

פרויקט סייבר במסגרת תכנית גבהים

RAID5



רועי זהר

209896174

מרכז חינוך ליאו באק

הגנת סייבר

מנחים

שרית לולב

אלון בר-לב

תוכן עניינים

1	תוכן עניינים
3	מבוא
4	ארכיטקטורה
4	דיאגרמה
5	המשתתפים במערכת
5	שירותים שה Frontend Server מציע
6	הרקע התיאורטי
6	RAID5
6	Multicast
7	מושגים בתהליך הגיבוי והשחזור אשר מתבצע
7	טכנולוגית הגיבוי במערכת
7	מבנה הדיסקים
8	קריאה מדיסק
9	כתיבה לדיסק
10	מושגי תקשורת
11	מושגי מערכות הפעלה
11	שפות התכנות אשר נעשה בהם שימוש
12	מושגים נוספים
13	מימוש
13	דיאגרמת בלוקים
13	דיאגרמת בלוקים עבור Frontend Server
14	דיאגרמת בלוקים עבור Block Device Server
15	Sequence Diagrams
18	מבני נתונים
20	פרוטוקולי תקשורת
20	UDP/Multicast
20	HTTP Services
23	הזדהות
25	מכונות מצבים
28	אתגרים במימוש ודרך הפתרון

28	לולאה מרכזית
29	פונקציות עזר מרכזיות
31	בעיות ידועות
32	התקנה ותפעול
32	קבצי Configuration
32	config.ini - Frontend Server
33	config0.ini - Block Device Server
34	התקנה
34	פירוט לגבי התפעול
38	Logging
40	תוכניות עתידיות
41	פרק אישי
42	תיעוד קוד + קוד הפרויקט
43	נספחים
43	נספח א' - Sequence Diagram Source

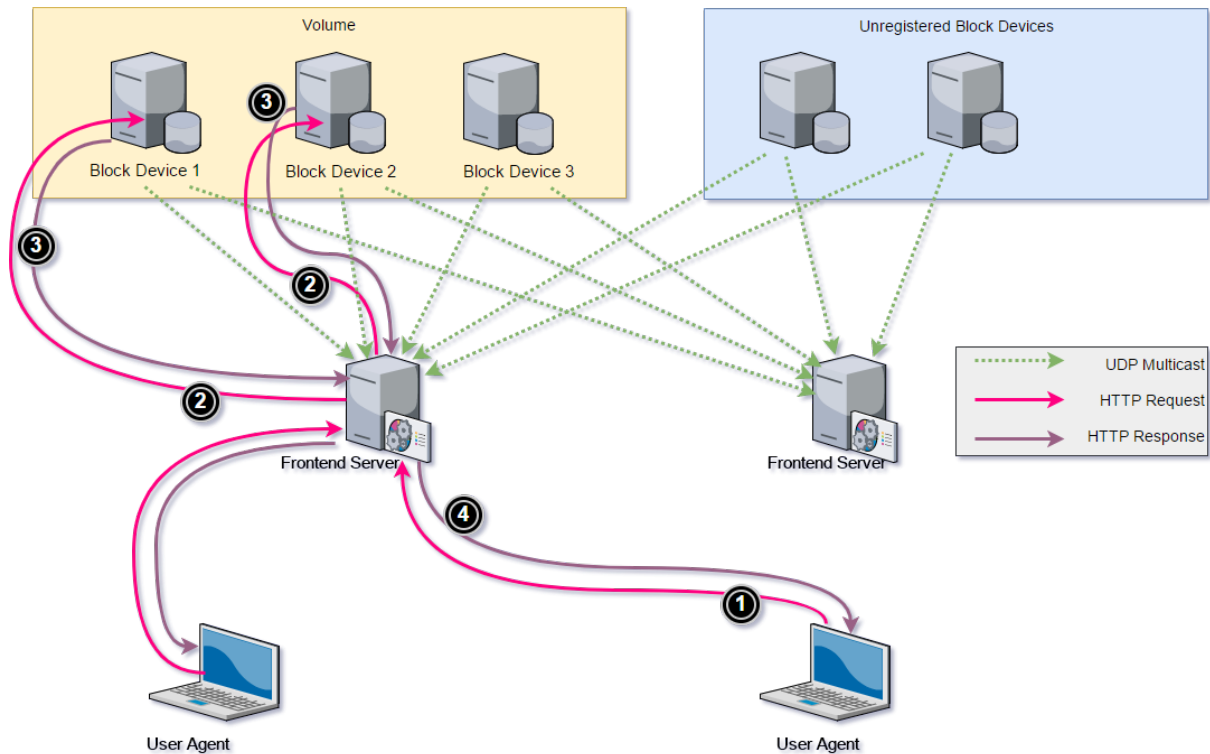
מבוא

לכולנו יש מידע אישי אשר היינו מעוניינים שיהיה שמור ומוגן באופן קל ונוח. בעולם האמיתי לצערנו, תקלות הן דבר אופייני אשר יש להתגבר עליו. דוגמא לתקלה קשה בעולם שמירת גיבוי הנתונים, היא **נפילת שרת**, אשר איתו נאבד כל המידע אשר היה עליו. בין אם מדובר בפגיעה פיזית בשרת, בעיית חומרה/תוכנה, או כיבוי יזום, מערכות אשר מבטיחות שמירת מידע צריכות להיות מסוגלות להתגבר על בעיות מסוג זה.

הפרויקט בראש ובראשונה בא להציע למשתמש מערכת לשמירה וגיבוי של קבצים ומידע. לפני הכל, המערכת מאפשרת למשתמש לעלות ולהוריד "בלוקים" של מידע מן השרתים שברשותו, לנהל את השרתים, להציג את מצבם, ועוד. בתצורה הזו, המשתמש לא חייב לשמור את הנתונים שלו על גבי המחשב האישי שלו אלא על שרתים המיועדים לכך. ויוכל לגשת אליהם מכל מקום. הבעיה העיקרית אשר המערכת יודעת לטפל בה היא נפילת דיסק, או שרת. במקרה כזה, המערכת יודעת לדווח על התקלה למשתמש, אך עדיין להמשיך לאפשר לו לגשת לנתונים האישיים שלו כפי שעשה קודם לכן. באופן מפתיע, המערכת אפילו יודעת לתת למשתמש לגשת לאותו שרת אשר נפל ולראות את הנתונים אשר היו בו, ואפילו לכתוב אליו! (כיצד היא עושה זאת יפורט בהמשך). בנוסף לכך, המערכת מאפשרת בנייה מחדש של השרת בעת חזרתו לפעולה, גם אם הנתונים עליו נמחקו. המערכת גם מאפשרת הזדהות ברמה הבסיסית ביותר, בין המשתמש לשרתים ובין השרתים לבין עצמם, כך שיש למערכת מימד מסויים של אבטחה.

ארכיטקטורה

דיאגרמה



Stage	Description
1	The User Agent sends a HTTP request to a Frontend Server, requesting one of the many services it offers.
2	The Frontend Server processes the HTTP request, and creates multiple HTTP requests to the Block Device Servers in order to fulfill the request.
3	Each Block Device Server receives and processes the Frontend's HTTP request, and responds with the relevant information.
4	The Frontend Server receives these responses, and combines them to a new HTTP response to the User Agent.

לעיל מתוארת עיבוד בקשה במערכת, כגון כתיבה לאחד מהדיסקים.

המשתתפים במערכת

- **Frontend Server** - השרת המרכזי, אשר מנהל את התקשורת עם הלקוחות השונים, ופונה ל-Block Device Servers. זוהי הנקודה הראשונה, אליה מגיעה כל בקשה של המשתמש בפרוטוקול HTTP. ה-Frontend בודק את סוג הבקשה ואת הרשאות המשתמש, ומוציא אותה לפועל.
- **Block Device Server** - שרת HTTP המאפשר קריאה וכתיבה של "בלוקים" על גבי קבצים דמויי דיסקים. הדיסקים מחולקים ל"בלוקים" בגודל 4K. שרתים אלו לעיתים ייקראו דיסקים, משום שהם מדמים דיסקים מרוחקים אשר ניתן לגשת אליהם ולכתוב/לקרוא מהם מידע. אילו שרתים פשוטים יחסית, ללא לוגיקת גיבוי.

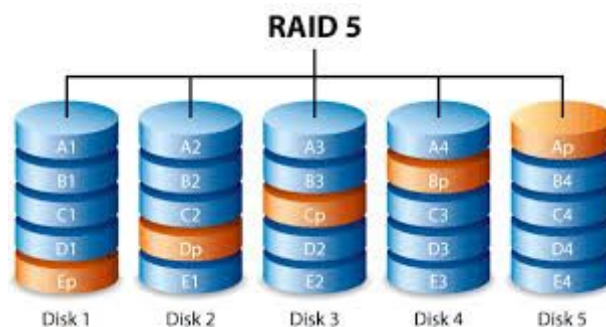
שירותים שה-Frontend Server מציע

- כתיבה של בלוקים אל דיסק.
- קריאה של בלוקים מדיסק.
- יצירת volumes מדיסקים פנויים.
- "ניתוק" דיסק. הדמיה של נפילת שרת.
- חיבור דיסק ל-volume קיים.
- הצגת מצב המערכת וכל ה-volumen הפעילים.

הרקע התיאורטי

RAID5

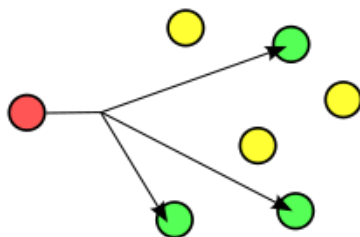
בעברית מערך יתיר של דיסקים עצמאיים, היא שיטה שבה מאחדים מספר דיסקים קשיחים ליחידה לוגית אחת. איחוד זה מאפשר גישה במקביל למספר דיסקים, המשפרת את מהירות העבודה, ושמירת מידע יתיר, המאפשר לשחזר מידע שאבד בתקלה. ישנן שיטות שונות לניהול מערך הדיסקים. שיטות אלו מכונות **רמות RAID** (באנגלית RAID levels) וכל אחת מהן מספקת יתרונות שונים.



רמת RAID5 מתארת מערך של שני דיסקים לפחות, שבהם המידע והזוגיות מפוזרים בין כל הדיסקים, והזוגיות נכתבת ברמת הבלוק כפי שמוצאצ בשרטוט לעיל. מכיוון שהזוגיות מפוזרת בין כל הדיסקים, הכתיבה אינה תלויה בדיסק אחד וניתן לבצע מספר פעולות כתיבה בו-זמנית.

Multicast

Multicast היא פעולת ניתוב של נתונים מאלמנט מקור אל קבוצת אלמנטים של תקשורת באופן מקבילי. בפרויקט יש שימוש ב-Multicast כאשר כל Block Device Server מכריז על עצמו בפני ה-Frontend Servers. השימוש בפרוטוקול זה נעשה למען גילוי ה-Block Devices בידי ה-Frontend Servers. למעשה כל שרת בלוקים "מכריז" על עצמו על גבי כתובת מסוימת, אשר אליה מאזינים ה-Frontend Servers:



ממול מתוארת סקיצה של Multicast, בו נקודה אדומה מנסה לתקשר עם נקודות ירוקות ספציפיות. הנקודות הירוקות הן שרתי ה-Frontend, ואילו הנקודה האדומה מסמלת שרת ה-Block Device אשר מכריז על עצמו. ברגע ש-Frontend מזהה Block Device, הוא שומר אותו במבנה הנתונים שלו ומעדכן את הזמן בו הוא התחבר אליו.

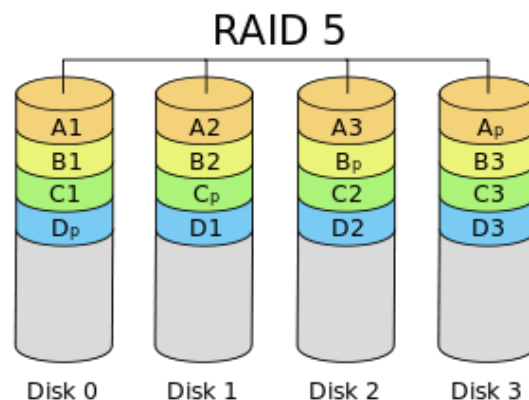
מושגים בתהליך הגיבוי והשחזור אשר מתבצע

- **RAID Levels** - אוסף של קונפיגורציות על דיסקים קשיחים כדי לייצר מאגרי נתונים אשר ניתן לשחזרם בעת פגיעה בדיסק אחד. השלבים הללו משתמשים בטכניקות שונות על מנת להגיע לתוצאות אמינות ויעילות. אנו נעסוק בטכניקת RAID5, אשר מפורטת יותר בהמשך.
- **parity** - זוגיות בעברית, טכניקה אחת לגבות ולשחזר קבצים בעת תקלה. משתמשת בפעולת XOR כדי לקבל מידע בנוגע לביטים בשאר הדיסקים.
- **Logic Disk / Physical Disk** - הדיסקים הלוגיים הם רפרזנטציה לדיסקים אותם המשתמש חושב שהוא עורך. מבחינתו, הדיסקים הללו רציפים. הדיסקים הפיזיים הם כיצד המידע של המשתמש נשמר בפועל, וזה לאו דווקא באופן רציף כפי שמוצג לו. צריכה להיות המרה ברורה בין המיקום בדיסק הלוגי לבין המיקום בדיסק הפיזי.
- **Volume** - אוסף דיסקים אשר מגבים אחד את השני. מהווים יחידה לוגית בעלי מאפיינים משותפים, כגון סיסמא משותפת.

טכנולוגית הגיבוי במערכת

מבנה הדיסקים

נתבונן במקרה בו יש ברשותנו 4 דיסקים, או Block Devices.



RAID5 מקצה עבור כל סדרת בלוקים, דיסק אחר אשר ישמור את הזוגיות של שלושת הבלוקים אחרים. לדוגמא, עבור סדרת הבלוקים הראשונה, הקצה RAID5 את דיסק 4 לזוגיות ($A_4 = A_p$), ומתקיים:

$$(A_1) XOR (A_2) XOR (A_3) = A_p$$

פעולת XOR, מחזירה לנו את הparity (זוגיות) של הבלוקים, ואנו שומרים את התוצאה הזו ב

A_p . בכל סדרת בלוקים, אנו מקצים דיסק אחר לזוגיות. הסתברותית, מספר הכתיבות לכל דיסק לצורכי

גיבוי יהיה $\frac{1}{4}$, ובכך הורדנו את מספר הכתיבות הממוצע לדיסק והורדנו עומס מן המערכת, בניגוד למצב שבו דיסק אחד היה צריך לעדכן את הזוגיות כל הזמן. במקרה כזה, $\frac{3}{4}$ ממקום האחסון על הדיסקים הוא לשימוש הלקוח, ו $\frac{1}{4}$ הוא לצורכי גיבוי.

