**מגישים:** דביר דוד ביטון

יואב פרץ

**גיטהאב:** https://github.com/dbiton/LedgerProject

**מערכות מבוזרות**

**תרגיל בית 2**

**CLIENT**

**SHARDS**

**LEADER**

**FOLLOWER**

**FOLLOWER**

**FOLLOWER**

**ארכיטקטורה**

המערכת מורכבת ממספר רסיסים (shards), כאשר כל רסיס אחראי על תת-תחום במרחב כתובות הארנקים. הרסיסים מהווים חלוקה של המרחב (כלומר, אין חיתוך בתחומי הרסיסים). כל רסיס מורכב ממספר שרתים, כאשר שרת אחד זוכה לתפקיד המנהיג.

כאשר אחד מהשרתים מהמערכת מקבל בקשת כתיבה (כאשר מבוצעת עסקה) נבדקת הכתובת של הארנק **המשלם** בעסקה, בכל עסקה ארנק יחיד משלם וארנק אחד או יותר **מקבלים**. הבקשה מופנית אל הרסיס שכתובת הארנק המשלם שייכת לתחום שלו. הבקשה מופנית אל המנהיג של הרסיס, אשר מתעד את הפעולה (בהינתן שהיא חוקית) בזיכרון שלו. לאחר מכן, **המנהיג** מעביר את הבקשה אל **העוקבים**, שאר השרתים שברסיס, ורק אז הם מתעדים את הפעולה בזיכרון שלהם.

התוצאה (הצלחת תיעוד העסקה או כישלון בהינתן עסקה לא חוקית) חוזרת אל הלקוח ששלח את הבקשה בכיוון ההפוך דרך השרתים בשרשרת אל המנהיג.

כמובן שבהינתן השרת שמקבל את בקשת הכתיבה וכתובת הארנק המשלם בעסקה, יכולות להיחסך הודעות בין שרתים.

כאשר אחד מהשרתים במערכת מקבל בקשת קריאה (כאשר אנחנו רוצים להחזיר מידע על העסקאות המאוכסנות במערכת), באופן דומה הבקשה תופנה אל שרת ברסיס הרלוונטי – אולם, לא תתבצע הפניה אל המנהיג. השרת יחזיר תשובה על פי העסקאות שמאוכסנות בזיכרון המקומי שלו ולא יעביר הודעות נוספות לשרתים אחרים ברסיס.

התוצאה תוחזר אל הלקוח בדומה לבקשת הכתיבה, אולם הפעם התוצאה יכולה להיות גם רשימה של עסקאות, רשימה של העברות שלא בוזבזו ועוד.

השרתים מאכסנים בזיכרון שלהם שתי טבלאות – עסקאות שהתקיימו והעברות לא מבוזבזות. נרחיב בהמשך על אופן המימוש של המערכת.

**תקשורת וניהול**

**gRPC**

התקשורת בין השרתים מבוצעת באמצעות gRPC. בעת יצירת שרת אנחנו מעבירים את הכתובות והפורטים של כל השרתים האחרים במערכת (גם של השרת עצמו – נסביר למה בהמשך), ובאמצעות הכתובות הנ"ל השרת יכול לשלוח הודעות לשרתים אחרים. מלבד ניתוב של הודעות אל השרתים שיכולים לטפל בהן (מאחר ששרתים מקבלים גם הודעות שלא מיועדות להם), שרתים שולחים הודעות אשר מאפשרות לתחזק את המבנה. למשל, המנהיג מעביר פיקודים לשאר השרתים ברסיס על מנת לתחזק את שכפול המידע.

כאשר עסקה מעודכנת לזיכרון של שרת, נשלח פיקוד לשרתים של הארנקים אשר **מקבלים** את את התשלומים (לעיתים מקבל התשלום והמשלם יהיו משויכים לרסיסים שונים) שעליהם להוסיף העברה לא מבוזבזת (UTxO) אל הזיכרון שלהם.

