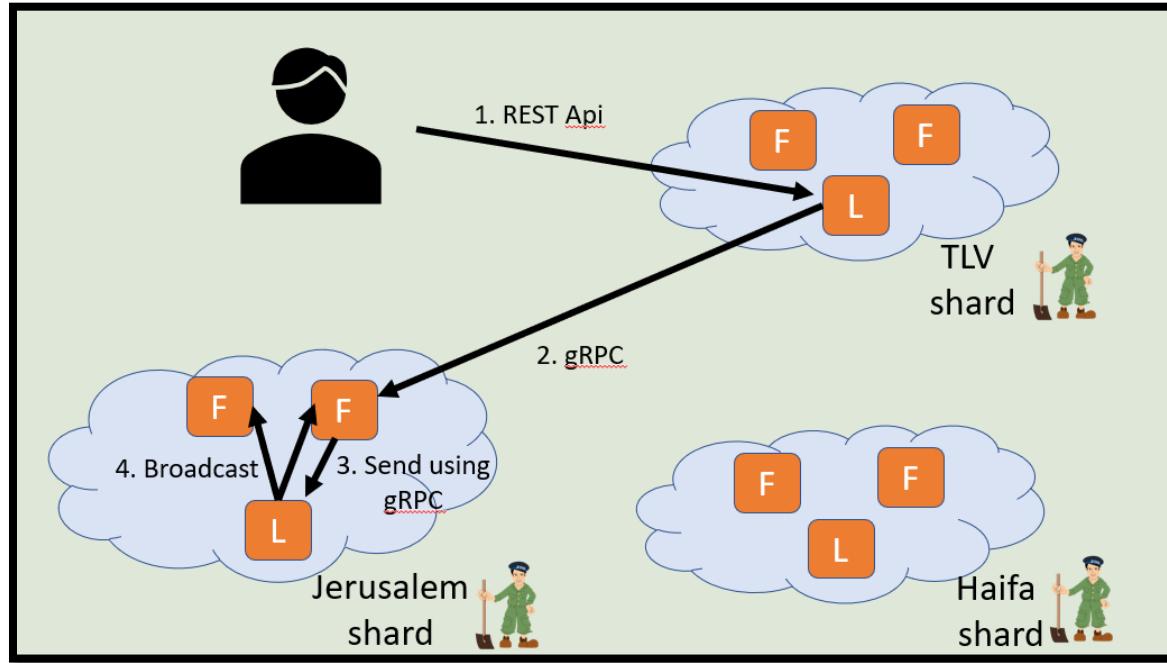


תרגיל בית מבוזרות

**תומר ורונוב 315155465
עידן רז 209276013**

Design:



הסבר ארכיטקטורה:

כל עיר מורכבת מshard של כמה שרתים וכל shard פועל בארכיטקטורת Leader-Followers REST; כלומר קיים בה leader יחיד והרבה followers. leader של כל עיר מספקת בנוסף שירות REST שכל הלוקוחות יכולים לפנות אליו.

כאשר לקוח רוצה לשלוח בקשה הוא ייצור קשר באמצעות REST API עם אחד הleaders של אחת הערים במערכת. נניח שהוא פונה (במקרה המסווג) לleader של shard של העיר תל אביב אף הוא מבקש לנסוע ליצאת מהעיר ירושלים לחיפה. השירות שקיבל את הבקשתו ינתב את הבקשת לשרת כלשונו בshard של ירושלים באמצעות gRPC. נניח שהשרת התל אביבי מנתב את הבקשת (במקרה המסווג) לשרת X שהוא follower בירושלים, אז השירות זהה ישלח את הבקשת לשרת הירושלמי שהוא יטפל בבקשתו ואז יעשה broadcast לכל שאר השירותים בshard של ירושלים. לבסוף, leader הירושלמי מחזיר את התשובה ללקוח דרך מסלול ההגעה: דרך follower X, לשרת leader בתל אביב ועד הלוקוח (או שהוא יחזיר את התשובה בעצמו במידה והוא היה השירות REST המקורי).