באופן כללי, יכולות האחסון של המערכת בבתיים הן כדלקמן:

$$Physical\ Capacity = |Disks| \times \min(\{|disk| \mid \forall disk \in (Disks)\})$$

כאשר Disks מייצג את ה-Block Device Servers הפועלים, וגודל כל אחד מתאר את גודל הדיסק (קובץ) המקסימלי.

$$Usable\ Capacity = Physical\ Capacity \times \frac{|Disks| - 1}{|Disks|}$$

$$Usable\ Capacity\ Percentage = 100 \times \frac{|Disks| - 1}{|Disks|}$$

נשים לב שככל שמספר הדיסקים ב-volume גדול יותר, כך האחוז של מקום אחסון עבור המתמשש גדל.

קריאה מדיסק

במקרה שבו אנו מעוניינים לקרוא מדיסק, נפעל באופן הבא:

- (1) ננסה להתחבר אל הדיסק ישירות ולקרוא ממנו.
- (2) במקרה שההתחברות נכשלה מסיבה כלשהי, נעבור לשחזור התוכן של הדיסק משאר הדיסקים

(a) נשים לב לכמה עובדות חשובות אשר יסייעו לנו לשחזר את התוכן שעל הדיסק האבוד:

$$(A1) \ XOR \ (A2) \ XOR \ (A3) = Ap \quad (i)$$

$$(Ai) \ XOR \ (Ai) = 0 \quad (ii)$$

$$XOR \text{ פעולה קומוטטיבית} \quad (iii)$$

(b) נניח שאנו מעוניינים בתוכן A1, אך הדיסק עליו הוא שמור נפל (דיסק 0).

$$(A1) \ XOR \ (A2) \ XOR \ (A3) = Ap \ / \ XOR(Ap)$$

$$(Ap) \ XOR \ (A1) \ XOR \ (A2) \ XOR \ (A3) = 0 \ / \ XOR(A1)$$

$$(Ap) \ XOR \ (A2) \ XOR \ (A3) = A1$$

ממבנה המערכת, נסיק שניתן להשיג את תוכן A1 בעזרת XOR של כל שאר הדיסקים בvolume. אשר מאפשר לנו לגשת לכל התוכן שהיה על הדיסק למרות העובדה שהשרת נפל.

כתיבה לדיסק

כתיבה אל הדיסק הינה משימה יותר קשה מאשר קריאה ממנו, אך מתבצעת באופן דומה. בעת כתיבת בלוק, נרצה להתחבר ולכתוב אל שני דיסקים:

- (1) הדיסק אשר המשתמש ביקש לכתוב אליו
- (2) parity disk, כדי לעדכן את הזוגיות לשם שחזור עתידי.

נניח שוב, לשם הפשטות, שאנו מעונינים לשנות את בלוק A1. נגדיר מספר סימונים לשם פשטות ההסבר:

- A_p - תוכן בלוק הזוגיות לפני העדכון
- $\langle A_p \rangle$ - תוכן בלוק הזוגיות אחרי העדכון
- A_1, A_2, A_3 - תוכן בלוקי המידע לפני העדכון
- $\langle A_1 \rangle$ - תוכן הבלוק אשר אנו רוצים לעדכן, תוכן הבלוק אחרי העדכון

נפצל שוב לשני מקרים:

(1) שני הדיסקים פעילים: עלינו לבצע שתי כתיבות:

- (a) כתיבה אל הדיסק אשר התבקשנו לכתוב אליו: זה מתבצע ללא מאמץ ואנו כותבים אליו ישירות, כלומר מחליפים את A_1 ב- $\langle A_1 \rangle$.
- (b) כתיבה אל parity disk על מנת לעדכן את הזוגיות של המערכת. למעשה, עלינו לכתוב אל parity disk את הביטוי הבא: $\langle A_1 \rangle XOR (A_2) XOR (A_3)$, משום שזהו מצב הזוגיות במערכת לאחר עדכון הבלוק הנדרש. נזכיר ש- A_2 ו- A_3 אינם ידועים ל-Frontend Server, ושעלינו לגשות אל ה-Block Devices כדי לקבל אותם.
- (c) ביטוי זה אינו יעיל במיוחד, משום שעלינו לגשת לכמעט כל הדיסקים שברשותנו. ולכן במקרה שבו אנו מנהלים דיסקים רבים, נקבל פגיעה משמעותית ביעילות. אציג כעת פיתוח מתמטי אשר מוריד את מספר הקריאות מדיסקים:

$$\langle A_1 \rangle XOR (A_2) XOR (A_3) = \langle A_p \rangle / XOR(A_1)$$

$$\langle A_1 \rangle XOR (A_1) XOR (A_2) XOR (A_3) = \langle A_p \rangle XOR (A_1)$$

$$\langle A_1 \rangle XOR (A_p) = \langle A_p \rangle XOR (A_1) / XOR(A_1)$$

$$(< Ap >) = (Ap) XOR (A1) XOR (< A1 >)$$

- (d) ובכך הפחתנו את מספר הקריאות מהדיסקים מ $|Disks| - 1$ ל-2 בלבד! קל לראות את היתרונות של הפיתוח הזה ביחס לשיטה הקודמת, במיוחד במקרה שבו מדובר במספר רב של דיסקים. נכתוב את הביטוי לעיל לתוך בלוק הזוגיות וסיימנו.
- (2) אחד משני הדיסקים הללו לא פעיל: במקרה שההתחברות לאחד משני הדיסקים נכשלה, נצפה לבעיות מהאלגוריתם לעיל. נפצל שוב לשני מקרים:
- (a) הדיסק הרצוי לא פעיל: נכתוב לתוך דיסק הזוגיות את הביטוי המקורי שלנו:
- $$(< A1 >) XOR (A2) XOR (A3)$$
- , משום שכעת אין לנו את A1 ביעילות. נשמור את $<A1>$ ב-cache עד שהדיסק הרצוי יחזור לפעול.
- (b) דיסק הזוגיות לא פעיל: נכתוב לתוך הדיסק הרצוי את $<A1>$, נחשב את הביטוי המוכר $(< A1 >) XOR (A2) XOR (A3)$, והפעם נשמור אותו ב-cache עד שדיסק הזוגיות יחזור לפעול.

מושגי תקשורת

- **HTTP** - פרוטוקול תקשורת שנועד להעברת דפי HTML ואובייקטים שהם מכילים ברשת האינטרנט. הפרוטוקול פועל בשכבת היישום של מודל ה-OSI ובשכבת היישום של מודל TCP/IP. בקשת HTTP מורכבת ממספר חלקים, בהם שיטת הבקשה, שדות כותרת וגוף הבקשה. נתייחס לשני מושגים בהקשר זה, והם **HTTP request**, בקשת HTTP מצד הלוקח, ו**HTTP response**, שהיא תשובת השרת אל הלוקח. שני סוגי בקשות HTTP הנתמכות בפרויקט:
- **GET** - מיועדת לקבלת אובייקט שנמצא על השרת, בכתובת שניתנת בתחילת ההודעה. בקשות GET הן הנפוצות ביותר ברשת האינטרנט.
- **POST** - בקשה דומה ל-GET, אשר הארגומנטים מועברים בדרך שונה. אני משתמש בסוג מסויים של פרוטוקול ב-POST, והוא **multipart/form-data**.
- **UDP - User Datagram Protocol**, הוא פרוטוקול השייך לשכבת התעבורה של מודל ה-OSI ולשכבת התעבורה של מודל ה-TCP/IP, המאפשר העברת נתונים לא אמינה אך פשוטה. העברת הנתונים מתבצעת ללא חיבור מקושר (Connectionless).
- **TCP - Transmission Control Protocol** הוא פרוטוקול בתקשורת נתונים הפועל בשכבות התעבורה של מודל ה-OSI ובמודל ה-TCP/IP, ומבטיח העברה אמינה של נתונים בין שתי תחנות ברשת מחשבים באמצעות יצירת חיבור מקושר (Connection Oriented).

- [IPv4 Address](#) - כתובת אינטרנטית בפרוטוקול האינטרנט הנפוץ ביותר IPv4. כתובת IPv4 מורכבת מ-32 סיביות ומיוצגת באמצעות 4 מספרים עשרוניים המופרדים בנקודה. כל מספר מהווה מקבץ של 8 סיביות וגודלו נע בין 0 ל-255.
- [Port](#) - הוא תהליך ספציפי שדרכו יכולות תוכנות להעביר נתונים באופן ישיר. השימוש הנפוץ ביותר בפורט הוא בתקשורת מחשבים במסגרת הפרוטוקולים הנפוצים בשכבת התעבורה: TCP ו-UDP. פורט מזוהה לכל כתובת או פרוטוקול מסוים על ידי מספר באורך 16 ביטים היוצר 65536 כתובות אפשריות ל-UDP ו-65535 כתובות אפשריות ל-TCP. כתובת זו נקראת "מספר הפורט".

מושגי מערכות הפעלה

- [Asynchronous IO](#) - גישה לטיפול ב-IO (קלט ופלט) אשר אינה תוקעת את התכנית עד לסיום כל ה-IO, אלא מאפשרת לתכנית להמשיך הלאה במצב של תקיעה. גישה זו מתבצעת על ידי [polling](#), בו אובייקט מסוג [poller](#) "מעיר" רק את האובייקטים אשר קיבלו IO.
- [POSIX](#) – אוסף תקנים בסיסיים שנאגד במטרה לשמור על תאימות בין מערכות הפעלה, בעיקר בין מערכות מבוססות UNIX.
- [daemon](#) – תכנית מחשב שרצה כתהליך ברקע ואינה נמצאת בשליטה של אינטרקציה עם משתמש. בפרויקט זה קיימת תמיכה של הרצת תכניות השרתים בתור תכניות daemon.
- [file descriptor](#) - מספר המייצג קובץ פתוח במערכת ההפעלה. כאשר אנו נפתח קובץ, נקבל file descriptor אשר בעזרתו נוכל לקרוא, לכתוב ולסגור את הקובץ.

שפות התכנות אשר נעשה בהם שימוש

- [Python](#) - שפת תכנות נפוצה ששמה דגש על קריאות קוד והרכבת תכניות ומבני נתונים מסובכים בדרך קצרה ופשוטה. בשפה זו נכתב מרבית הפרויקט.
- [HTML](#) - שפה מבוססת תגיות המאפשרת עיצוב של דפי אינטרנט שתצוגתם נתמכת בדפדפן. הדפים אשר מוצגים למשתמש נכתבו בבסיסים בשפת HTML
- [CSS](#) - שפת תכנות לעיצוב דפי אינטרנט. הגליונות קובעים את עיצובם של תגים ב-HTML, XHTML וכל שפה דומה ל-XML לבניית אתרי אינטרנט. נעשה שימוש בשפה על מנת לעצב את ממשק המשתמש.
- [Javascript](#) - היא שפת תכנות דינמית מונחית-עצמים המותאמת לשילוב באתרי אינטרנט ורצה על ידי דפדפן האינטרנט בצד הלקוח.

מושגים נוספים

- **קובץ log** – קובץ שמטרתו לתעד את האירועים ואת השגיאות שמתרחשים במהלך ריצה של תכנית. פרויקט זה תומך בהגדרת מסמך log, אליו נכתבים פרטים של חיבורי רשת שנפתחים ונסגרים במהלך הריצה, כמו גם תיעוד של שגיאות אפשריות.
- **Context-Based Programming** – תכנות שמבוסס על הרכבת מבנה נתונים מרכזי בתכנת, Context, והעברה שלו בין מחלקות וגורמים שונים במהלך הריצה. פרויקט זה מבוסס על סוג זה של תכנות באמצעות יצירה של מבני נתונים מסוג מילון dictionary.
- **UUID** – הוא מספר פסאודו אקראי המשמש לזיהוי ייחודי של אובייקט בכל הקשר בו הוא עשוי להופיע. קיימים 2^{128} מזהים כאלה כך שאף על פי שיתכן שמזהה מסוים יוגרל פעמיים הסבירות לכך נמוכה מאוד. בפרויקט אני משתמש בזיהוי הזה כדי לייחד Disks ו Volumes.
- **קובץ Configuration** – הוא קובץ המכיל מידע שמשמש כהגדרות של יישום מחשב אחד או יותר. לכל שרת בפרויקט יש קובץ קונפיגורציה המתאר את התצורה בה הוא עולה.
- **מטמון - Cache** – אוסף נתונים על בסיס ערכים מקוריים אשר מאוחסנים במיקום אחר, או שהופקו קודם לכן באמצעות חישוב כלשהו. השימוש במטמון מאפשר שליפה מחודשת של המידע במהירות במקום לחזור אל המאגר המקורי שהוא יחסית איטי או מרוחק. השימוש ב Cache בפרויקט מתבצע לשמירת בלוקים אשר אמורים להיכתב לדיסק מנותק.
- **תכנות מונחה אירועים - Event driven development** – תפיסה בתכנות, אשר על פיה בתוך תוכנית המחשב קיימים חלקים, הממתינים לקבלת אות. האות נקרא "אירוע" (event) והוא מתקבל כאשר מתרחש אירוע מסוים במערכת, אליו קשוב היישום.
- **תרשים רצף - Sequence diagrams** – תרשים רצף הנועד לתאר תהליך ולהסביר כיצד המערכת ורכיביה מבצעים תהליך זה.

