"ממנים" מנהיג חדש.
קבוע הזמן יוגדר כך, שבמידה ובמהלכו תהליך המנהיג קרס, לא יגרם נזק למערכת.
על כן, בחירת המנהיג החדש תתבצע בזמן שאינו יגרור אחריו כשלים במערכת ועל כן, הנפילה של אותו תהליך אינה מורגשת.

בנוסף, במידה ותהליכים מסוימים רצים בסבב אחר ומחזיקים במנהיג שונה, ברגע שהם יקבלו הודעת start מתהליך, שהסבב שלו מעודכן וגדול משלהם, הם יעברו באופן ישיר לאותו הסבב ויישרו קו מול שאר התהליכים בהגדרת התהליך המנהיג, בסופו של דבר, בדרך זו, כל התהליכים יסכימו על מנהיג אחד.
נראה כי בדרך זו האלגוריתם יודע להתמודד גם עם קריסות של תהליכים - חוזקה וגם לצאת ממצבים לא טובים, בלי הפרעה- יציבות.

חלקי המערכת:

- צד השרת – ריצת האלגוריתם לבחירת מנהיג:
מימוש באמצעות: GO.

דימינו עבודה של מערכת מבוזרת, בה רצים מספר תהליכים באופן מקבילי ועליהם לבחור מנהיג משותף.

כיוון שהפרויקט הוא פרויקט דמה, הכנסנו פרמטרים שונים שייצרו עבורנו סביבה הדומה למציאות ככל שניתן.

לצורך מימוש צד השרת, יצרנו שני אובייקטים-
אובייקט שחקן (Player)-
אובייקט זה מדמה תהליך במציאות אמת, בעל ID ייחודי, מחזיק בערוץ בעל ID זהה לID שלו וכן, מחזיק בערוצים איתם ניתן לגשת לשאר השחקנים שרצים במקביל אליו. השחקן יודע את כמות השחקנים המשתתפת בריצה.
באובייקט השחקן, מתבצעת ריצת האלגוריתם לבחירת מנהיג.
אובייקט ערוץ (Channel)-
מדמה את הבאפר שמחזיקים התהליכים המתוארים במאמר, לצורך שליחת וקבלת הודעות. בנוסף, כיוון שהסבב בו נמצא שחקן הינו חלק מההודעה שנשלחת, אובייקט הערוץ מחזיק אצלו גם את השדה שמכיל את הסבב בו נמצא השחקן ומשתנה במהלך ריצת האלגוריתם.

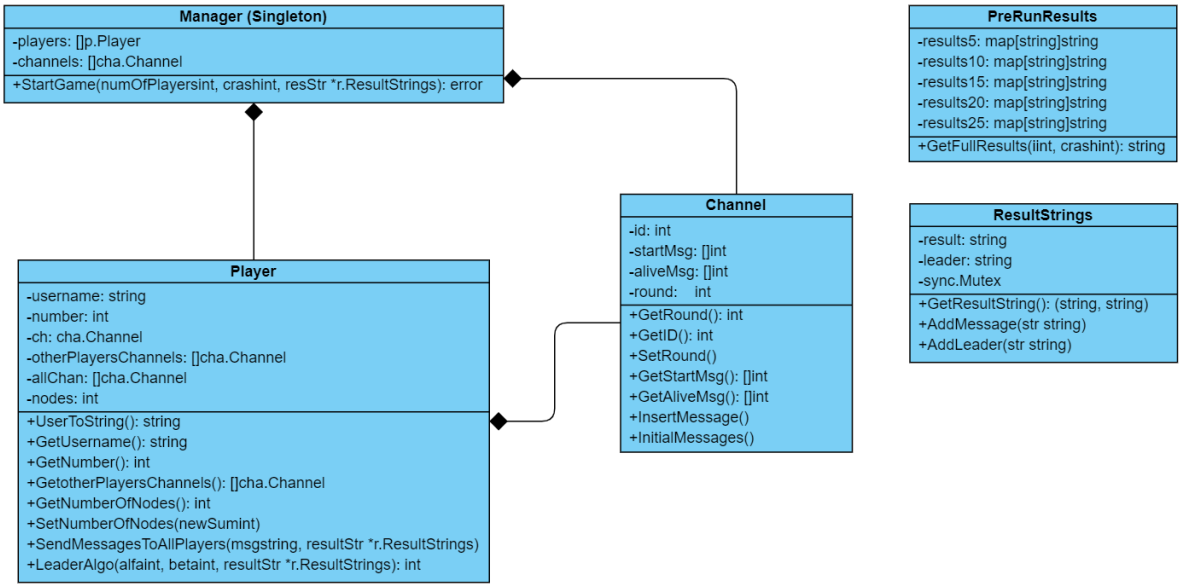
השרת מקבל 2 פרמטרים- האם תתקיים קריסה במהלך ריצת האלגוריתם או לא ומס' התהליכים ("השחקנים") שישתתפו באותה הריצה.
הפרמטרים נשלחים למחלקת הManager שיצרנו.
מחלקה זו, הייתה אחראית על ייצור האובייקטים של השחקנים, ייצור האובייקטים של הערוצים וחיבור לשחקנים בהתאמה וכן, על הרצת האלגוריתם באופן מקבילי אצל כל אחד מהאובייקטים של השחקנים שנוצרו.

לצורך בחינת היציבות של האלגוריתם, יצרנו כל שחקן בסבב אחר (ומתוך כך, לכל שחקן בתחילת האלגוריתם היה גם מנהיג אחר).
לצורך בחינת החוזק של האלגוריתם, הגדרנו ריצות בהן "הרגנו" את השחקן המנהיג ונתנו לאלגוריתם להמשיך לרוץ.

כיוון שרצינו לבצע בדיקה של ריצת האלגוריתם ולהציג תוצאות, לא ניתן היה לאפשר את ריצת האלגוריתם באופן תמידי (כמו במערכת אמיתית - כפי שמתואר במאמר). לכן, הגדרנו מספר מסוים של ריצות בהן ירוץ האלגוריתם. המספר התקבל לאחר ביצוע מספר הרצות של האלגוריתם עם מספרים שונים וניסיון לדמות מציאות ככל שניתן. את קבועי הזמן שהגדרנו לטיימרים של שליחת וקבלת ההודעות, המרנו למספר ריצה מסוים של האלגוריתם.

