**מגישים:** דביר דוד ביטון

יואב פרץ

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

**מערכות מבוזרות**

**תרגיל בית 2**

**CLIENT**

**SHARDS**

**LEADER**

**FOLLOWER**

**FOLLOWER**

**FOLLOWER**

**הערה**

חלק מהתיאורים שרשומים כאן הם מה **שניסינו** לממש. למרבה הצער, במהלך התרגיל נתקלנו בבעיות מרובות עם קוטלין וג'אווה – לא הצלחנו להריץ את השלד שקיבלנו לתרגיל ולאחר הרבה מאמצים החלטנו להתחיל את התרגיל מחדש ב-Java.

<https://github.com/dbiton/DistributedProject>

<https://github.com/dbiton/DistributedLedger>

הקישור הראשון הוא השלד של הפרויקט שהעלנו לגיטהאב, הקישור השני הוא ההתחלה של פיתרון התרגיל מהתחלה ב-Java.

בסופו של דבר הצלחנו להריץ את התרגיל אחרי ששילבנו את הפתרון ב-Java בתוך השלד שהיה נתון ב-Kotlin ואחרי המון מאמץ הצלחנו לגרום לזה לרוץ על המחשב שלי (דביר).

אנחנו מציינים זאת מאחר שלמראית עין, נראה שהעבודה בפרויקט שקישרנו בגיטהאב התחילה רק לפני מספר ימים, אולם למעשה במהלך תקופת הזמן שהיה לעבוד עם התרגיל קיימנו מספר גדול של פגישות זום לבד (דביר ויואב) וביחד עם סטודנטים אחרים בקורס (זוג אחד שלא הצליח לסיים את הפרויקט ושני זוגות אחרים שהגיעו לפתרון חלקי). כמובן שגם בנפרד כל אחד מאיתנו ניסה לפתור את הבעיות שנתקלנו בהם.

במצטבר, אני (דביר) התעסקתי עם הגדרות בערך 20 שעות (כולל פגישות עם זוגות אחרים, פגישות שלי ושל יואב ביחד ועוד זמן שניסיתי לבד) ויואב לפחות 30 שעות (מלבד הפגישות שהוא היה בהם איתי, הוא גם ניסה די הרבה להפעיל את זה על המחשב שלו).

אני רוצה לציין שבשעות האלו לא נספר עבודה על התרגיל אלא **רק ניסיון לגרום לחבילות השונות שיש בתרגיל להגיע למצב שהם מתקמפלים**. מלבד הזמן הזה, גם עבדנו על התרגיל עצמו. אני לא סופר בשעות האלו שעות שיואב השקיע בלכתוב קוד בלי שום IDE (מאחר שהוא לא הצליח לגרום לו לפעול ומועד ההגשה התקרב), הקוד הזה בסופו של דבר "הלך לפח". עד עכשיו, הקוד עדיין לא פועל במחשב של יואב.

במקרה וצריך, אני יכול להוכיח שאנחנו לא מגזימים עם כמות השעות. המספר האמיתי כנראה באזור פי 1.5 מהזמן הזה, אבל כללנו רק שעות שאני יכול "להוכיח" בצורה כלשהי שעבדנו בהם על לגרום לקוד לעבוד באמצעות שיחות זום, הודעות וואטסאפ וכו.

אני גם לא סופר בזמן הזה שעות רבות שבהם כתבתי קוד רק בשביל להתחיל מחדש בגלל שהקוד לא מצליח לרוץ (יש פתרון שאני התחלתי לכתוב בקוטלין גם ללא IDE שהלך גם כן לפח כי בסוף פתרנו ב-Java).

בשלב מסוים כשהבנתי שהסיכוי שנצליח לסיים את התרגיל בזמן קלוש ביקשתי במייל לפתור את התרגיל בפייתון או ב-CPP, אבל זה כבר היה מאוחר מדי ולא קיבלנו אישור.