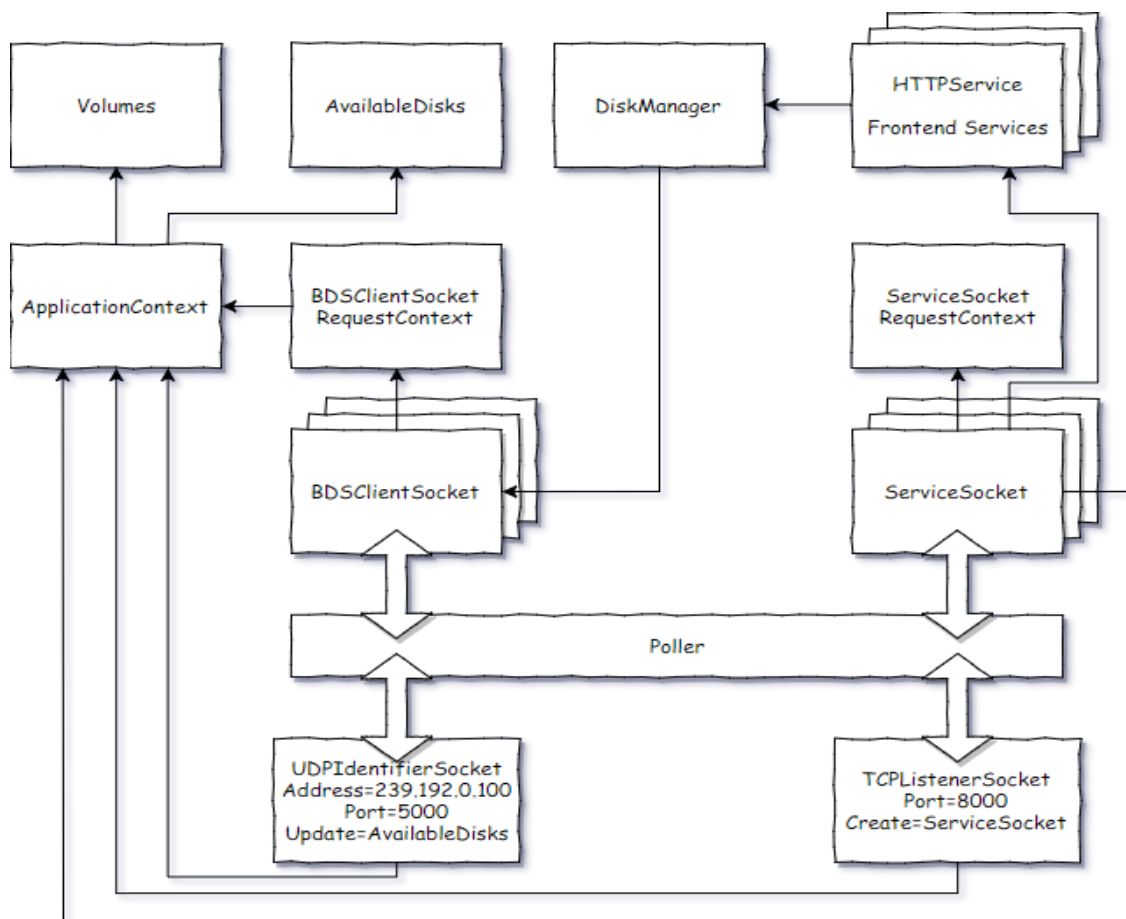
הפרויקט משלב את כל המושגים הללו בתוכו, על מנת לאפשר מערכת דינאמית וחזקה.

מימוש

דיאגרמת בלוקים

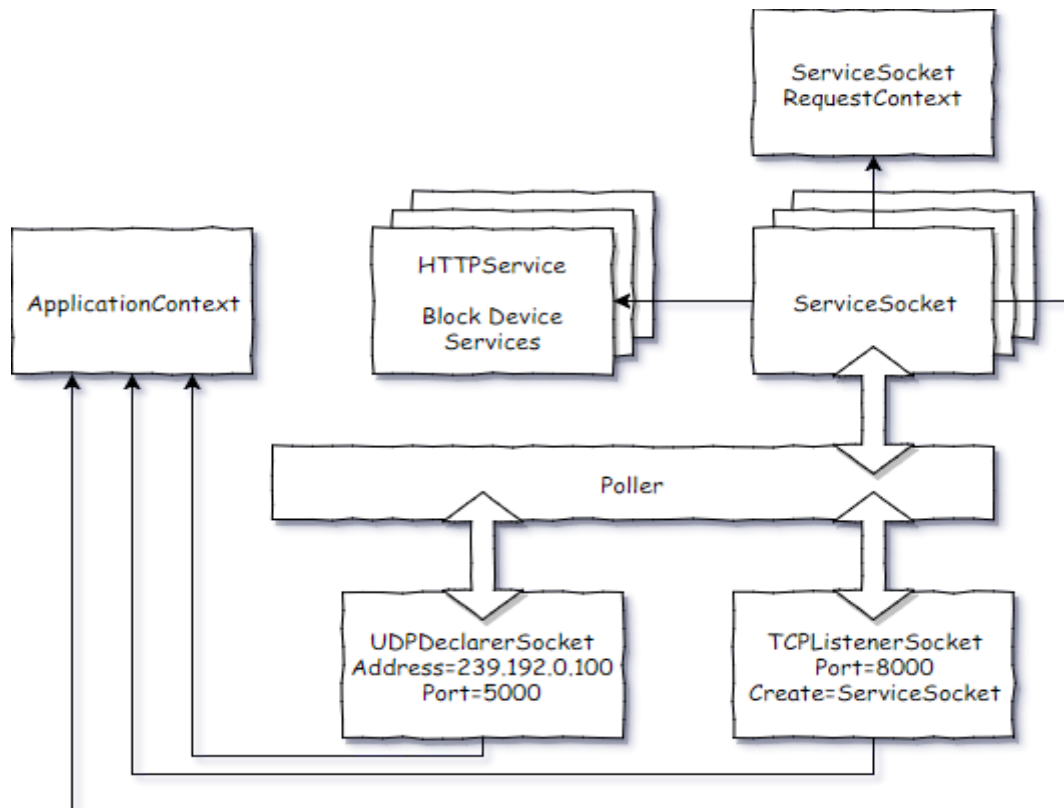
ראשית כל נציג את מעבר הבלוקים במערכת שלנו. במהלך הפרויקט עיצבתי שתי מערכות, אחת של Frontend Server ואחת של Block Device Server. לכל אחת יהיה מעבר בלוקים שונה בתוכנית אך הבסיס של שתיהן יהיה דומה, משום שבסיס של שתיהן יושב poller המאפשר AsynchronousIO.

דיאגרמת בלוקים עבור Frontend Server



בתרשים הזה אנו רואים שה Frontend Services יוצרים pollables בשם BDClientSocket. כך ניגשים ה Frontend Services אל המידע ב Block Device Servers.

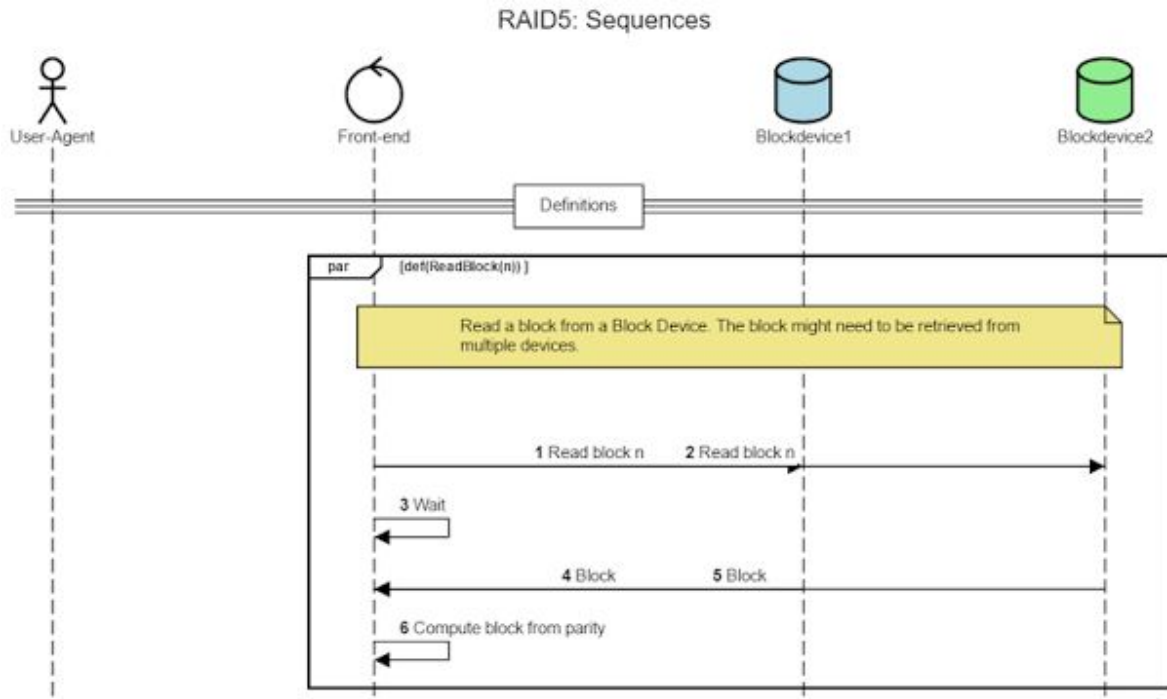
דיאגרמת בלוקים עבור Block Device Server



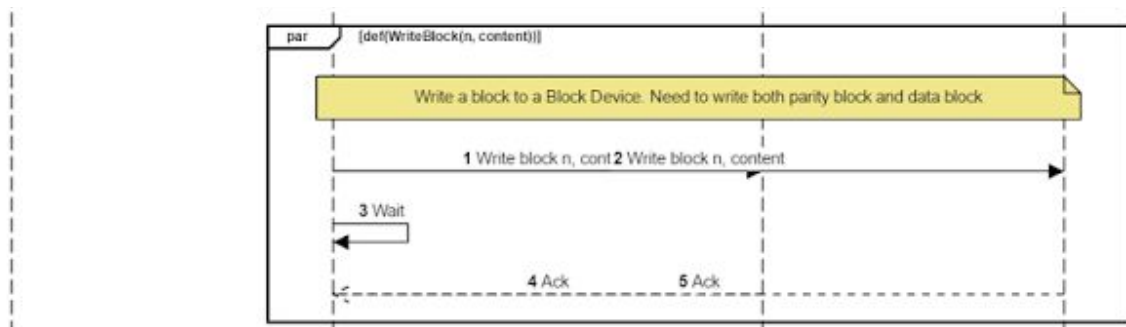
קיים דמיון רב בין שתי המערכות. נשים לב כי בFrontend יש pollable בשם **UDPIdentifierSocket**, ואילו בBlock Device Server יש במקומו pollable בשם **UDPDeclarerSocket**. הסימטריות הזו אינה מקרית, משום שהBlock Device Server מכריז על עצמו בעזרת UDP, והFrontend Server מחפש את ההכרזות הללו ומעדכן את **Available Disks**.

Sequence Diagrams

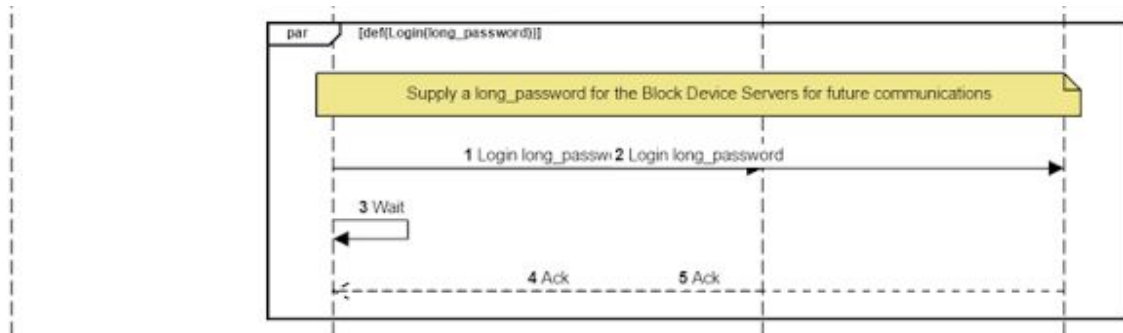
על מנת להמחיש את רצף האירועים בחלקים עיקריים בתוכנית, יצרתי את הדיאגרמות הבאות להמחשתם:



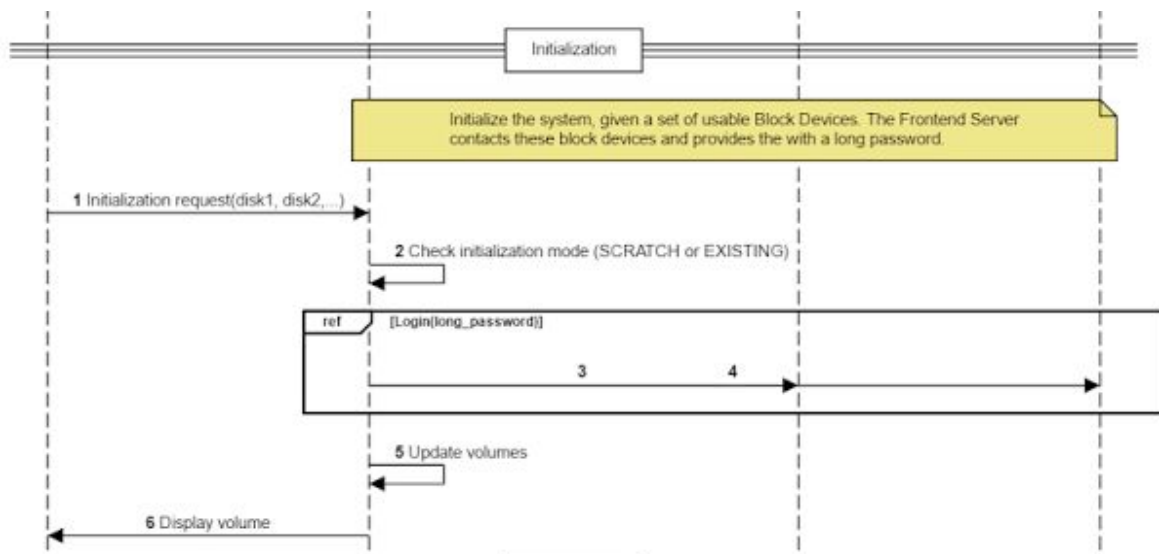
- **ReadBlock** - קריאה של בלוק אחד מ-Block Device מסוים. במקרה של נפילת שרת Block Device, ייתכן ויש לשחזר את הבלוק הרצוי על ידי שאר הבלוקים, כפי שהוסבר קודם לכן.