לאחר מכן, מקבל התשלום יוכל להשתמש בהעברה זו על מנת להעביר מטבעות לאדם אחר. הסבר מפורט יותר בנוגע למימוש התשלומים ימצא בהמשך המסמך.

**ZooKeeper**

בפעם הראשונה שעל שרת ברסיס למסור הודעה למנהיג, אנחנו משתמשים ב-ZooKeeper על מנת לכתוב צומת ב"תיקייה" אשר תואמת לרסיס בו נמצא הצומת (תחת leaders/{shard}). הצומת רושם בצומת את הכתובת בה משתמשים על מנת להתחבר אליו, ולאחר מכן קורא את שאר הצמתים המאוכסנים בתיקייה. כל השרתים שברסיס מבצעים את התהליך הנ"ל בסופו של דבר, ומאחר שהצמתים בתיקייה מיוצרים פעם אחת בכל פעם שהמערכת עולה וכמו כן קיים סדר בינהם, אנחנו יכולים להשתמש בסדר של הצמתים הנ"ל על מנת לקבוע את סדר המנהיגים.

כאשר מנהיג מפסיק לענות על הודעות, אנחנו מפסיקים לשלוח לו הודעות ועוברים לצומת הבא שאחריו בסידור.

שרת אשר קורא את התיקייה לא מחכה ששאר השרתים יכתבו את הצומת שלהם – מאחר שלפני הקריאה הוא כותב את עצמו, בתיקייה יש לפחות צומת אחד. אולם, יכול להיות שלאחר שמנהיג מוסר מרשימת המנהיגים לאחר נפילה, לא ידוע מי המנהיג הבא. במקרה הנ"ל השרת יקרא את התיקייה שנית על מנת לטעון שרתים שרשמו את עצמם מאז האתחול או הנפילה הקודמת.

לכל הפחות, בתיקייה יש שרת פעיל אחד שיכול להיות המנהיג (זה השרת עצמו אשר יצר את הבקשה).

אנחנו מניחים כי כל השרתים עלו לפני שמבוצעת הבקשה הראשונה, ולכן לא נצטרך לשמור בקשות עבור שרתים שלא מגיבים למקרה שהם יעלו בשלב מאוחר יותר.

כמו כן, שרת שנפל לא עולה בחזרה ולכן אנחנו לא שומרים על קשר עם שרתים שהפסיקו לענות.

מלבד קביעת המנהיג, אנחנו משתמשים ב-ZooKeeper גם על מנת לייצר את המזהים של העסקאות. קיימת תיקייה (תחת transactions/) שבה אנחנו רושמים צומת סדרתי (מקבל מזהה מספרי) וזמני (לא נשמר לאחר שהמערכת יורדת) בכל פעם שמבוצעת עסקה. ZooKeeper מעניק לצומת מזהה מספרי אשר מגדיר סדר דטרמיניסטי, ואנחנו משתמשים במזהה הנ"ל כמזהה לעסקה.

יתרון של הגישה הזו היא שמזהה העסקה משמש גם כחותמת זמן, מה שמאפשר לנו למיין את העסקאות בסדר סיבתי/הגיוני בצורה נוחה.

**REST**

למרבה הצער לא הספקנו לממש את התקשורת בין משתמש הקצה (הלקוח) לבין השרתים באמצעות REST – כרגע, התקשורת מתבצעת בין הלקוחות לשרתים מתבצעת באמצעות gRPC, בדיוק כמו התקשורת בין השרתים עצמם.

אולם, על פי התכנון הלקוח היה פונה אל השרתים באמצעות בקשות HTTP, הטיפוס של הבקשה היה מכתיב את אופי הפעולה, והארגומנטים היו מועברים באמצעות ה-URI.