כל שחקן, מחזיק בתוך האלגוריתם משתנה המייצג את המנהיג שלו ע"י הID של אותו המנהיג. לאחר שמספר הריצות שהגדרנו נגמר, מחזיר כל שחקן לאובייקט ה"מנהל" את מספר המשתנה של המנהיג שלו, כך, שבסופו של דבר, ע"פ הכתוב במאמר (ובסופו של דבר, גם ע"פ התוצאות שקיבלנו) כלל השחקנים יחזירו את אותו מנהיג, הן במקרה הרגיל (הבוחן יציבות בלבד) והן במקרה הקריסה (הבוחן יציבות וחוזק יחד).

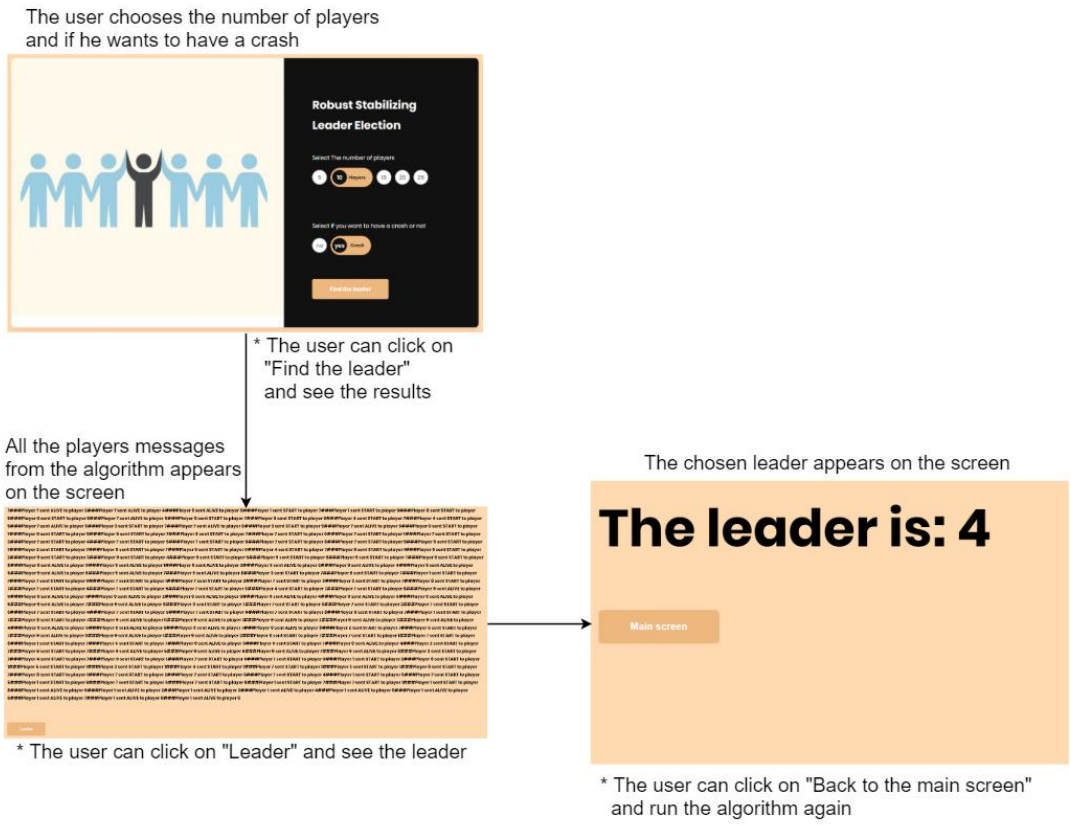
דיאגרמת UML של חלקי המערכת בשרת:



- צד הלקוח - תצוגת תוצאות האלגוריתם למשתמש:
מימוש באמצעות: Javascript, HTML5, CSS.

באמצעות אפליקציית web פשוטה לתפעול, המשתמש יכול לראות את תוצאות ריצת האלגוריתם על כמות שחקנים מצומצמת – 5, 10, 15, 20, 25 וניתנת לו גם היכולת לבחור האם הוא רוצה שתתבצע קריסה בזמן האלגוריתם או לא.
במסך הראשון, המשתמש מכניס את הפרטים הנ"ל ועובר למסך השני לתוצאות.
במסך השני, המשתמש רואה את כל ההודעות הנשלחות בזמן האלגוריתם. פרטי ההודעות הם: מי השולח, למי ההודעה נשלחה ומהי ההודעה שנשלחה (START/ALIVE). המשתמש יכול ללחוץ על הכפתור בתחתית המסך ולהגיע אל המסך השלישי.
במסך השלישי, המשתמש רואה בגדול את תוצאת האלגוריתם: המנהיג שנבחר. למשתמש יש אפשרות לחזור אל המסך הראשון על מנת להריץ את האלגוריתם שוב, גם עם נתונים שונים.

דיאגרמה שממחישה את האפליקציה - תיק מסכים:



- האינטגרציה בין השרת ללקוח:

מימוש באמצעות: קריאות REST API בין הלקוח בjavascript אל השרת בGO.

בצד השרת, השתמשנו בספרייה - "github.com/gin-gonic/gin" על מנת ליצור שרת שמקבל בקשות ומטפל בהן ובספרייה - "github.com/gin-contrib/cors" על מנת להימנע מחסימות אבטחה בין הדומיינים (פרטים נוספים ב"אתגרים ופתרונות" בהמשך...).

```
r := gin.Default()
// Use cors for go server
r.Use(cors.Default())
projectAPI := r.Group("/project")
projectAPI.GET("/", handleRequest)
r.Run()
```

כאשר המשתמש לוחץ על הכפתור "Find the leader", הלקוח שולח בקשה לשרת עם פרטי האלגוריתם הנדרשים שהוכנסו ע"י המשתמש: מס' שחקנים (5/10/15/20/25) והאם תהיה קריסה במערכת (0/1).

```
request.open('GET',"//localhost:8080/project/?numOfPlayers=" + numOfPlayers +
"&crashes="+crashes, true);
request.setRequestHeader("Content-Type", "text/plain;charset=UTF-8");
request.send('');
```

השרת מקבל את הבקשה, מפרסר אותה על מנת לקבל את הנתונים ושולף מהאובייקט – preRunResults את התוצאות המתאימות ע"פ הנתונים שהוזנו לו. התוצאות נשלחות אל הלקוח בצורת string המכיל את המנהיג ואת כל ההודעות שנשלחו במהלך האלגוריתם ע"י השחקנים אחד לשני.