Implementation justification:

- ראשית, נתון לנו כי מספר לא ידוע של שירותים מכל shrad יכול ליפול. על מנת להבטיח fault tolerance מksamימי בחרנו לנוהל כל עיר במודל leader followers ולא במודלים אחרים שראינו בקורס. במודל leader follower המרבית יכולה להמשיך לתפקיד גם כאשר קיימ שרת אחד במערכת לכל shard ולכן הוא מתאים להתמודד עם נפילה של מספר לא ידוע של שירותים (לעומת מודלים אחרים מבוססי sumoquם בהם צריך שישארו sumoquם של שירותים בחים על מנת שהמערכת תוכל לתפקיד). בנוסף, מכיוון שהfollowers תמיד מעודכנים במצב של leader ההטאוששות מנפילה היא מהירה.
- בחרנו שככל leader יהיה אחראי גם על שירות REST על מנת להשתמש בו fault tolerance שקיים לנו בכל shard על מנת להבטיח fault tolerance גם לשירותי REST בלי השקעה של משאבים נוספים עבור שירותים ייעודיים למטרה זו. בנוסף, כאשר leader אחראי על שירות REST, במקרה ולקוח פונה אליו עם בקשה שנוגעת לעיר שלו הוא יכול לטפל בה בקלות ובלי תקשורת נוספת. שירות REST של כל עיר נמצאים תמיד באותו כתובות لكن הלקוחות יכולים לפנות אליהם בצורה נוחה. כפי שראינו בהרצאה ניתן ממש לבנייה כזה במציאות ע"י שימוש servers name או בקן virtual שאליו הלקוחות יכולים לפנות.

הסבר סוגי בקשות נתמכות:

האתר תומך בשלושה סוגים בקשות: askRide, offerRide וsnapshot.

:OfferRide

בofferRide המשתמש מספק `json` שמכיל את כל השדות הרלוונטיים עבור הצעת נסעה חדשה. הבקשה מנוטבת כי שהוסבר לעלה ומגיעה בסופו של דבר לשרת `leader` של העיר שמנוהה המשתמש מציע יציאה, כל שרת של עיר מחייב 2 מבני נתונים רלוונטיים:

- מפה (Map של Java) של `ridesVacancies` שבה המפתח הוא התאריך של הטרםפ, והערך הוא מפה של `RideOfferObject: RideCurrVacanciesInt`.
- מפה של `rides` שבה המפתח הוא התאריך של הטרםפ והערך הוא רשימה של `rideOffers` בתאריך הנ"ל.

כאשר שרת `leader` מקבל הצעה חדשה הוא בודק שההצעה לא קיימת עד כה במערכת, ואם לא אז מוסיף את ההצעה למפת `rides` וمعدכן את מפת `ridesVacancies` להציג ערך חדש בתאריך של הטרםפ וועשה `broadcast` לכל השירותים בעיר (שאר השירותים בעיר) בעזרתו `Zookeeper` ולבסוף `leader` מחזיר תשובה לשרת REST הראשון דרך מסלול (במידת הצורך).

:Implementation justification

- בזכות מגנון ההפצה של `followers` אנחנו משאים את כל השירותים בעיר מסווגרנים בבקשת המגיעות ועל ידי כך אנחנו דואגים ל-fault tolerance (במידה ונctrar להחליף `leader` אך `leader` החדש יהיה מעודכן בבקשת) והואנו דואגים ל-linearizability (מכיוון שלכל שירות בעיר יהיה state אחד).
- התשובה ללקוח נעשית רק לאחר הוספת `rideOffer` החדש לבני הנתונים והפצה שלו לשאר השירותים. פועלה זו מבטיחה לנו אוטומיות במובן שהבקשה או שתתקבל או שתידחה (ואנו לא נחזיר ללקוח תשובה חיובית/שלילית טרם ידנו שהoffer נכנס בלי בעיות).