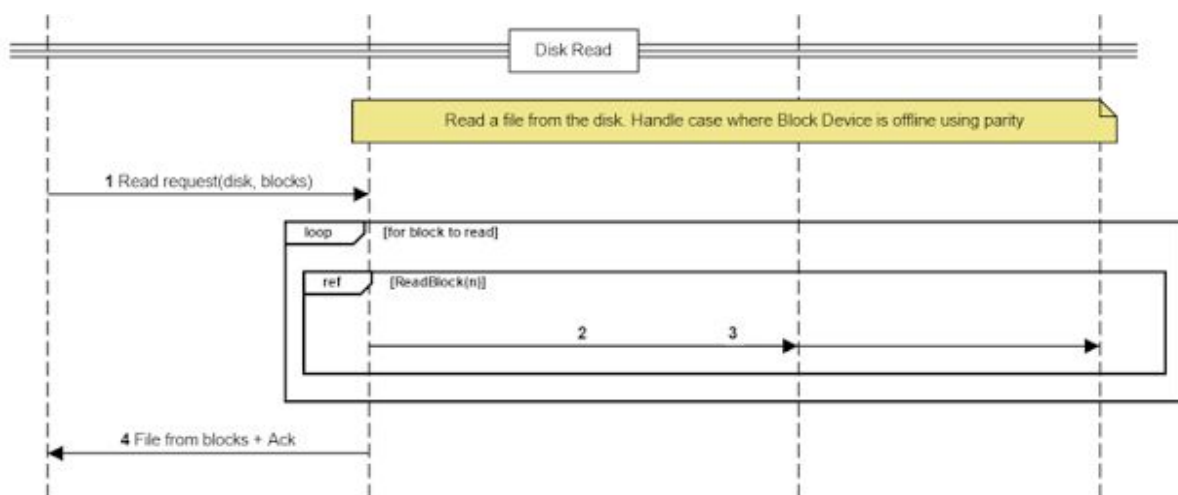
- **WriteBlock** - כתיבה של בלוק אחד ל-Block Device מסוים. בכתיבה יש לכתוב גם את בלוק ה-parity, וגם את בלוק ה-data.



- **Login** - השרת מספק לBlock Devices סיסמא ארוכה כדי לנהל תקשורת עתידית.

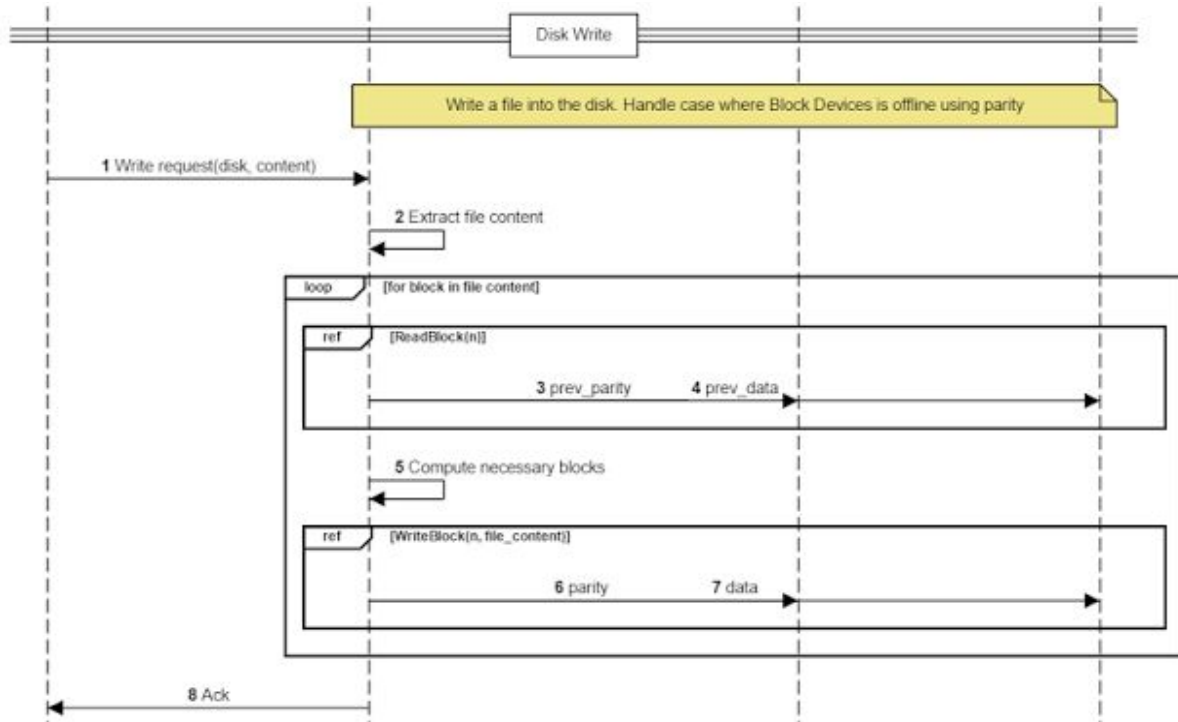


- **Initialization** - שירות HTTP אשר Frontend Server מציע למשתמש. בעזרת שירות זה, יכול ליצור המשתמש volumes לוגיים. במהלך האתחול הזה, מבצע שרת Frontend פעולת Login אל כל אחד משרתי Block Device על מנת להתחיל את התקשורת ביניהם.



- **Disk Read** - שירות HTTP אשר Frontend Server מציע למשתמש. בעזרת שירות זה, יכול המשתמש לקרוא מדיסק לוגי מסויים מספר בלוקים כרצונו. השירות יודע לפנות אל

הדיסקים הפיזיים המתאימים על מנת למלא את הבקשה. יודע לטפל גם במקרה שבו אחד מהדיסקים לא פעיל.

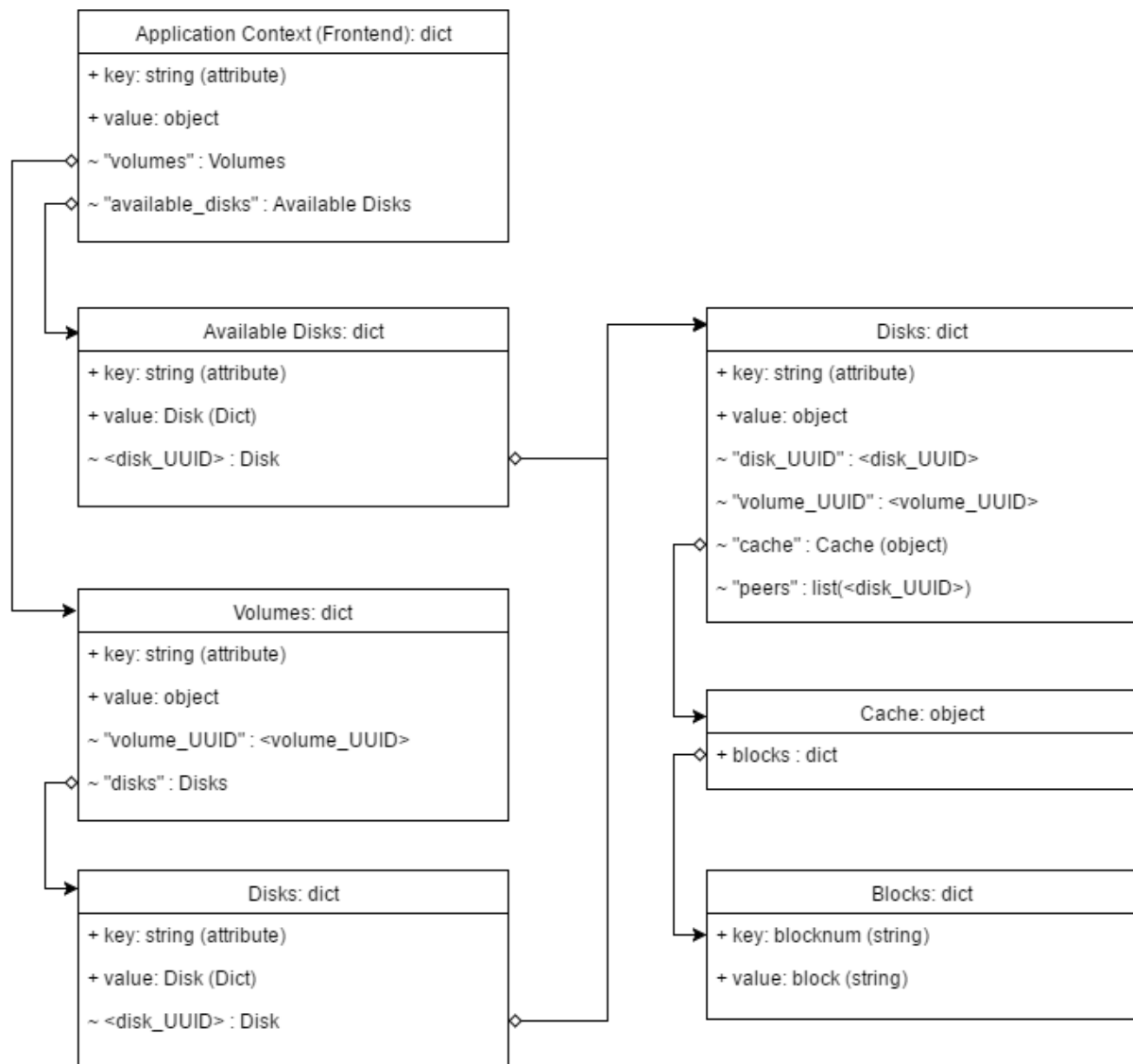


- **Disk Write** - שירות HTTP אשר Frontend Server מציע למשתמש. בעזרת שירות זה, יכול המשתמש לכתוב אל דיסק לוגי מסויים קובץ מסויים. השירות יחלק את הקבץ לבלוקים ויכתוב אותו בבלוקים הפיזיים. יודע לטפל במקרה שבו אחד מהדיסקים לא פעיל.

מבני נתונים

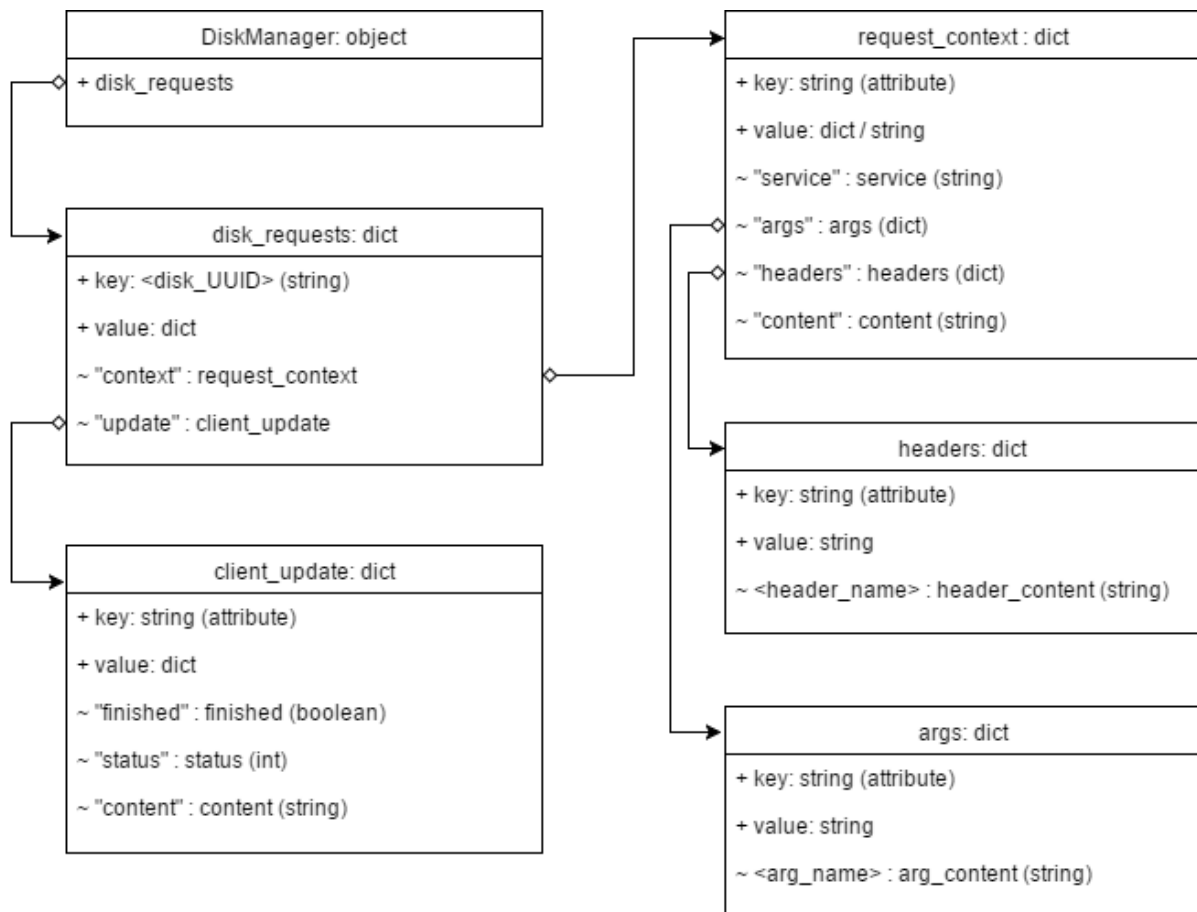
בשל העובדה שהפרויקט שלי מתעסק עם שרתים רבים ובניהול כמויות גדולות של מידע, מבני הנתונים היוו חלק גדול מהמימוש של הפרויקט.

ראשית כל אתייחס למבנה הנתונים אשר מנהל את כל המערכת של ה(Frontend Block Devices) יש מבנה דומה אך פשוט בהרבה). להלן מבנה הנתונים אשר השתמשתי לניהול המערכת וה



בפרויקט אפשרתי למשתמשים ליצור volumes, שהם למעשה יחידות לוגיות המכילות מספר דיסקים קבוע מראש. לכל volume היו את המאפיינים שלו, כגון הדיסקים שהוא מכיל, הסיסמא המשותפת לכל הדיסקים בו, ועוד מידע רב הקושר אותו ליחידה לוגית.

בנוסף למבנה הנתונים הזה, המערכת שלי כללה מבנה נתונים נוסף. כאשר Service מסויים רוצה להתחבר אל Block Devices, עליו ליצור מספר בקשות לכל אחד מהם, ולחכות לתשובה מכל אחד מהם. על מנת לארגן את כל המידע הזורם בין Frontend Services ו-Block Devices, יצרתי מחלקה בשם DiskManager אשר מנהלות את כל החיבורים הללו. להלן מבנה הנתונים אשר המחלקה DiskManager מתחזקת:



ניתן לראות שכל פנייה ל-Block Device שמורה בתוך המילון disk_requests. לכל בקשה יש שני חלקים - request_context, אשר איתו היא פונה אל השרת, וה-client_update, היכן שתשובת השרת נשמרת. באופן כזה, ניתן לנהל מספר רב של בקשות בקלות, ולהבין למשל אם כל הבקשות התקבלו בהצלחה.