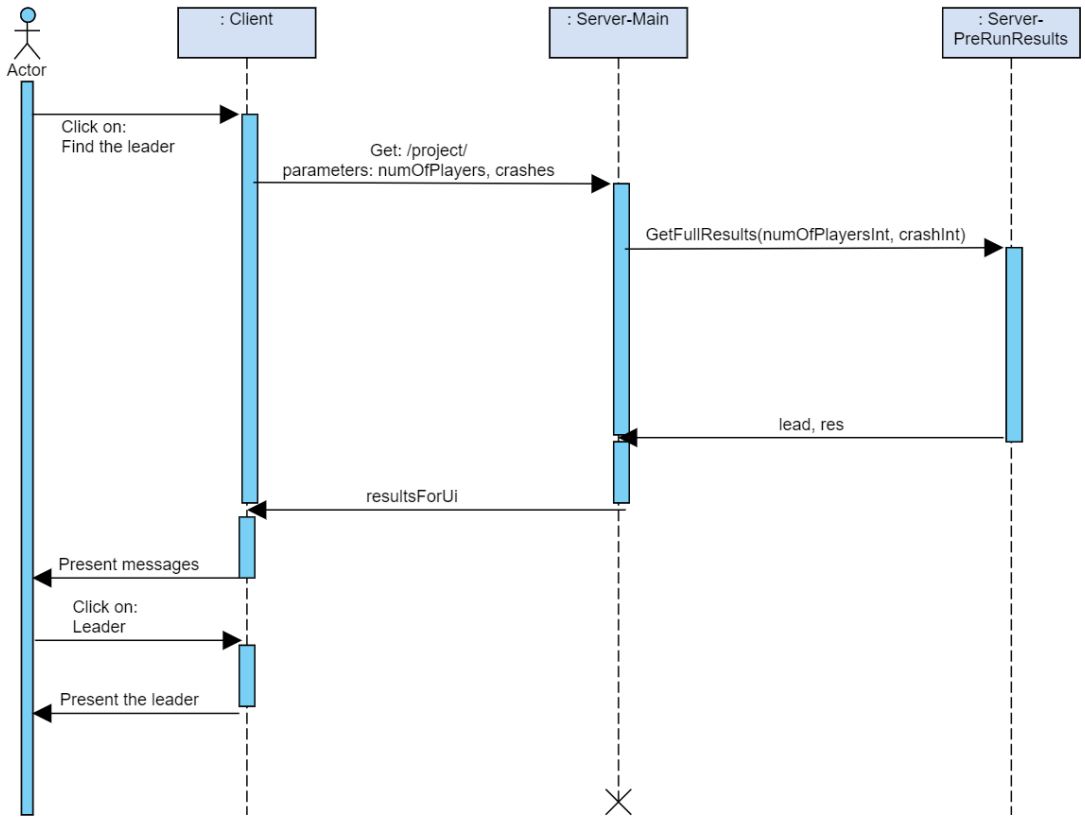
```
c.Writer.WriteHeader(http.StatusOK)
c.Writer.WriteString(resultsForUi)
```

הלקוח מקבל את התוצאות, שומר אותן בlocal storage ומעביר את המשתמש למסך הבא.

```
localStorage.setItem("algoResults", response);
window.location.pathname = "algorithmPage.html";
```

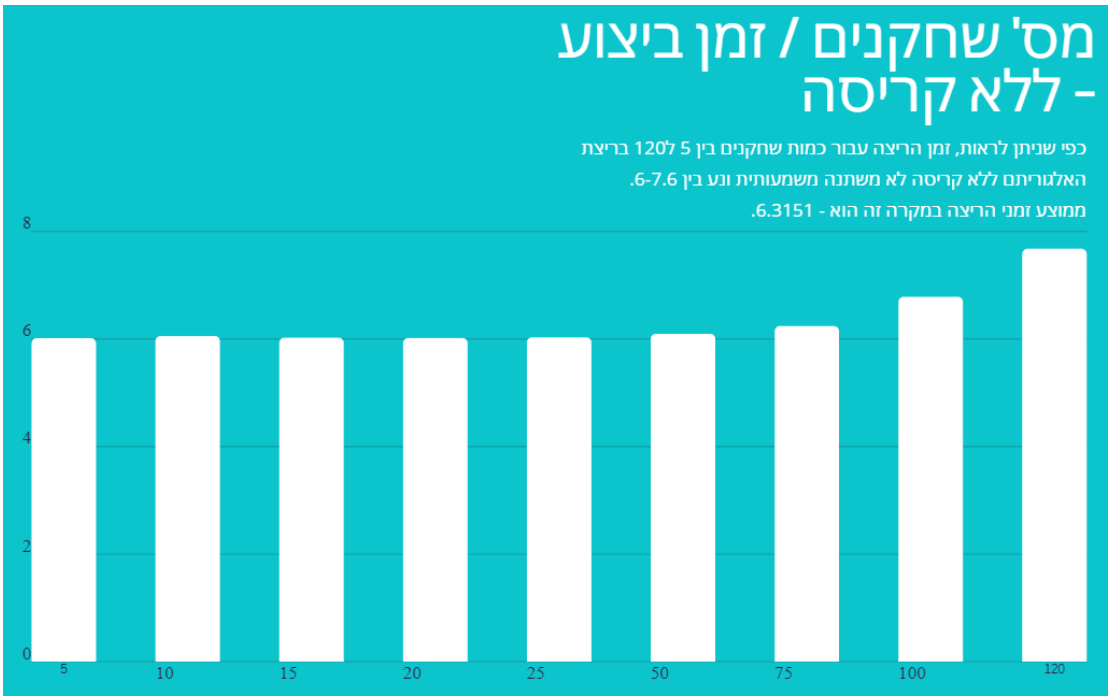
במסך השני, הלקוח מפרסר את התוצאות ומציג את הפלט על המסך. כאשר המשתמש לוחץ על הכפתור בתחתית, הלקוח שומר בlocal storage את התוצאה של הleader ומעביר את המשתמש למסך האחרון שבו מוצג המנהיג וכפתור לחזרה למסך הראשי.