מספר הבעיות שנתקלנו בהם בתרגיל הזה באמת אסטרונומי.

כרגע, נראה שמחלקה כלשהי בשם slf4j נכללית פעמיים בפרויקט (לפי מה שאני מבין) ולא מאפשרת להריץ את SpringBoot בלי לקרוס מיידית. זאת הבעיה שגרמה לנו לעבור ל-Java מ-Kotlin, חוץ מכך שאנחנו לא מכירים את Kotlin.

מימשנו קליינט שמדבר עם השרתים באמצעות gRPC בשביל לבדוק שהכל פועל, אבל צפיתי שנצליח לתקן את הבעיה עם SpringBoot. יש את כל הקוד בשביל תקשורת באמצעות REST עם השרת בפרויקט שלנו ב-GitHub (או לפחות גרסה בסיסית שלה מאחר שאני לא יכול לבדוק שזה באמת רץ), קיוויתי שאני אצליח לפתור את הבעיה עם REST לפני מועד ההגשה אבל נראה שלא נספיק.

הגרסה שנציג מולכם תהיה האחת שבה התקשורת בין הלקוחות והשרתים היא באמצעות gRPC, הכל בה ממומש ובדקתי שהוא פועל. בגיטהאב זו הגרסה לפני כל ה-Commits שנוגעים ל-REST, כתוב בה "Word document for distributed".

בכל מקרה, אנחנו מודים על הארכות הזמן שקיבלנו ועל שעות הקבלה הרבות שהיו – בלעדיהם לא היינו מגישים כלום.

התרגיל היה מאוד מעניין אבל זה מצער שביזבזנו כל כך הרבה זמן על התפל במקום על העיקר.

בקצרה – הבונוסים שממישנו הם:

1. רשימות אטומיות
2. כל היסטורית העסקאות
3. REST (כשזה מקבל בקשה זה קורס מיידית בגלל שיש שני bindings, אבל אני חושב שהמימוש עצמו נכון)
4. Docker (גם כאן עולות בעיות, אבל נראה ש"כל החלקים שם")

**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**

כאשר המערכת עולה, כל אחד מהשרתים רושם את הכתובת דרכה אפשר ליצור אליו חיבור (כלומר, host ו-port) תחת התיקייה /servers, בתור צומת זמני וסדרתי (כלומר, כאשר המערכת תעלה מחדש הצומת הנ"ל לא ימצא וכמו כן כל אחד מהצמתים בתיקייה מקבל חותמת זמן ייחודית).

לאחר מכן, השרת מחכה מספר שניות על מנת לאפשר לשאר השרתים לרשום את הכתובת שלהם גם כן. על פי סדר חותמות הזמן מוחלט לאיזה רסיס כל שרת שייך, וכמו כן השרתים יכולים ליצור את חיבורי ה-RPC בינהם באמצעות הכתובות שרשומות בתיקייה.

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.

אנחנו מודעים לכך שהיה אפשר להשתמש בסדר שנמצא ב-/servers בשביל להחליט על סדר המנהיגים, אבל בחרנו לממש זאת ככה.

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 וזמני. 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**

לכל שרת יש port אחד עבור gRPC ואחד עבור REST. כל הבקשות שמגיעות מהלקוח מתקבלות כבקשות מעל HTTP, הן מפורסרות ונשלחות ל-LedgerServer שמטפל בהן ומחזיר את התשובה. LedgerServer מבצע את התקשורת עם השרתים האחרים באמצעות gRPC אם נדרש לעשות זאת, או מחזיר את התשובה מהשדה repository שלו בו נשמרות העסקאות.

התשובה נשלחת בחזרה אל הלקוח גם כן מעל HTTP, בתור מסמך Json.