פרוטוקולי תקשורת

הפרויקט שלי כלל שימוש במספר פרוטוקולי תקשורת.

UDP/Multicast

באופן דיפולטי, הכתובת בה מכריז Block Device ובה Frontend מאזין היא 239.192.0.100:5000. למעשה היה ניתן לבחור כל port לא שמור, וכל כתובת multicast לא שמורה בטווח של כתובות 239.255.255.255 - multicast 224.0.0.0. השימוש בפרוטוקול UDP אינו מקרי. חבילותיו של הפרוטוקול לעיתים הולכות לאיבוד, או מגיעות בסדר מבולגן. הסיבה לבחירתו הוא שהוא פשוט יחסית, והמידע שאנו מעבירים אינו בעל חשיבות רבה מדי, כך שאיבוד packet בדרך לא יפריע ל-Frontend ולא יגרום לו לשכוח את Block Device.

הכרזת שרת Block Device היא מהצורה הבאה:

disk_UUID	\r\n	bind_port	\r\n	volume_UUID	\r\n	\r\n
-----------	------	-----------	------	-------------	------	------

- ה-disk_UUID הוא ID של הדיסק אותו גילינו
- ה-bind_port הוא port אשר בעזרתו ניתן להתחבר אל השרת (הכתובת כבר ידועה)
- ה-volume_UUID הוא ID של volume אליו משוייך הדיסק. אם הדיסק עוד לא מאוחלל לאף volume, יוחזר כאן סטרינג ריק.

אם גילינו דיסק אשר טרם אוחלל, ה-Frontend Server ישמור אותו תמיד אצלו, ויציע למשתמש לאתחל אותו יחד עם דיסקים אחרים לא מאוחללים. אם מדובר ב-volume_UUID אשר אינו סטרינג ריק, ה-Frontend Server יראה אם הוא מכיר את ה-volume הזה, ואם כן יוסיף את הדיסק הזה לרשימות שלו. אחרת, יתעלם מהדיסק הזה משום שהוא לא מכיר אותו וככל הנראה שייך ל-Frontend Server אחר.

HTTP Services

כל הפונקציונאליות של הפרויקט נמצאת בידי ה-HTTP Services, אשר מציעים למשתמש שירותים שונים לניהול המערכת.

השירותים ייקראו באמצעות שתי מתודות של HTTP:

(1) GET - מרבית השירותים משתמשים במתודה הזו

(2) POST - שירות כתיבת קובץ לדיסק והעלאת קובץ נעשים במתודה הזו

השורה הראשונה בבקשת ה-HTTP שונה בין שתי המתודות:

GET /service_name?parameters HTTP/1.1

POST /service_name HTTP/1.1

השוני המרכזי בין שני המתודות הוא בתצורת העברת הארגומנטים. כפי שניתן לראות, במתודת GET, הארגומנטים מועברים בשורה הראשונה, ואילו במתודת POST הם מועברים גוף ההודעה. שירותים אשר נכתבו במתודת POST הם מסוג multipart/form-data. זהו מבנה אשר בעזרתו ניתן להעביר forms בעזרת POST, אשר יכול לכלול בתוכו קבצים. התוכן של form מהצורה הזאת מחולק לתתי חלקים, המופרדים על ידי "boundary" באופן הבא:

boundary	--AaB03x			
headers	content-dispositio n	form-data;	name="field1"	
content	Roy Zohar			
boundary	--AaB03x			
headers	content-dispositio n	form-data;	name="pics"	filename="file1.txt "
content	... contents of file1.txt ...			
end_boundary	--AaB03x--			

כך שהארגומנטים גם הם מועברים ב-contents. לדוגמא, הארגומנט "field1" עבר ב-form כאן, וערכו Roy Zohar. לאחר מכן הועבר קובץ ששמו file1.txt.

להלן רשימת השירותים אשר שרת ה-Frontend מציע:

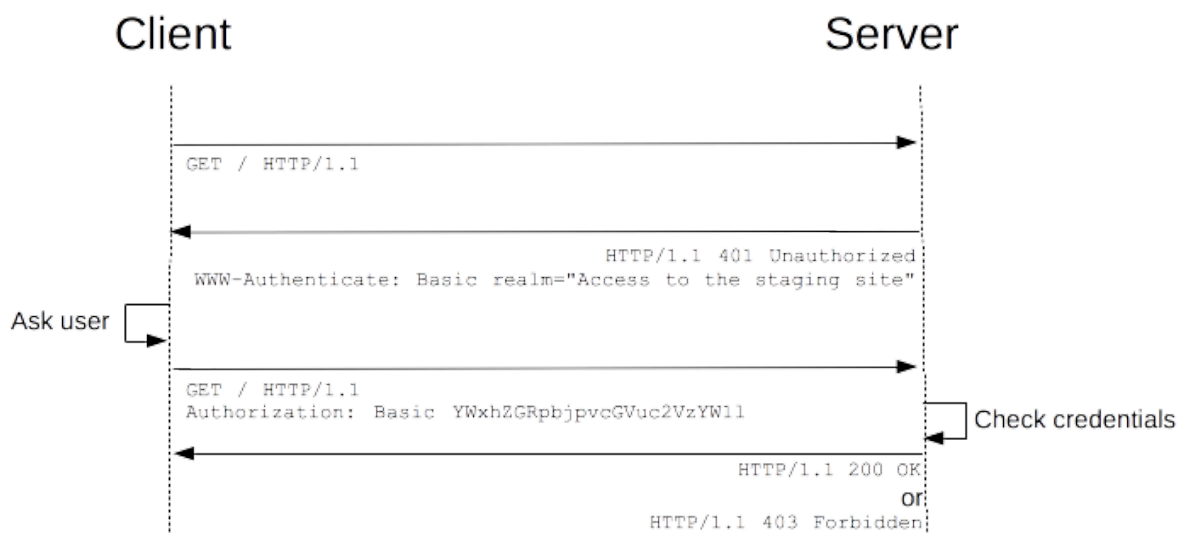
השירות	מתודה	שם השירות	מה השירות עושה	ארגומנטים
--------	-------	-----------	----------------	-----------

הצגת קובץ	GET	"שם הקובץ המבוקש"	התוכן שיוחזר הוא הדף שהתבקש. אלו בעיקר קבצי ניווט במערכת, קבצי עיצוב, תמונות ועוד. שירות זה ייקרא בהיעדר שירות אחר שנמצא.	אין
שירות הצגת הדיסקים	GET	display_disks	התוכן שיוחזר יכיל דף HTML עם טבלה המתארת את כל ה disks שה Frontend גילה, וה volumes שאותחלו במערכת.	אין
שירות אתחול מערכת	GET	init	בהינתן UUID's של דיסקים מסויימים, המערכת מנסה ליצור volume לוגי מהם. הדף שיוחזר הוא אותו אחד ב display_disks.	דיסקים שנבחרו מהצורה: name: disk(disknum) val: disk_UUID בנוסף קיים ארגומנט בוליאני בשם scratch אשר מציין אם לאתחל את ה volume מחדש.
שירות קריאה מדיסק	GET	disk_read	בהינתן מיקום של דיסק במערכת, המערכת קוראת מספר בלוקים מסויים ממקום מסויים מהדיסק הלוגי שהתבקש.	<ul style="list-style-type: none"> volume_UUID disk_UUID firstblock blocks
שירות כתיבה לדיסק	POST	disk_write	בהינתן מיקום של דיסק במערכת, המערכת כותבת קובץ מסויים אל הדיסק במיקום שהתבקש	<ul style="list-style-type: none"> volume_UUID disk_UUID firstblock file
שירות ניתוק דיסק	GET	disconnect	בהינתן דיסק מסויים ב volume , השירות ינתק את ה disk מה	<ul style="list-style-type: none"> volume_UUID disk_UUID

	volume, ויגדיר אותו במצב .OFFLINE			volumen
<ul style="list-style-type: none"> volume_UUID disk_UUID 	<p>בהינתן דיסק מסויים בvolume</p> <p>, השירות יחבר את הdisk</p> <p>חזרה אל הvolumen, ויגדיר אותו במצב ONLINE. השירות יבנה אותו במידת הצורך</p>	connect	GET	<p>שירות</p> <p>חיבור</p> <p>דיסק</p> <p>לvolume</p>

הזדהות

Basic Access Authentication מתבצעת באופן הבא:



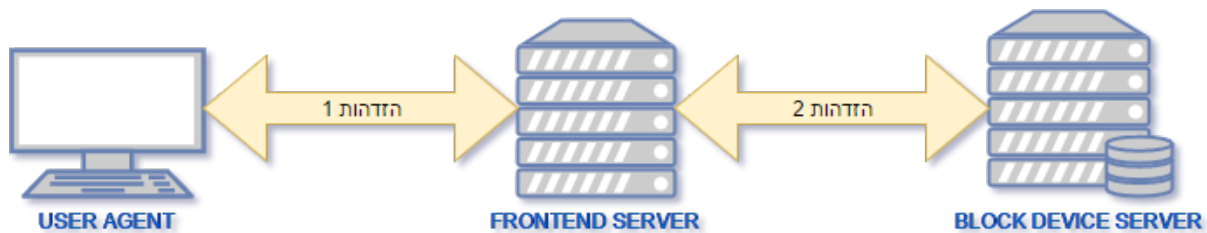
- 1) הדפדפן מבקש מהשרת שירות כלשהו.
- 2) אם השירות הנ"ל דורש הזדהות, השרת יחזיר תשובה 401 Unauthorized.
- 3) הדפדפן מבקש מהמשתמש להזדהות בפניו (שם משתמש וסיסמא).
- 4) הדפדפן שולח אל השרת בקשה מעודכנת, כאשר הפעם הheader של Authorization מכיל שם משתמש וסיסמא, מקודדים בbase64:

	Username	:	Password
Regular	Alladin	:	OpenSesame

Base64 Encoded	QWxhZGRpbjPcGVuU2VzYW1l
----------------	-------------------------

(5) השרת בודק את פרטי ההזדהות ומגיב בהתאם.

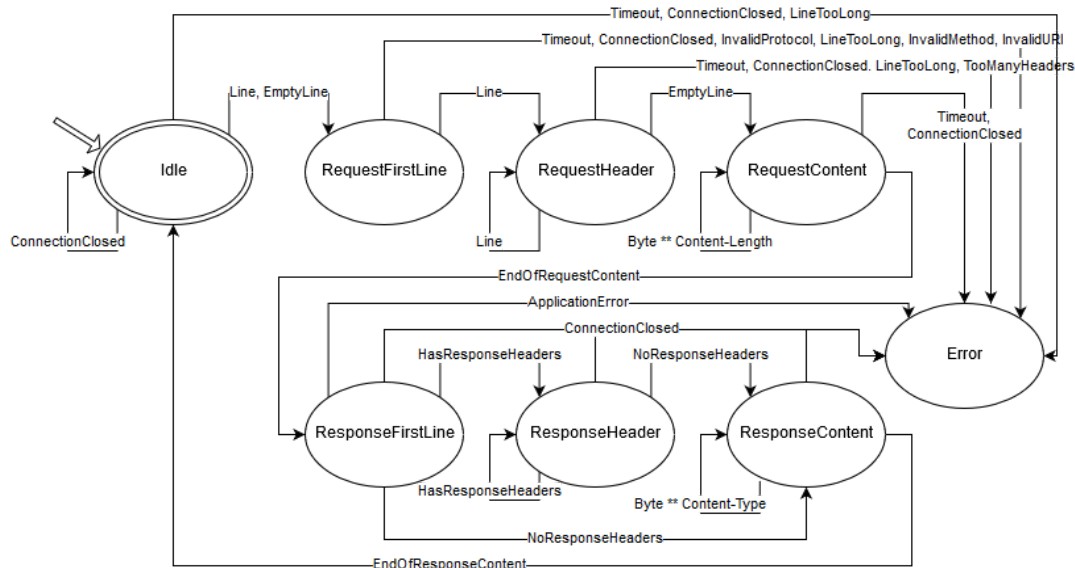
כל ההזדהות בפרויקט מתבצעת על ידי Basic Authentication. ניתן לדבר על הזדהות בשני תווכים:



- הזדהות 1 - הזדהות של המשתמש מול הFrontend Server. המשתמש והסיסמא אשר על פי הם מתבצעת ההזדהות נמצאים בConfiguration file של Frontend Server. הזדהות זו נועדה למנוע התחברות של clients לא רצויים לFrontend Server של המשתמש.
- הזדהות 2 - הזדהות של Frontend Server מול Block Device Server. הזדהות זו נועדה כשכבת אבטחה בין השרתים, המספקת לBlock Device Server ביטחון לגבי מי מבקש ממנו מידע. למעשה ההזדהות הזו מתבצעת בשני שלבים:
 - השלב הראשון מתבצע בזמן האתחול של Volumes של Block Device Server. הוא חלק ממנו. כחלק משלב זה, שולח Frontend Service סיסמא ארוכה לBlock Device Server אשר תשמש לתקשורת ביניהם.
 - השלב השני הוא שלב ההזדהות בכל פעם שהFrontend ניגש לBlock Device. כל גישה לאחר השלב הראשון, תתבצע אך ורק עם הזדהות לBlock Device.