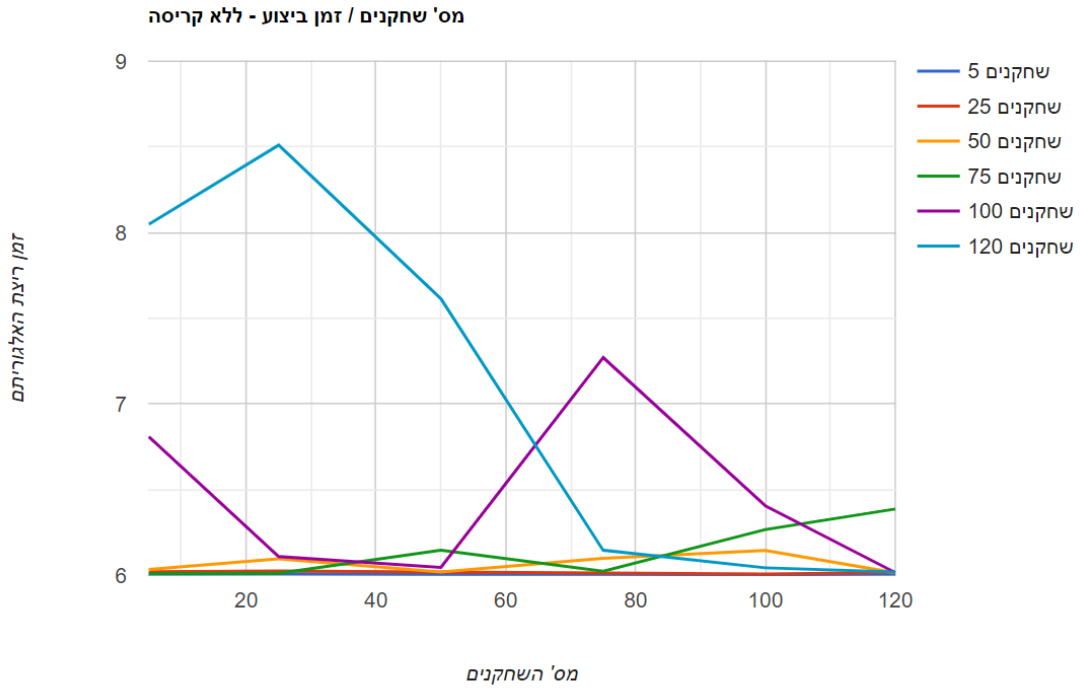
<u>דיאגרמת Sequence המתארת את הFlow הראשי:</u>