למשל, מה שכרגע ממומש באמצעות מחלקת "לקוח" אשר מתחברת אל השרתים באמצעות gRPC, מציגה מתודות כגון getTransaction ומתרגמת אותם לקריאת פעולה מרוחקת – היה ממומש באמצעות port נוסף אשר פתוח על השרת, ובהינתן בקשת HTTP מהסוג GET עם ה-URI “transaction/{id}” לדוגמה, היה מבצע את הפעולה המתאימה על השרת.

השימוש ב-REST היה חוסך את הצורך בכתיבת מחלקת הלקוח בשבילנו, והיה מאפשר למערכת לעבוד בצורה הרבה יותר גנרית, בצורה נוחה עם מערכות שכבר קיימות כיום באינטרנט.

היתרונות בשימוש ב-REST ברורים, אחת מהבעיות שנתקלנו בהם בפרויקט היא שלא הצלחנו לממש זאת.

**Docker**

יצרנו תמונה של קוד ה-Java אשר בו מומש השרת: ראשית היה עלינו ליצא את מחלקת השרת בתור קובץ jar, לאחר מכן יצרנו תמונת Docker אשר מריצה את קובץ ה-jar שכללנו באמצעות Dockerfile. את המערכת הגדרנו בתוך קובץ yml – סך הכל היא כוללת את 3 שרתי ה-ZooKeeper ועוד 8 שרתים שמריצים את התמונה שיצרנו.

באמצעות עריכת קובץ ה-yml אנחנו יכולים להיות גמישים ביותר – בקלות אפשר להוסיף עוד מופעים של התמונה שכתבנו על מנת לטפל בקצב גדול יותר של בקשות.

Docker גם דואג לאזן את העומס על השרתים, כך שאם לדוגמה הרבה מהלקוחות יהיו מרוכזים במספר קטן של רסיסים, השרתים שבאותם הרסיסים יזכו בחלק גדול יותר ממשאבי המערכת.

רעיון מעניין להרחבת הפרויקט היא הוספה דינמית של שרתים לרסיס בהינתן כמות גדולה יותר של בקשות אליו.

**מימוש**

**zookeeper**

יצרנו מספר מחלקות מעטפת אשר מקלות על השימוש ב-zookeeper. המחלקה Connection מאפשרת להתחבר ולהתנתק משרתי ה-zookeeper, בעוד המחלקה Manager אשר כוללת מימוש ו-interface מנהלת מופע קיים של ZooKeeper אשר קיים כשדה שלה.

היא חושפת מתודות אשר עמם אפשר ליצור צמתים, להחזיר את הילדים של צומת מסוים, לבדוק האם צומת קיים ולעדכן צומת.

**ledger**

שאר המימוש נמצא תחת התיקייה ledger, אשר כוללת מספר תת-תיקיות אשר בתקווה יעזרו למצוא את החלקים השונים של המימוש בקלות.

**LedgerController** נמצא תחת התיקייה controller ומממש קליינט שעמו ניתן לתקשר עם שרת בהינתן הכתובת שלו. את המימוש של המחלקה הזו היינו רוצים להחליף ב-RESTful API אשר היה רץ על השרת (הקליינט הנ"ל רץ על המכונה של הלקוח) ומטפל בבקשות אשר היו מיוצרות בצורה גנרית יותר על ידי הלקוח.

**Transaction, Transfer, UTxO** הם טיפוסי נתונים בסיסיים (POD), אין הרבה להוסיף עליהם. אליו המערכת הייתה ממומשת עם REST, במקום להחזיק רשימות Transaction הייתה שומרת מיפוי OneToMany אל Transfer ו-UTxO, וכמו כן שני המחלקות הנ"ל היו משתמשות במזהה אשר היה מיוצר בצורה אוטומטית (הוא לא מעניין אותנו בכל מקרה).

שינוי מעניין אחד, הוא ש-UTxO שומר את מספר המטבעות ששווים לערך שלו. הלקוח לא חשוף לבחירה הזו ולא מקבל את הערך הנ"ל כאשר הוא מקבל בחזרה UTxOs, אבל זה מקל עלינו כאשר אנו