מכונות מצבים

בשל האופי האסינכרוני של השרתים, היה עליי לממש מכונות מצבים. הסיבה לכך היא שאנו לא מקבלים אם כל input בבת אחת, ולכן קיים צורך בשמירת מצב נוכחי בחלקים מסויימים בתוכנית. אחת ממכונות המצבים העיקריות בתוכנית היא זו של HTTP States המוצגת כאן:

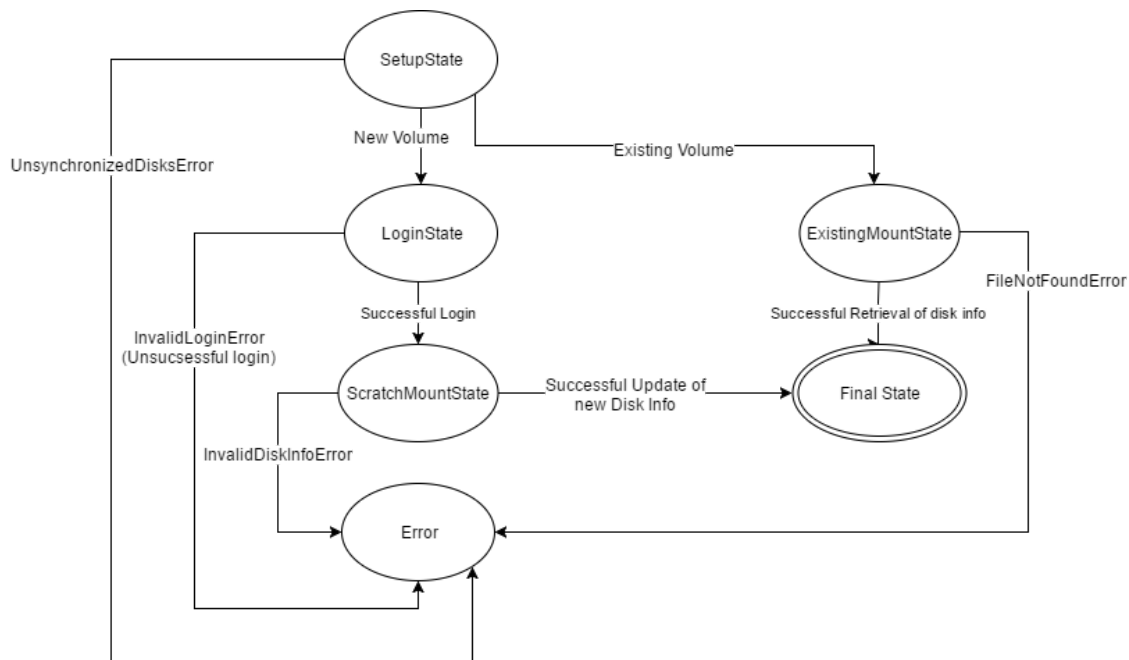


מכונת המצבים הזו מפרקת בקשת HTTP בשלבים, ומחזירה תשובה מתאימה. התשובה המתאימה תיקבע כמובן על ידי HTTP Services.

מצב	הסבר
Idle	שלב המתנה לבקשה.
Request First Line	שלב קבלת שורת הסטטוס. השרת מקבל את שם השירות המבוקש.
Request Header	שלב קריאת ה-Headers. מכילים מידע אודות סוג ואורך ההודעה המתקבלת ועליהם מבוסס הפרוטוקול.
Request Content	שלב קריאת תוכן ההודעה.
Response First Line	שלב שליחת שורת הסטטוס של התגובה. השרת מודיע ללקוח על כישלון או הצלחה בביצוע השירות.
Response Headers	שלב שליחת ה-Headers. השרת שולח ללקוח מידע אודות התגובה.

Response Content	שלב שליחת תוכן התגובה (עבור ממשק המשתמש, תוכן התגובה יהיה דף HTML המכיל את הדף הרצוי).
Error	במידה באחד מהשלבים שתוארו התקבלה שגיאה בשרת, השרת מנתק את החיבור.

למעשה, כמעט כל HTTP Service אשר ניגש אל Block Devices, מממש State Machine מסוג כזה או אחר. זה מכיוון שעליו לחכות לשתובת השרת ולשמור את מצבו הקיים, אך בו זמנית לא לתקוע את התוכנית.. להלן דוגמא למכונת מצבים אחת מהפריויקט, של InitService (שירות אתחול של volume):



במכונת המצבים קיימים שני מצבים לאתחול volume:

1. אתחול volume מדיסקים חדשים (SetupState -> LoginState -> ScratchMountState -> FinalState)
2. אתחול volume ממצב קיים (SetupState -> ExistingMountState -> FinalState)

מצב	הסבר
SetupState	בדיקה של Disks אשר התקבלו בארגומנטים. יש לבדוק אם הדיסקים כבר שייכים לvolume מסויים ואם כן לוודא שהם שייכים לאותו בלוק, אחרת

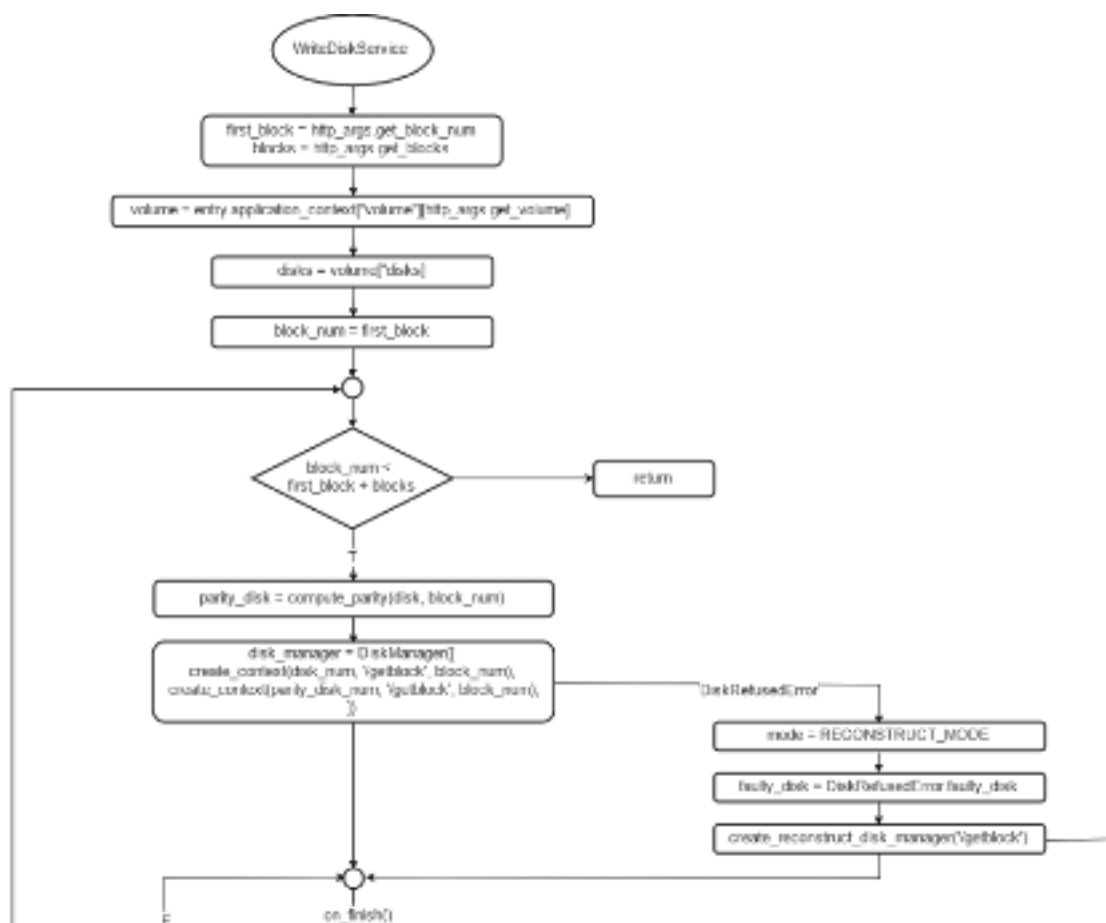
תקרה UnsynchronizedDisksError .	
שלב ההזדהות הראשוני מול הBlock Device Servers. בשלב זה מספק ה Frontend סיסמא לכל אחד מהBlock Devices כדי לנהל תקשורת מאובטחת עתידית.	LoginState
שלב עדכון קובץ המידע אשר נמצא בBlock Devices. מתבצעת כאן העלאה של קובץ בעזרת מתודת הPOST.	ScratchMountState
שלב קריאת קובץ המידע אשר נמצא בBlock Devices שכבר שייכים ל volume מסויים.	ExistingMountState
שלב שליחת תוכן התגובה (עבור ממשק המשתמש, תוכן התגובה יהיה דף HTML המכיל את הדף הרצוי).	Response Content
במידה באחד מהשלבים שתוארו התקבלה שגיאה בשרת, השירות ישלח שגיאה חזרה ללקוח.	Error

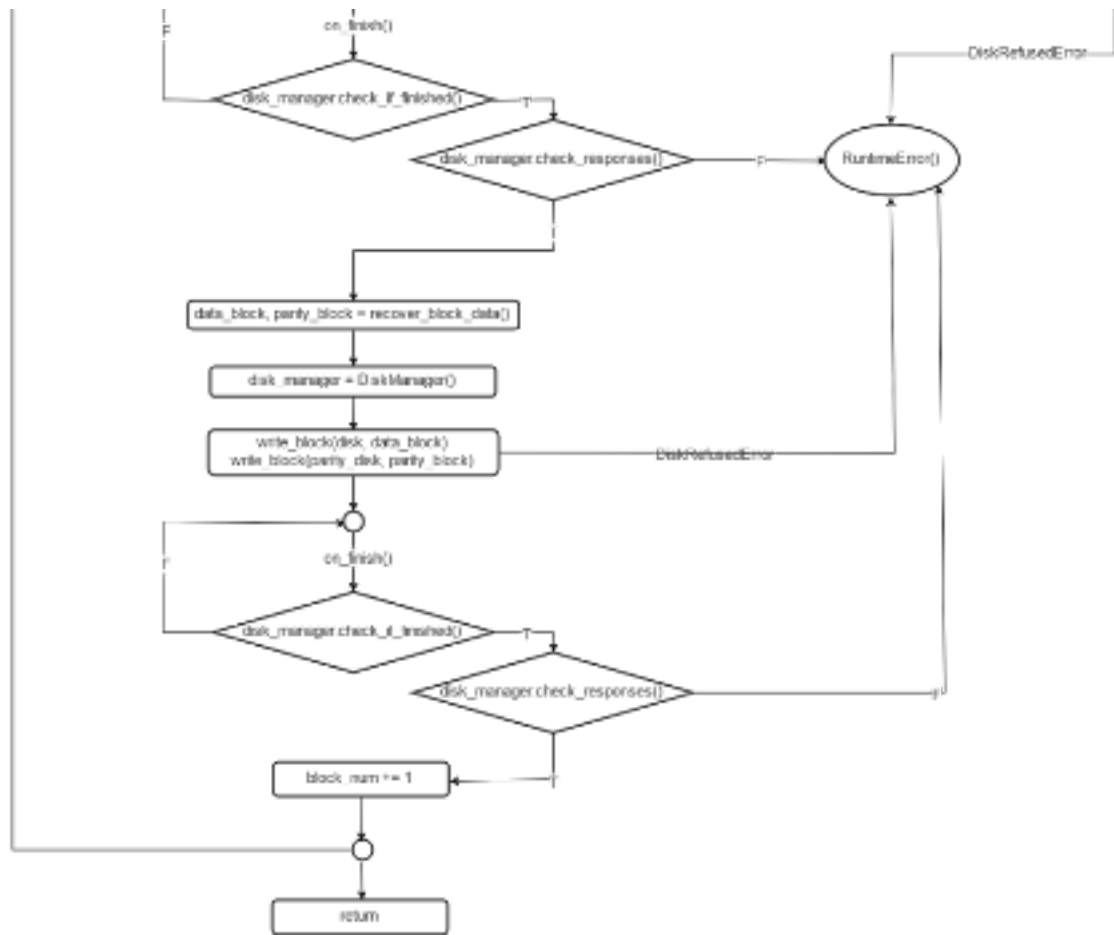
אתגרים במימוש ודרך הפתרון

האתגר הכי גדול בפרויקט היה לממש את Service של כתיבה של קובץ לדיסק, בהתחשב בעובדה שאחד מהדיסקים יכול להיות לא פעיל.

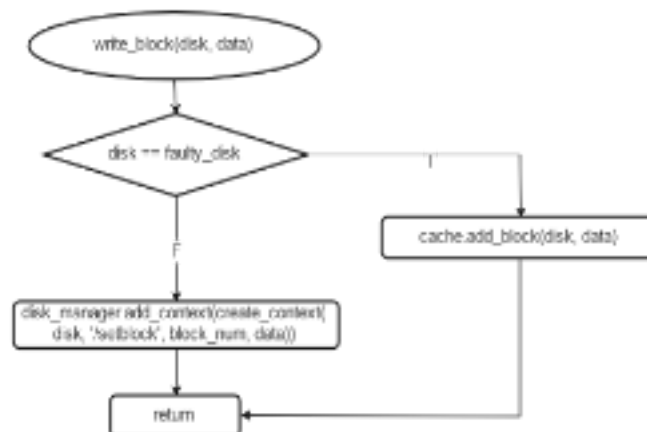
אילו כל הדיסקים פעילים, אין בעיה לגשת לכל אחד מהם ולקרוא/לכתוב. אך במקרה שאחד מהם לא פעיל, יש לבצע פעולות רבות עם שאר הדיסקים כדי לשחזרו (כפי שהוצג בחלק התיאורטי). דרך הפתרון הייתה לחלק את המשימה לתתי משימות קטנות, ואז לנסות להרכיב אותן לפתרון אחד. להלן תרשים זרימה איכותי המתאר את הפתרון לבעיית הכתיבה:

לולאה מרכזית





פונקציות עזר מרכזיות





בעיות ידועות

הבעיה המרכזית בפרויקט היא נפילה של Server. נפצל את הבעיה לשני מקרים:

(1) נפילה של FrontendServer - בעת נפילה של הסרבר המרכזי, התקשורת עם הלקוחות תיפסק מיד, ולא יהיה ניתן לגשת ל-volumes אשר הוא ניהל. כאשר הסרבר יעלה מחדש, נוכל לאתחל שוב את ה-volumes הרלוונטיים ממצבם הקיים ולהמשיך כרגיל. המידע על הדיסקים ב-volumes שהוא ניהל לרוב יהיה שמור ובטוח, אך עולה בעיה במקרה שה Frontend עולה בחזרה למצב אשר בו לא כל הדיסקים היו מסונכרנים (למשל אם אחד היה מנותק לפני הנפילה של השרת).

(2) נפילה של Block Device Server - בעת נפילה של אחד מסרברי הדיסקים, המערכת תתריע ללקוח כי הוא יכול להמשיך לתפעל את המערכת אך אילו עוד דיסק היה מתנתק, הוא לא יוכל להמשיך לכתוב אל הדיסקים יותר. מגבלה זו הינה ידועה, ונובעת מהמבנה של RAID5, אשר מאפשר שחזור של דיסק אחד בעזרת האחרים, אך לא יותר מכך. לכן בעת ניתוק של שני דיסקים, נעצור את המשתמש ונדרוש יעלה את אחד משני השרתים חזרה.