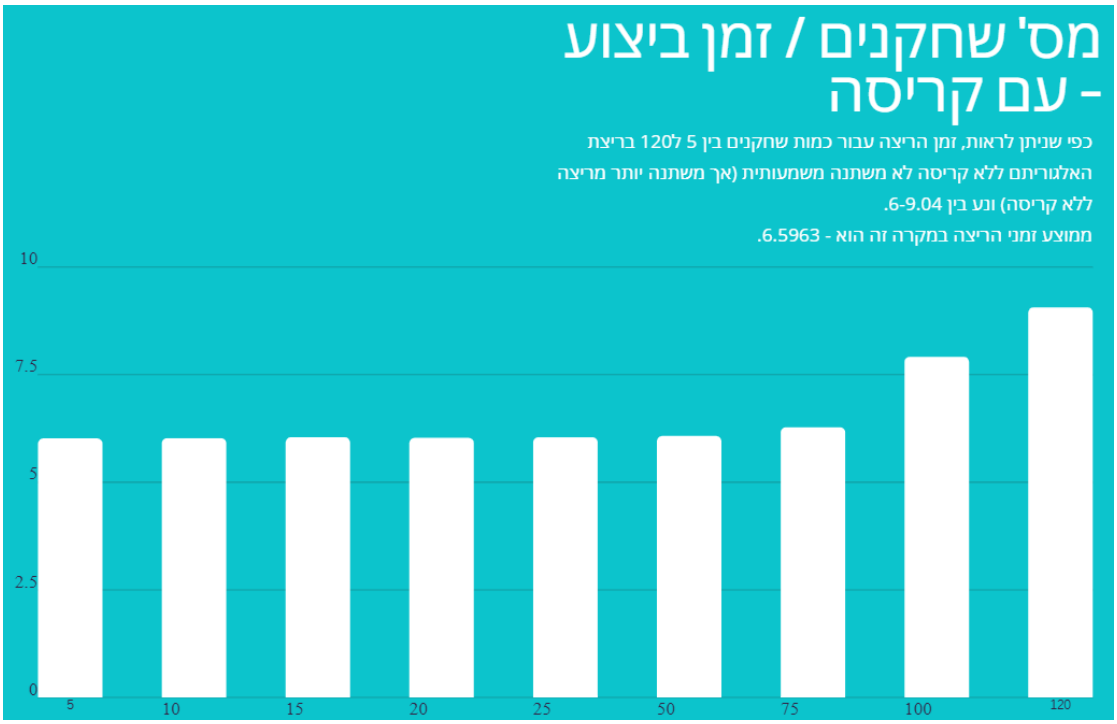


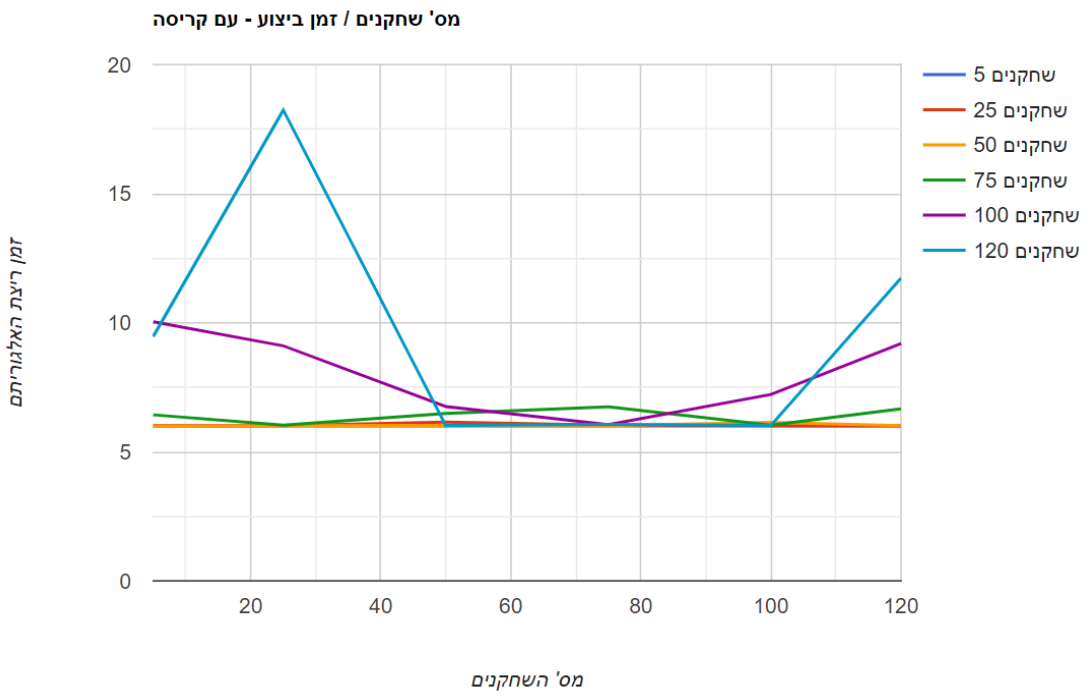
- <u>בדיקות זמני ריצה ומס' הודעות המוחלפות בין השחקנים בזמן הריצה:</u>
בקובץ excel בשם "בדיקות.xlsx" ניתן למצוא את כל החישובים שהובילו ליצירת הגרפים.