כמו כן, אנחנו מעבירים לכל שרת פרטים כגון מספר הרסיסים במערכת, הכתובת של zookeeper ועוד באמצעות המתודה setup שגם כן חלק מה-REST API שלו. כך למעשה אפשר להתחיל מחדש את השרתים בזמן ריצה בלי לעצור את ריצתם (קריאה ל-setup למעשה מבצעת reset לשרת ומאפשרת להגדיר פרמטרים חדשים עבורו).

**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. גם מימשנו מתודות עבור leaderElection ו-discovery.

**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 החלפנו ב-RESTful API אשר מטפל בבקשות שמיוצרות בצורה גנרית יותר על ידי הלקוח, אולם כרגע זה לא פועל. הקובץ הזה הוא חלק מהגרסה המוקדמת יותר של הפתרון שלנו שבה השתמשנו ב-gRPC בשביל לתקשר עם השרתים (ההיסטוריה קיימת ב-GitHub).

בתיקייה **rest** אשר נמצאת בגרסה האחרונה של הפרויקט ממומש ה-RESTful API של השרתים. יש Controller אשר מגדיר כיצד המשתמש מתקשר עם השרת, ו-RestServer שהוא SpringBootApplication גנרי.

**Transaction, Transfer, UTxO** הם טיפוסי נתונים בסיסיים (POD), אין הרבה להוסיף עליהם.

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 אנחנו מבצעים את sendCoins. במתודה הנ"ל אנו משתמשים במתודה – ConsumeUTxO אשר מחזירה רשימה של UTxO כך שסכומם הוא לפחות סכום הכסף שיש להעביר בתשלום. אנחנו מוציאים את ה-UTxO הנ"ל מה-DB שלנו בשביל למנוע תשלומים כפולים. כאשר אנחנו מחזירים את הרשימה הנ"ל, לעיתים האדם אשר משלם צריך לשלם לעצמו את העודף שנשאר מהרשימה, אם סכום ה-UTxO שהוצאו גדול מהסכום שהוא מעביר.

עצם זה שאנחנו שומרים את סכום המטבעות ש-UTxO שווה חוסך לנו תשאול של רסיסים אחרים בנוגע לערך ה-Transfer אשר השרה את ה-UTxO כאשר מדובר ברסיס שונה מהאחד של המשלם.

בתיקייה **util** מימשנו פונקציות שמתרגמות בין הודעות rpc לטיפוסי ה-POD שהגדרנו – למשל, המרה מ-rpcTransaction ל-Transaction.

ב-**LedgerRepository** מימשנו את מסד הנתונים אשר מאכסן את ה-UTxOs והעסקאות שבוצעו.

בגרסה שבה השתמשנו ב-RPC על מנת לתקשר עם השרתים, LedgerService כלל את כל הלוגיקה של המערכת (העברת הודעות וכו). בגרסה העדכנית יותר, LedgerService רק מעביר הודעות RPC, בעוד הלוגיקה של ניהול המערכת נמצאת ברובה ב-LedgerServer. המחלקה LedgerServiceClient מהווה מעין Socket בשביל לבצע תקשורת עם שרתים אחרים מעל RPC. היא כוללת stub לתקשורת עם השרת בנוסף לפרטים על השרת כמו הרסיס אליו הוא שייך.

במימוש הישן, מחלקה בשם **LedgerController** מממשת את התקשורת עם השרת (במימוש החדש אפשר להשתמש ב-curl במקום). חלק מעניין במימוש הוא שעבור רשימת עסקאות אטומית מימשנו stream אל השרת ב-gRPC, ויצרנו מחלקה שטיפלה שירשה מ-Stream ומימשה מחדש את הפונקציות next, onCompleted ו-onError.

במימוש הישן **ClientMain, ServerClient** הם מחלקות שאפשר ליצור מהם קובץ jar וממנו ליצור תמונה עבור docker. בהינתן פרמטרים שונים ניתן לקנפג את הקליינט והשרת.

המחלקה **Test** שימשה אותנו על מנת לבצע בדיקות שונות ולוודא שהקוד פועל כצפוי.