התקנה ותפעול

קבצי Configuration

להלן שתי דוגמאות לקבצי קונפיגורציה, עבור שני השרתים שבמערכת:

config.ini - Frontend Server

Section Name	Key Name	Value Type	Default Value	Explanation
MulticastGroup	address	string	239.192.0.100	כתובת ה-IP שאליה מקשיב IdentifierSocket
MulticastGroup	port	int	5000	פורט שאליו מקשיב ה IdentifierSocket
Authentication	common_user	string	Roy	שם משתמש בין ה UserAgent לבין ה Frontend
Authentication	common_password	string	12345	סיסמא בין ה UserAgent לבין ה Frontend
Volume1	long_password	string	lpkfZ99ynKpEi1G ECaxQRYp1lyMx EyijlClgKOMMNy WDtqEWSa2Un OVuBcqU06W8	סיסמא ארוכה לתקשורת בין Frontend לבין Block Device אשר נמצא ב Volume1
Volume1	volume_uuid	string	f11a1d8f-e35b-44 a0-9be3-d571166 a06d6	UUID של Volume1

config0.ini - Block Device Server

Section Name	Key Name	Value Type	Default Value	Explanation
MulticastGroup	address	string	239.192.0.100	כתובת ה-IP שאליה מקשיב IdentifierSocket
MulticastGroup	port	int	5000	פורט שאליו מקשיב IdentifierSocket
Authentication	common_user	string	Roy	שם משתמש בין ה-UserAgent לבין ה-Frontend
Authentication	common_password	string	12345	סיסמא בין ה-UserAgent לבין ה-Frontend
Authentication	long_password	string	lpkfZ99ynKpEi1G ECaxQRYp1lyMx EyijlClgKOMMny WDtqEWSa2Un OVuBcqU06W8	סיסמא ארוכה לתקשורת בין Frontend לבין Block Device אשר השתייך כבר ל-volume מסויים
Server	volume_uuid	string	f11a1d8f-e35b-44 a0-9be3-d571166 a06d6	ה-UUID של ה-volume אליו משתייך הדיסק.
Server	disk_uuid	string	7cd9b55e-9970-4 44a-926f-7ffe810 a37da	ה-UUID של הדיסק.
Server	disk_info_name	string	block_device/disk s/disk_info0	מיקום קובץ של ה-disk_info

Server	disk_name	string	block_device/disk s/disk0	מיקום קובץ של הדיסק
--------	-----------	--------	------------------------------	---------------------

התקנה

על מנת להריץ את הפרויקט יש לבצע את השלבים הבאים:

1. הורידו את גרסת הפרויקט מתוך releases ב-github. קישור בסוף התיק, תחת הכותרת - קוד פרויקט.

2. הורידו דפדפן אינטרנט מודרני כלשהו.

a. סעיף זה מיועד רק עבור משתמשי Firefox. הגרסה האחרונה של Firefox [מבטלת](#)

באופן דיפולטי את אפשרות ה-Basic Authentication. יש לאפשר זאת באופן ידני:

i. רשמו בתיבת החיפוש את about:config

ii. לחצו על הכפתור "I'll be careful"

iii. מצאו את האפשרות הבאה:

network.negotiate-auth.allow-insecure-ntlm-v1

iv. שנו את ערכה מ-False ל-True.

3. שינוי קבצי הקונפיגורציה אשר פורטו מעל. (אופציונלי). קיים python script אשר מאתחל

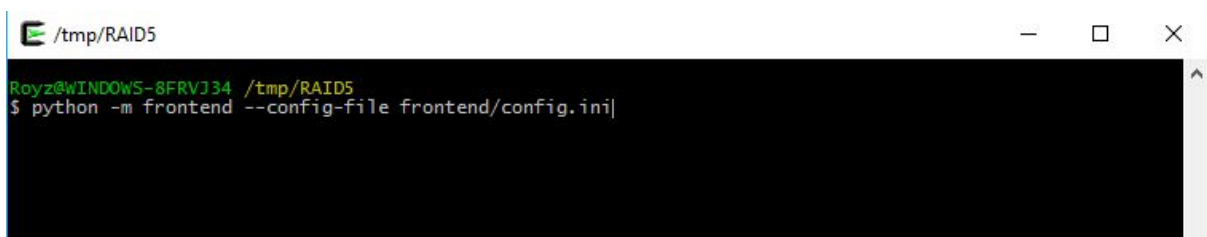
את קבצי הקונפיגורציה מחדש (מוחק כל volume שהיה על ה-Frontend Server, ומגריל

UUID מחדש). ניתן להריץ אותו

פירוט לגבי התפעול

ניתן להריץ את כל השרתים מהcommand line.

הרצת Frontend Server:



```

/tmp/RAID5
Royz@WINDOWS-8FRVJ34 /tmp/RAID5
$ python -m frontend --config-file frontend/config.ini

```

הרצת Block Device Servers:



```

/tmp/RAID5
Royz@WINDOWS-8FRVJ34 /tmp/RAID5
$ python -m block_device --config-file block_device/disks/config1.ini --bind-port 8091

/tmp/RAID5
Royz@WINDOWS-8FRVJ34 /tmp/RAID5
$ python -m block_device --config-file block_device/disks/config0.ini --bind-port 8090

```

ההרצות לעיל מתבצעות רק עם הארגומנטים שהם חובה (required):

עבור Frontend Server:

- Configuration File

עבור Block Device Server:

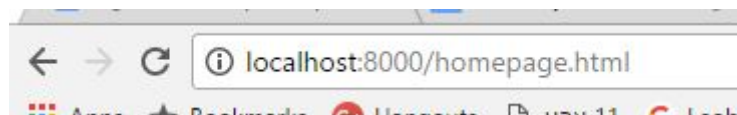
- Configuration File
- Bind port

לרשימת הארגומנטים המלאה ניתן לרשום את השורה:

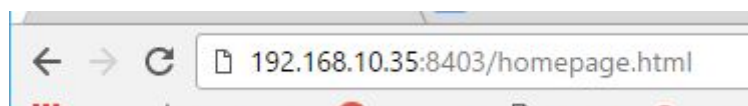
`python -m <SERVER_NAME> --help`

ולבחור את כל הארגומנטים הרצויים.

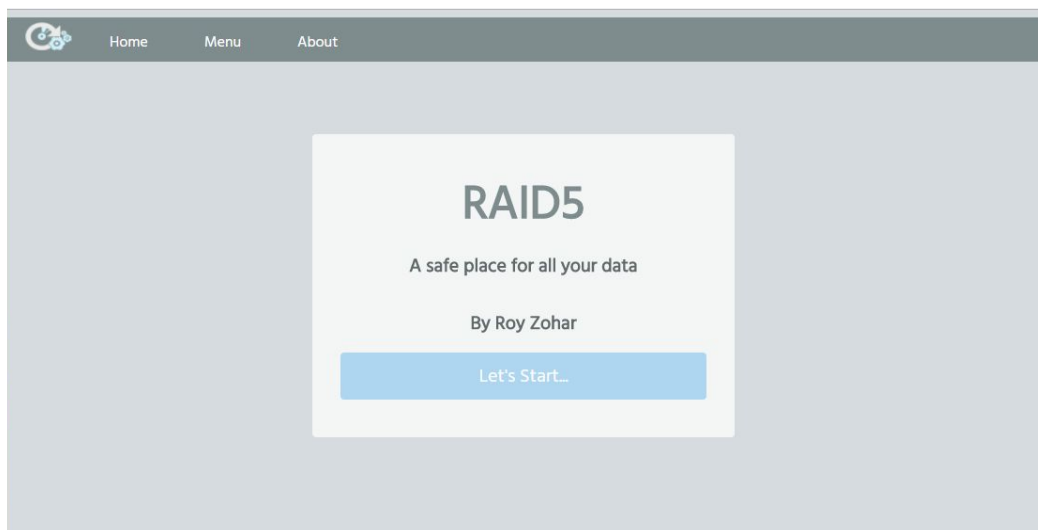
כעת יש ללכת לכל browser, ולכתוב:



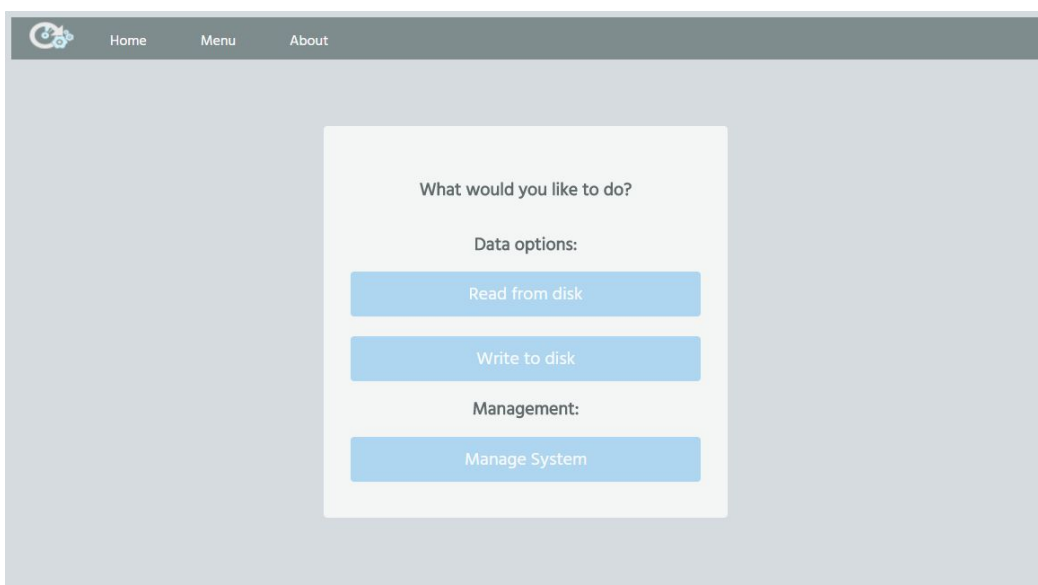
שימו לב שה Frontend Server שלכם רץ בכתובת הרשומה לעיל. למשל עבור Frontend Server שרץ בכתובת 192.168.10.35, בפורט 8403, יש להריץ לכתוב את השורה הבאה:



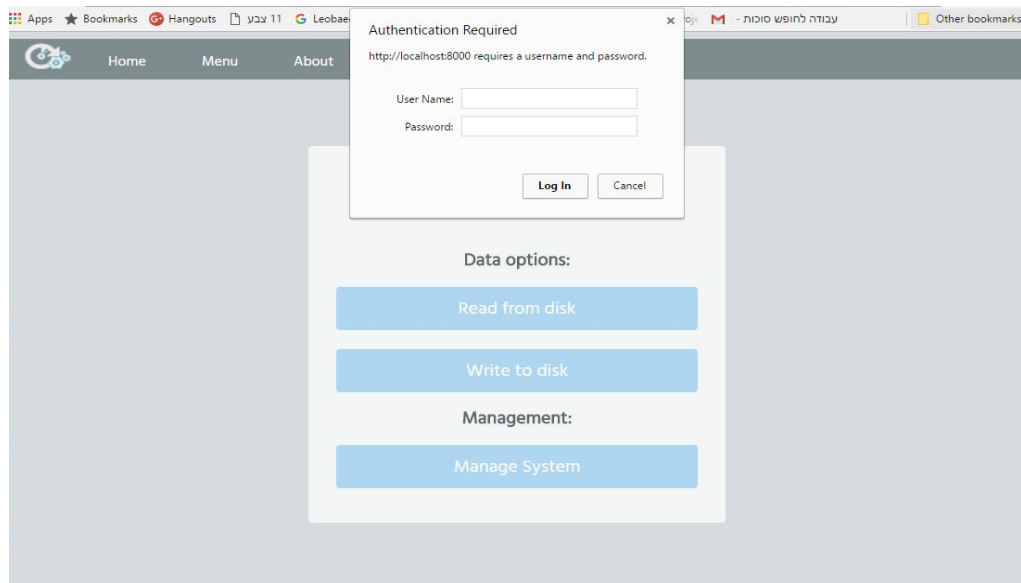
כעת Frontend Server אמור להציג לכם את חלון הכניסה הבא:



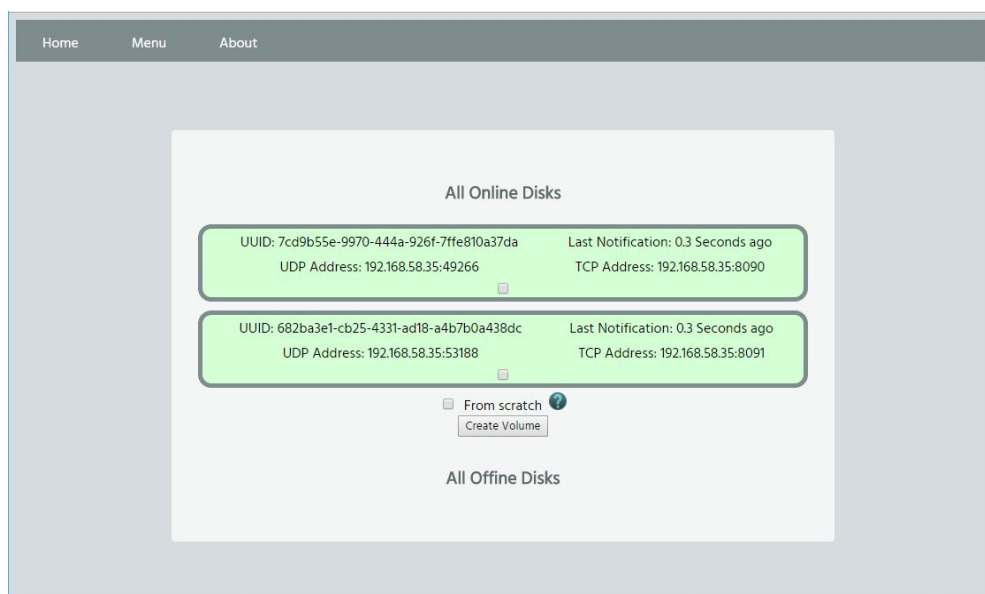
לחצו על כפתור הLet's Start כדי לעבור לתפריט הראשי:



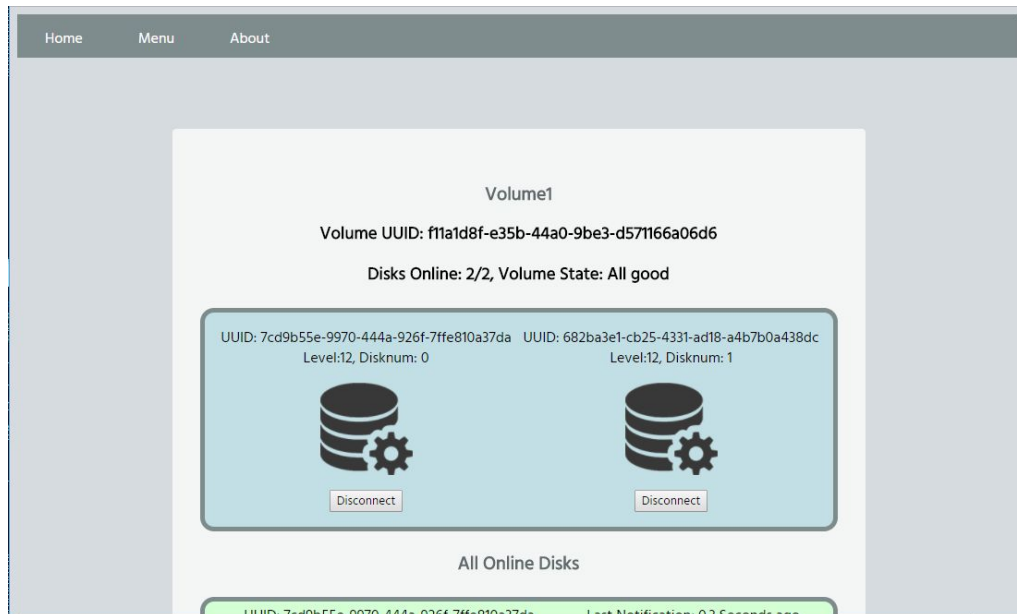
אם תנסו להיכנס לניהול (Manage System), החלון הבא יקפוצ.