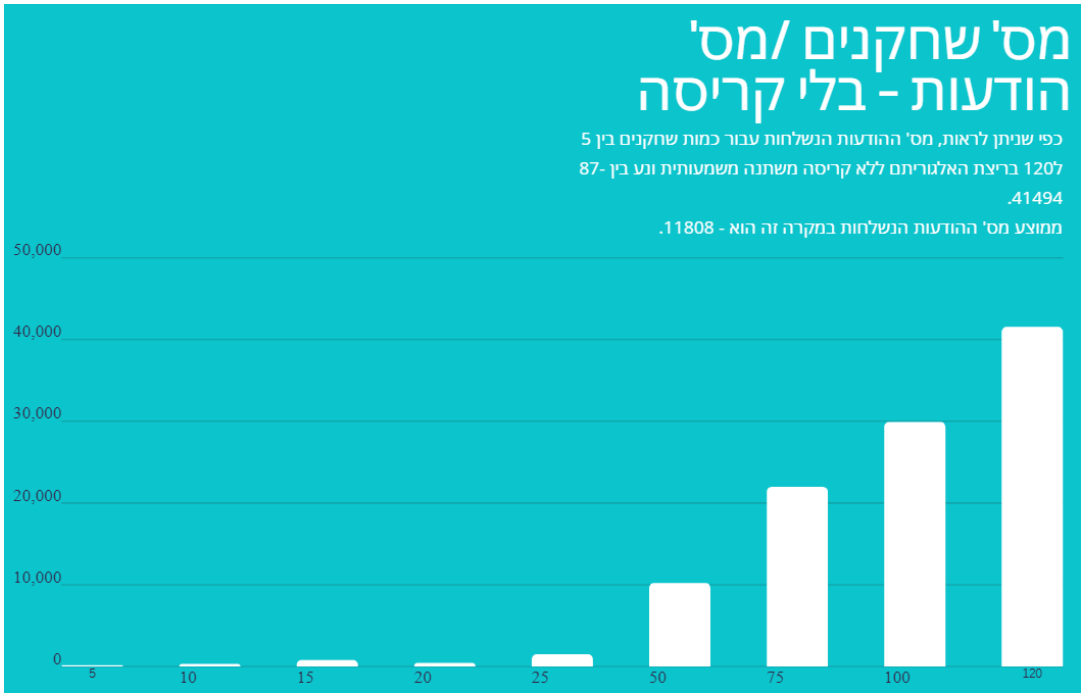
<u>גרפים המשקפים את תוצאות הבדיקות:</u>

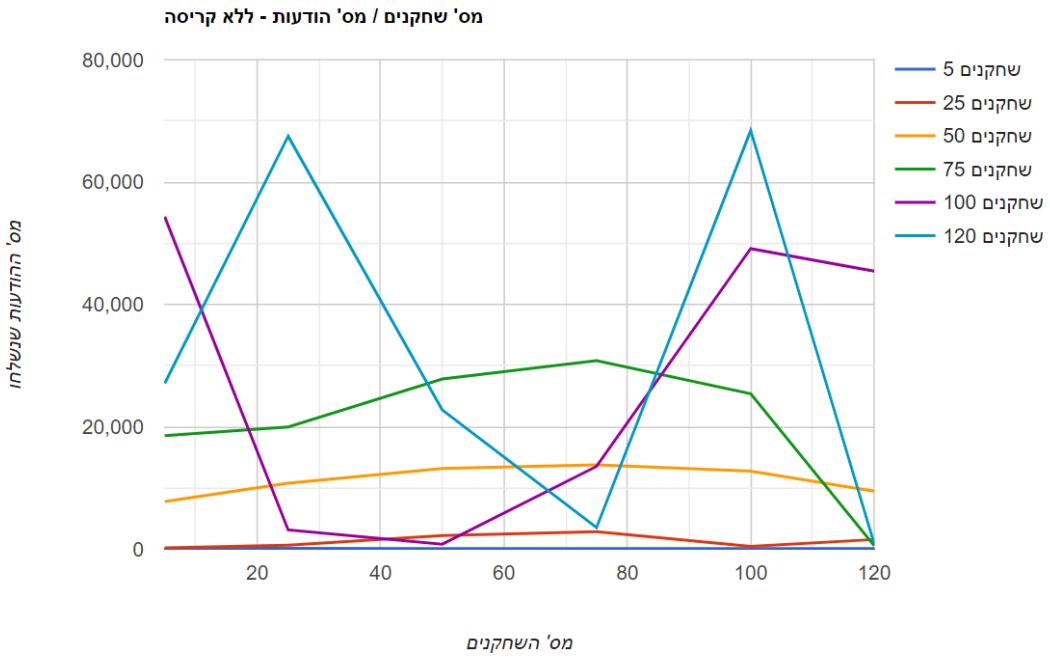


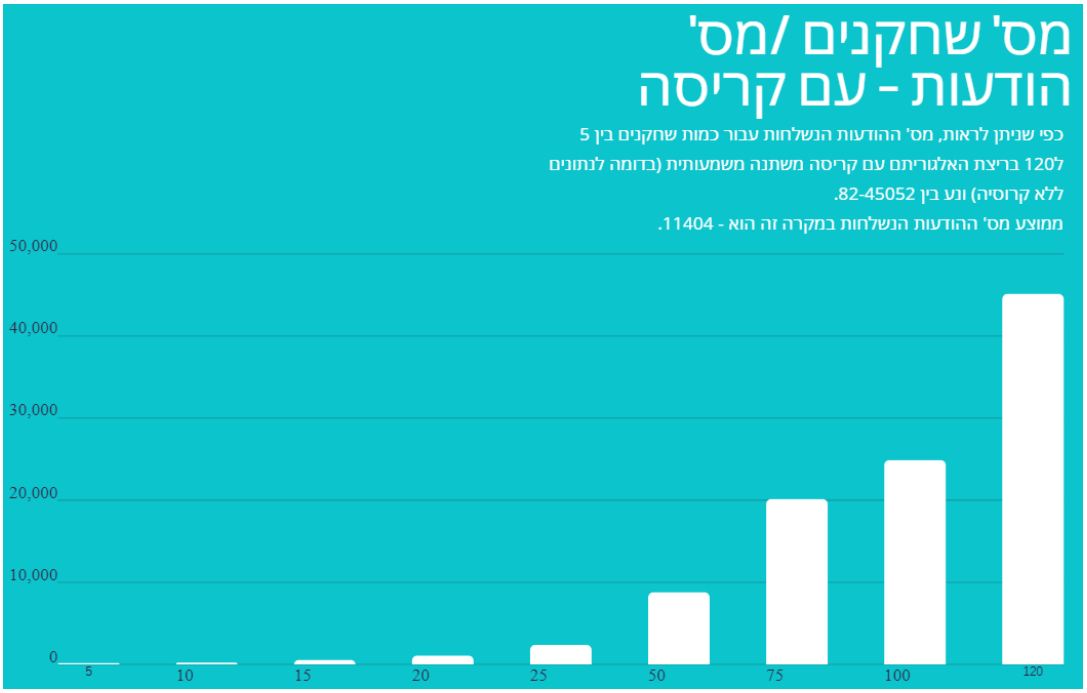


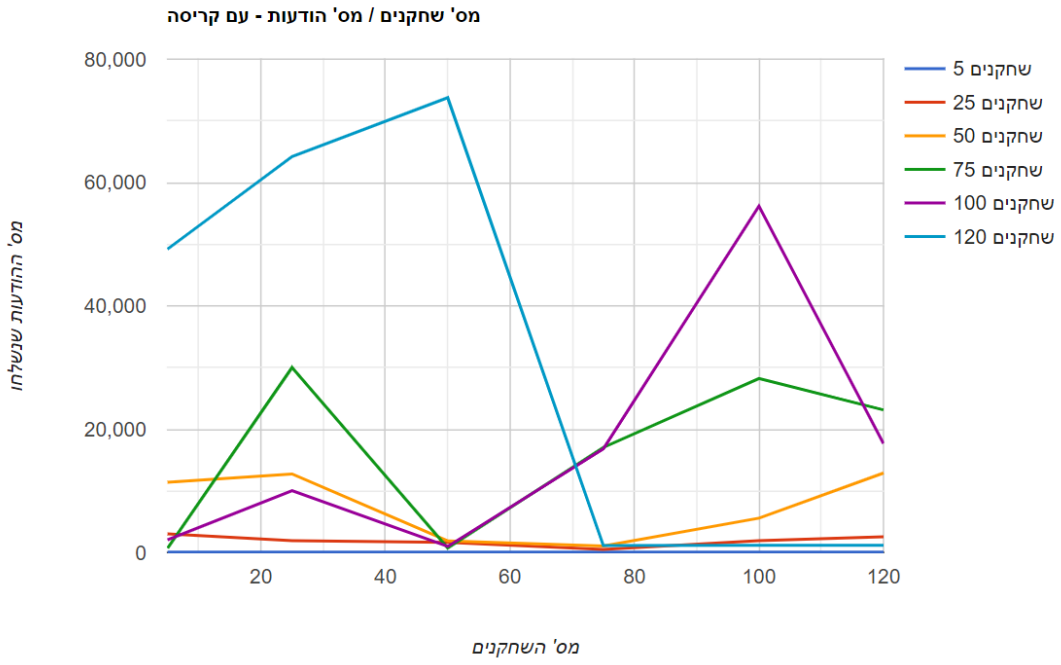

מס' שחקנים / זמן ביצוע
- עם קריסה
כפי שניתן לראות, זמן הריצה עבור כמות שחקנים בין 5 ל120 בריצת
האלגוריתם ללא קריסה לא משתנה משמעותית (אך משתנה יותר מריצה
ללא קריסה) ונע בין 6-9.04.
ממוצע זמני הריצה במקרה זה הוא - 6.5963.
10
7.5
5
2.5
0
5
10
15
20
25
50
75
100
120


מס' שחקנים / זמן ביצוע - עם קריסה
20
15
10
5
0
20
40
60
80
100
120
זמן ריצת האלגוריתם
מס' השחקנים
שחקנים 5
שחקנים 25
שחקנים 50
שחקנים 75
שחקנים 100
שחקנים 120


מס' שחקנים /מס' הודעות - בלי קריסה
כפי שניתן לראות, מס' ההודעות הנשלחות עבור כמות שחקנים בין 5 ל120 בריצת האלגוריתם ללא קריסה משתנה משמעותית וגע בין -87 41494.
ממוצע מס' ההודעות הנשלחות במקרה זה הוא - 11808.
50,000
40,000
30,000
20,000
10,000
0
5
10
15
20
25
50
75
100
120