תידרשו להכניס שם משתמש וסיסמא על מנת להיכנס לתחום הניהול. אם לא ערכתם את קובץ הקונפיגורציה של Frontend Server, השם משתמש יהיה Roy והסיסמא היא 12345. ניתן כמובן לשנות נתונים אלו בקובץ הקונפיגורציה של Frontend Server לכל שם משתמש וסיסמא שתרצו. לאחר שתכניסו שם משתמש וסיסמא, חלון הניהול יופיע:



בשלב זה מופיעים שני דיסקים במערכת. תוכלו לסמן את שניהם, ולאחר מכן ללחוץ על 'Create Volume' כדי ליצור משניהם Volume לוגי. לאחר שתיצרו את Volume, ממשק הניהול ישתנה מעט וכעת יופיע:



כעת ניתן לראות Volume שנוצר במערכת. כעת ניתן לבצע פעולות רבות על Volume, כגון כתיבה, קריאה, ניתוק וחיבור.

Logging

את מסמך הלוג ניתן למצוא בתיקיה הראשית של הפרויקט. באופן דיפולטי, הלוג מודפס ב command line, אך ניתן להעביר ארגומנט של log file לתוכנית, אשר תיצור את log file ותכתוב אליו:

```

/tmp/RAID5
Royz@WINDOWS-8FRVJ34 /tmp/RAID5
$ python -m block_device --config-file block_device/disks/config1.ini --bind-port 8091 --log-file logger

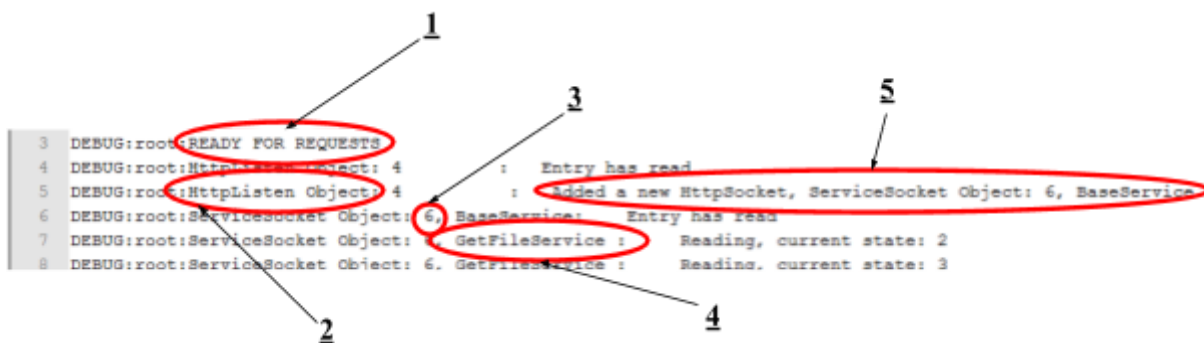
```

כעת כל הדיבאגינג של התוכנה יימצא בתיקייה הראשית בקובץ בשם logger. אם אתם מריצים מספר שרתים מאותו מחשב, מומלץ לתת שמות שונים ל log files כך שלא יהיה logging של שתי תוכניות לאותו קובץ. להלן קובץ logger של שרת Frontend לדוגמא:

```

1 DEBUG:root:STARTED RUNNING..
2
3 DEBUG:root:READY FOR REQUESTS
4 DEBUG:root:HttpListen Object: 4 : Entry has read
5 DEBUG:root:HttpListen Object: 4 : Added a new HttpSocket, ServiceSocket Object: 6, BaseService
6 DEBUG:root:ServiceSocket Object: 6, BaseService: Entry has read
7 DEBUG:root:ServiceSocket Object: 6, GetFileService : Reading, current state: 2
8 DEBUG:root:ServiceSocket Object: 6, GetFileService : Reading, current state: 3
9 DEBUG:root:ServiceSocket Object: 6, GetFileService : Reading, current state: 4
10 DEBUG:root:ServiceSocket Object: 6, GetFileService: Entry has write
11 DEBUG:root:ServiceSocket Object: 6, GetFileService : Writing, current state: 6
12 DEBUG:root:ServiceSocket Object: 6, GetFileService : Writing, current state: 7
13 DEBUG:root:ServiceSocket Object: 6, GetFileService: Entry has write
14 DEBUG:root:ServiceSocket Object: 6, GetFileService : Writing, current state: 10
15 DEBUG:root:HttpListen Object: 4 : Entry has read
16 DEBUG:root:ServiceSocket Object: 6 : Added a new HttpSocket, ServiceSocket Object: 6, BaseService
17 DEBUG:root:HttpListen Object: 4 : Entry has read
18 DEBUG:root:HttpListen Object: 4 : Added a new HttpSocket, ServiceSocket Object: 8, BaseService
19 DEBUG:root:ServiceSocket Object: 6, BaseService: Entry has read
20 DEBUG:root:ServiceSocket Object: 6, GetFileService : Reading, current state: 2
21 DEBUG:root:ServiceSocket Object: 6, GetFileService : Reading, current state: 3
22 DEBUG:root:ServiceSocket Object: 6, GetFileService : Reading, current state: 4
23 DEBUG:root:ServiceSocket Object: 8, BaseService: Entry has read

```



מספר	הסבר
1	הודעה ממחלקת השרת (באותיות גדולות)
2	סוג הpollable שמתעד
3	file descriptor של אותו pollable
4	הHTTP Service של אותו pollable (אם הוא מתעסק בשירותי HTTP).
5	הודעת הדיבאגינג

תוכניות עתידיות

קיימות מספר דרכים אשר בהן הפרויקט הזה יכול להתפתח. להלן מספר תוכניות שהיו לי לגביו, אך מפאת קוצר זמן לא הספקתי לממשן:

ניהול מערכת קבצים:

ניהול המערכת על ידי שמות של קבצים ולא על ידי מספרי בלוקים וגדלים.

- בעת כתיבת קובץ למערכת, המערכת תשמור אותו בסדר מסוים, כך שיהיה ניתן לשחזר את סדר הבלוקים שממנו מורכב הקובץ.
- מיקום הקובץ לא יטריח את המשתמש, והמערכת תדאג לכך.

הצפנת ואבטחת המידע:

במבנה הנוכחי, המידע של הלקוח לא מוגן מפני האזנות והפרעות בתקשורת, ולכן כל אחד אשר מצליח להקשיב לערוצים שבין הסרברים חשוף למידע האישי של הלקוח. למשל כדי לשלוח את סיסמת הלקוח והסיסמא הארוכה בין Block Devices, כל מה שעל הפורץ לעשות הוא לבצע encode בבסיס 64, משום שהמידע לא מוצפן. הייתי רוצה לבצע את התוספות הבאות לקוד על מנת לאפשר אבטחה חזקה יותר.

- שימוש בSSL וHTTPS כדי לאבטח את המידע של הלקוח.
- הצפנת המידע בשרתי קצה (Block Devices).
- הזדהות חזקה יותר מ-Basic Authentication, המשלבת Cookies למשל.
- יצירת אפשרות למספר משתמשים.

פרק אישי

זו הייתה הפעם הראשונה אשר יצא לי לכתוב פרויקט בסדר גודל כזה, ואני למדתי ממנו המון. ראשית כל, למדתי המון רקע תיאורטי בתקשורת ובמערכות הפעלה. בנוסף לכך, רכשתי ניסיון בבניית פרויקט גדול, בחילוק עבודה, ובפירוק המשימה הגדולה לתתי משימות קטנות אשר ניתן לבצע כל אחת מהן בנפרד.

למדתי להשתמש בכלים רבים, כגון Github, Doxygen, ConfigurationFile, אשר לא הכרתי קודם לכן, ולמדתי להיעזר במקורות אינטרנטיים כאשר נתקלתי בבעיות.

במהלך הפרויקט ניסיתי לפשט את המימוש כמה שניתן, כדי שהתוצאה הסופית תהיה כמה שיותר ברורה ואסתטית. למדתי במהלך הפרויקט שארגון, סדר ותכנון, אפילו לפני תחילת הקידוד עצמו, הם תהליכים חשובים מאוד. במהלך הפרויקט נתקלתי לא פעם בבאגים אשר לא הצלחתי להסביר, אך למדתי לדבג אותם באופן יעיל ולרדת למקור הבעיה במהירות.

הפרויקט עורר בי סקרנות רבה בתחום הסייבר, ויצר אצלי התעניינות רבה בנושא. לא פעם מצאתי את עצמי מתעניין בתחום, וחושב על דרכים בהן אוכל לשפר את הפרויקט שלי.

לא ניתן לכמת את כמות התמיכה, העזרה והתדריך של המנחים שלי לפרויקט, שרית לולב ואלון בר לב, אשר ליוו אותי במהלך כל שלב בפרויקט בפרט, ובשלוש השנים האחרונות בכלל. המון תודה על הכל.

תיעוד קוד + קוד הפרויקט

ניתן למצוא את כל הקוד של הפרוייקט, ואת התיעוד שלו בקובץ הבא:

<https://github.com/Royz2123/RAID5/releases/tag/beta>

התיעוד נמצא בתיקייה המכוסת documentation1.0.zip. על מנת לצפות בדוקומנטציה, יש לפתוח את התיקייה, לעשות Extract All, ולפתוח את קובץ index.html.

נספחים

נספח א' - Sequence Diagram Source

לשימוש באמצעות <http://sequencediagram.org>

```

title RAID5: Sequences

actor User-Agent
control Front-end
database Blockdevice1 #lightblue
database Blockdevice2 #lightgreen

==Definitions==

autonumber 1
par def(ReadBlock(n))
    note over Blockdevice2,Front-end #khaki:Read a block from a Block Device. The block
might need to be retrieved from\nmultiple devices.
    parallel
        Front-end->>Blockdevice1: Read block n
        Front-end->>Blockdevice2: Read block n
    parallel off
        Front-end->>Front-end: Wait
    parallel
        Blockdevice1->>Front-end: Block
        Blockdevice2->>Front-end: Block
    parallel off
        Front-end->>Front-end: Compute block from parity
    end
end

autonumber 1

par def(WriteBlock(n, content))
    note over Blockdevice2,Front-end #khaki:Write a block to a Block Device. Need to write
both parity block and data block.
    parallel
        Front-end->>Blockdevice1: Write block n, content
        Front-end->>Blockdevice2: Write block n, content
    parallel off
        Front-end->>Front-end: Wait
    parallel
        Blockdevice1-->>Front-end: Ack
        Blockdevice2-->>Front-end: Ack
    end
end

```



```

        parallel off
    end

    autonumber 1

    par def(Login(long_password))
        note over Blockdevice2,Front-end #khaki:Supply a long_password for the Block Device
        Servers for future communications.
        parallel
            Front-end->Blockdevice1: Login long_password
            Front-end->Blockdevice2: Login long_password
            parallel off
            Front-end->Front-end: Wait
            parallel
                Blockdevice1-->>Front-end: Ack
                Blockdevice2-->>Front-end: Ack
            parallel off
        end
    end

    ==Initialization==
    note over Blockdevice2,Front-end #khaki:Initialize the system, given a set of usable Block
    Devices. The Frontend Server contacts these block devices and provides the with a long
    password.

    autonumber 1
    User-Agent->Front-end:Initialization request(disk1, disk2,...)
    Front-end->Front-end:Check initialization mode (SCRATCH or EXISTING)
        ref Login(long_password)
        parallel
            Front-end->Blockdevice1:
            Front-end->Blockdevice2:
        parallel off
    end
    Front-end->Front-end:Update volumes
    User-Agent<-Front-end:Display volume

    ==Disk Write==
    note over Blockdevice2,Front-end #khaki:Write a file into the disk. Handle case where Block
    Devices is offline using parity.

    autonumber 1
    User-Agent->Front-end:Write request(disk, content)
    Front-end->Front-end:Extract file content
    loop for block in file content
        ref ReadBlock(n)
        parallel

```

```

        Front-end->Blockdevice1: prev_parity
        Front-end->Blockdevice2: prev_data
    parallel off
end
Front-end->Front-end:Compute necessary blocks
ref WriteBlock(n, file_content)
    parallel
        Front-end->Blockdevice1: parity
        Front-end->Blockdevice2:data
    parallel off
end
end
User-Agent<-Front-end:Ack

==Disk Read==
note over Blockdevice2,Front-end #khaki:Read a file from the disk. Handle case where Block
Device is offline using parity.

autonumber 1
User-Agent->Front-end:Read request(disk, blocks)
loop for block to read
    ref ReadBlock(n)
        parallel
            Front-end->Blockdevice1:
            Front-end->Blockdevice2:
        parallel off
    end
end
end
User-Agent<-Front-end:File from blocks + Ack

```