מס' שחקנים / מס' הודעות - ללא קריסה
80,000
60,000
40,000
20,000
0
20
40
60
80
100
120
מס' ההודעות שנשלחו
מס' השחקנים
שחקנים 5
שחקנים 25
שחקנים 50
שחקנים 75
שחקנים 100
שחקנים 120


מס' שחקנים /מס' הודעות - עם קריסה
כפי שניתן לראות, מס' ההודעות הנשלחות עבור כמות שחקנים בין 5
ל120 בריצת האלגוריתם עם קריסה משתנה משמעותית (בדומה לנתונים
ללא קרוסיה) ונע בין 82-45052.
ממוצע מס' ההודעות הנשלחות במקרה זה הוא - 11404.
50,000
40,000
30,000
20,000
10,000
0
5
10
15
20
25
50
75
100
120


מס' שחקנים / מס' הודעות - עם קריסה
80,000
60,000
40,000
20,000
0
20
40
60
80
100
120
מס' ההודעות שנשלחו
מס' השחקנים
שחקנים 5
שחקנים 25
שחקנים 50
שחקנים 75
שחקנים 100
שחקנים 120

אתגרים ופתרונות ("חוויות קרב"):

- קושי: לצורך הרצה של האלגוריתם בשפת GO יש להוריד את הסביבה למחשב וכן, להתקין את הextension של השפה בסביבת העבודה. בהתחלה היה קשה להבין כי חובה לבצע את שתי הפעולות וחשבנו כי הורדה של הקובץ בלבד מספיקה. לאחר הורדת הextension, ניסינו להריץ את הקוד וגם אז הופיעה שגיאה בהרצה.
  התגברות: כיוון שמיקמנו את תיקיית הפרויקט במיקום שונה מתיקיית הsrc אליה מכוון הpath של הDEBUG של GO (וזאת לאור ההורדה שהתבצעה בשלבים), נראה היה כי הDEBUG לא מוצא את חבילות המחלקות השונות להן עשינו import בחלק ממחלקות הפרויקט ועל כן, היה צורך בשינוי של הקונפיגורציה של הסביבה או בשינוי מיקום הקובץ. כיוון שעבדנו עם git ביצענו דחיפה של כל הקוד לremote repository ולאחר מכן ביצענו clone למיקום אליו פונה הGOPATH. לאחר ביצוע פעולה זו הצלחנו להריץ את הקוד.

- קושי: העבודה עם visual studio code. כיוון שאיננו רגילות לכתוב בסביבת עבודה זו, היה עלינו ללמוד ולהבין כיצד עלינו לכתוב את הקוד ע"מ שהתוכנית תוכל לרוץ. בvisual studio code ישנם errors רבים, שאינם דומים לסביבות העבודה האחרות, כגון, שגיאה על אי כתיבת הערות לפני פונקציה, שגיאה על הצהרה של משתנה של השתמשנו בו ועוד.. לא תמיד ניתן היה להבין מה בדיוק השגיאה שמונעת את ריצת האלגוריתם ולפעמים היה מדובר על הכנסת ריווח בהערה.
  התגברות: קראנו הרבה באתרי אינטרנט שונים, נעזרנו רבות באתר stackoverflow ועם הזמן והכתיבה המרובה בסביבה, הבנו טוב יותר את צורת הכתיבה המתבקשת בסביבת עבודה זו, וכך הכתיבה של הקוד וההתקדמות, הייתה מהירה יותר והרגשנו שאנחנו מתעכבות פחות על דברים שוליים כנ"ל.

- קושי: בתחילת הדרך ידענו כי אנו רוצות לעבוד על חלקים שונים של הפרויקט, אבל כיוון שאנחנו גם עובדות במשרה מלאה ולוח הזמנים שלנו צפוף ידענו כי נאלץ לעבוד על הקוד בנפרד. בעבר, כשהיינו מבצעות יחד עבודות, היינו מעבירות את קבצי הקוד העדכניים במייל. בפרויקט זה, הבנו כי זו אינה דרך נכונה וכי קשה לזקק בכל עדכון באיזה קבצים בדיוק בוצעו שינויים וכן, לא לדרוס שינויים מהותיים אחת של השנייה ע"י העתקה של קבצים שלמים.
  התגברות: החלטנו לעבוד עם git. יצרנו repository לפרויקט שלנו באתר github . בכל שלב של בניית האלגוריתם יצרנו branch חדש עם השינויים שלנו ולאחר שבדקנו שהקוד בטוח ולא שובר את הmaster שלנו, ביצענו מיזוג של branch אל הmaster. דאגנו לבצע commits מסודרים עם הערות ענייניות, על מנת שבמקרה הצורך נוכל לחזור לגרסאות קודמות שעבדו לנו באופן תקין ולא נצטרך לגבות את הקוד באלפי תיקיות כפי שעשינו בעבר. דרך זאת, בהחלט

אפשרה לנו לעבוד יחד, בצורה מרוחקת על אותו הקוד ולהגיע בסוף לתוצאה מסונכרנת בצורה הטובה ביותר.

- קושי: בבניית הframework של הטסטים בפרוטוקול התקשורת של שליחת ההודעות היו שלבים, כגון טסטים שבודקים שליחה בchannels, שהיה קשה לנו לבדוק ולהעריך מה התוצאה שלהם.
התגברות: פתרנו את הקושי ע"י מחקר מעמיק של איך בונים נכון טסטים בGO והבנה של איך הchannels של GO עובדים.

- קושי: האלגוריתם המתואר במאמר אותו מימשנו, הוא אלגוריתם המיועד למערכות מבוזרות, בעלות מספר מעבדים, בסביבה אמיתית. מערכות אלו רצות באופן מתמשך ומצטרפים אליהם מעבדים חדשים לאורך הזמן. ביצירת השרת וכתיבת האלגוריתם, היה עלינו להביא את הפרמטרים שעטפו את האלגוריתם ואת האובייקטים השונים לדימוי הכי קרוב לריצה אמיתית וזאת ע"מ שנוכל לראות בצורה אמיתית, מעשית ונכונה את ריצת האלגוריתם אותו חקרנו. בנוסף, כאשר רצינו לממש ריצה של האלגוריתם בתרחיש של קריסת מנהיג, היה עלינו למצוא דרך יצירתית והגיונית לדמות את הקריסה (כיוון שהיה עלינו ליצור אותה באופן יזום) בצורה שתהיה הכי קרובה לקריסה של שרת באופן אקראי כפי שקורה במציאות.
התגברות: הקריאה והמחקר המקדימים לכתיבת הקוד סייעו רבות בהבנה עמוקה יותר של האלגוריתם ועל כן ידענו לאיזה התנהגות אנחנו מצפים. הרצנו את האלגוריתם עם פרמטרים שונים ולוגיקות שונות לדימוי קריסת מנהיג בכל פעם וראינו מה ההתנהגות שהייתה הכי קרובה להתנהגות לה ציפינו. בסופו של דבר הצלחנו להגיע לביצועים טובים והדמייה שמשקפת באופן אמיתי את הריצה של האלגוריתם המתואר במאמר.

- קושי: האלגוריתם, רץ כמובן בסביבה של מערכת בה ישנם מעבדים/שרתים הפועלים באופן מקבילי, לכן, לצורך הרצה נכונה של האלגוריתם היה עלינו להריץ את האלגוריתם במקביל אצל כל אחד מהאובייקטים שדימה שרת או מעבד בתוכנית אותה כתבנו.
התגברות: השתמשנו בשפת GO, שהיא שפה שתומכת בהרצה מקבילית. אצל כל אחד מה"שחקנים" האלגוריתם רץ על goroutines, שאלו המקבילים של ת'רדים בשפת GO . בחרנו להשתמש דווקא בשפה זו, כיוון שהgoroutines הם זולים ומהירים יותר מת'רדים בשפה אחרת. בנוסף, היה עלינו להבין כיצד מבצעים בשפה זו הרצה הדומה לmulti threading, לצורך כך, חקרנו באינטרנט ובספרי הדרכה של השפה, עד שהגענו להבנה - אילו פקודות בשפה ומה המבנה אותם צריך להכניס בקוד ע"מ שהריצה תתבצע כפי שרצינו.

- קושי: כאשר ניסינו להריץ את האלגוריתם במקביל עבור כל השחקנים בת'רדים נפרדים עם הפונק' go func() של threads בשפת GO, נתקלנו

בdeadlock. מכיוון שהאלגוריתם עדיין לא עבד בצורה חלקה, נותר מצב שבו חלק מהשחקנים בתוכנית חיכו להודעות משחקנים אחרים שלא הגיעו אליהם ולכן התוכנית נתקעה ושלחה הודעה Panic של deadlock.
התגברות: הרצנו את התוכנית במצב DEBUG והתאמנו המימוש למצב בו השחקנים לא נתקעים ומחכים להודעות אחד מהשני.

- קושי: לאורך כתיבת הקוד, נתקלנו בבאגים שונים אותם היינו צריכות לפתור. לאחר שהעברנו את הקוד לריצה מקבילית היה קשה מאוד לדבג את הקוד בצורה רגילה ובנוסף, לא תמיד הבאגים אותם רצינו לפתור חזרו על עצמם באותה צורה או בכלל, בשל הקושי הזה היה לנו קשר לאתר את מקור הבעיה ואת קטע הקוד שגורם לקריסת התוכנית.
התגברות: השתמשנו בהדפסות רבות לאורך כל הקוד, כאשר בכל הדפסה ציינו בנוסף למשמעות של אותה הדפסה, מי ה"שחקן" שרץ ובאיזה סיבוב הוא נמצא. לאחר ביצוע ההדפסות השונות, היכולת לזהות את מקור הבעיה הייתה מהירה ומדויקת וכך יכולנו לתקן בצורה טובה ומקצועית את הבאגים השונים בקוד.

- קושי: כאשר יצרנו את התקשורת הראשונית בין השרת ללקוח, נתקלנו בשגיאה הבאה:
"...has been blocked by CORS policy: Response to preflight request doesn't pass access control check: It does not have HTTP ok status"
לאחר מחקר, הבנו ששגיאה זו מתרחשת כאשר מנסים לייצר בקשות בין דומיינים שונים. הדפדפן לא יאפשר את הבקשות האלה מסיבות אבטחה. למשל על מנת למנוע מתקפות Phishing.
התגברות: השתמשנו ב- Cross-Origin Resource Sharing או בקצרה - CORS. זוהי דרך בטוחה לאפשר בקשות בין שני דומיינים שונים. בצד השרת, השתמשנו ב- open-source הבא: github.com/gin-contrib/cors, שיודע להוסיף את הheaders הנדרשים כדי לשלוח את הבקשות.

- קושי: לקראת מועד הגשת הפרויקט, הבנו כי אחד מהחלקים החשובים של הפרויקט, החלק המחקרי, לא בוצע. חברת הצוות, שהייתה אמונה על ביצועו פרשה מהפרויקט באופן מפתיע ולא צפוי ונותרנו עם חוסר גדול בפרוייקט.
התגברות: כאשר הבנו שהחלק לא בוצע ושלא ניתן לתקשר עם הסטודנטית שהייתה אחראית על ביצועו, הבנו שעלינו להשלים את הפער הקיים במהירות האפשרית. לצורך כך, חילקנו בינינו את העבודה שאותה הסטודנטית הייתה אמורה לבצע, כל אחת לקחה חלק בתחום שהיא יותר חזקה בו וידעה כי תוכל לעמוד במשימות במהירות ויעילות, ישבנו על ההשלמה ימים ולילות - שעות רבות ובסופו של דבר אנחנו מרגישות שהצלחנו להדביק את הפער, עד כמה שהיה ניתן.

הוראות התקנה:

הוראות התקנה ושימוש ניתן לראות בgithub של הפרויקט:
https://github.com/efi411/final_project3/blob/master/README.md

כיוונים עתידיים:

1. ניתן להמשיך ולממש על גבי האלגוריתם שלנו את האלגוריתם השלישי במאמר ולשפר עוד את אלגוריתם לבחירת המנהיג.

2. ניתן להפוך את החוויה לדינאמית יותר עבור המשתמש ע"י הכנסת נתוני הריצה בכל פעם לקובץ json שממנו נשלפות התוצאות וכך, בכל פעם, נוכל להציג למשתמש יותר ויותר אפשרויות לכמות של שחקנים ע"פ כמות התוצאות שקיימות בjson.

3. ניתן להוסיף טסטים לבדיקת הקוד עבור כל המחלקות בקוד מלבד הflow הבסיסי שנבדק.