- אנחנו בודקים האם offer חדש כבר קיים במערכת על מנת למנוע בעיות של שידור חוזר במידה ואחד הצדדים נפל לפני שההתשובה הגיעה ולכן הتبצע שידור מחדש.

:AskRide

בAskRide משתמש מספקโซז שמכיל את המסלול המבוקש ואת תאריך הנסעה.

הבקשה מגיעה לLeader של עיר אקראי. כעת השרת זהה יחלק את הבקשה לפי הסוגמנטים בpath ויסלח כל בקשה סגמנט לשרת leader האחראי על עיר המוצא של הסוגמנט. הוא שולח את הבקשה לכל השירותים ומחייבת לתשובה. במידה וקיבל תשובה חיובית מכל השירותים הוא שולח בקשה נוספת לביצוע commit. אם אפילו אחד מהמעורטים החזיר תשובה שלילית, הוא יבצע בקשה נוספת של cancel.

כל שירות של עיר מחזיק מפה בשם rideRequests שמחזיקה מפה שהמפתח שלה הוא הPair<PotentialRideOfferObject, Timestamp>公共服务 leader שהתקבל והערך שלה הוא RideRequestObject כאשר leader מקבל את הבקשה (הראשונית) לסוגמנט, הוא עובר על כל הטרמפים האפשריים הקיימים בתאריך המიוחל (בעזרת מפה rides על מפתח rides), ולכל ride הוא בודק כמה מקומות פנויים יש בו (בעזרת מפה ridesVacancies) ואם מצא טרמן רלוונטי הוא יזוויג בין המציע לבקשתו אחרית, הוא יפנה בעזרת RPC g לכל הערים האחרות (קרי לשירותים אקראיים בכל עיר שייעבירו את הבקשה לLeader שלהם והוא יפיץ את הבקשה broadcast לכל השירותים בעיר במידה ותתקבל) וישאל האם מישחו מאותה העיר מפה יכול לספק את הבקשה. כל אחד מהמעורטים יבצע את התנאי הבא:

- אם עיר ההתחלה של מציע הטרמן (זו גם העיר של השירות) == לעיר ההתחלה של מבקש הטרמן => אז נציג את הטרמן הנ"ל אפשרי, נשרין מקום בטרמן על ידי הורדת counter של המקומות הפנויים ב1, הוספה של חיזוג עם הקטגוריה הנוכחי למפה timestamp broadcast בעזרת Zookeeper, וביצוע broadcast לכולם שרותי העיר. הקטגוריה timestamp,rideRequests הנ"ל יבוא לידי שימוש לצורך בדיקות timeout של בקשות שעבר זמן.
- אם עיר ההתחלה של מציע הטרמן (זו גם העיר של השירות) != לעיר ההתחלה של מבקש הטרמן => נחשב את המרחק בין עיר ההתחלה של מבקש הטרמן לבין המסלול של

מציע הטרםפ, ואם המרחק הנ"ל קטן/שווה למרחק ה `PermittedDeviation` של מציע הטרםפ – היד! נבצע זיוג כפי שמתואר לעיל. אחרת, נמשיך הלאה.

אם השרת יקבל בקשה `commit` של סגמנט הוא יעדכן את ה `ridesRequests` ויסמן את `timestamp` בתור `null` (=כלומר לבקשת אין `deadline` להיות timed-out). אם השרת יקבל בקשה `cancel` הוא יעשה `revert` לכל מה שעשה (ימחק את ה `rideRequest` מהאובייקטים ויעלה את ה `ridesVacancies` של הטרםפ המוצע חזרה באחד). בכל פעולה של השרת מופעלת קראיה לפונקציה שעוברת על `rideRequests` ומנקה בקשות שעברו יותר מ30 שניות מאז קבלתן.

:Implementation justification

- בחרנו להשתמש במנגנון של `commit` ו `cancel` כדי למנוע מצב לא יציב של המערכת (למשל אם חלק מהסగמנטים אישרו טרמף וחלק אחר החזירו תשובה שלילית אז לא נרצה לסמן את הסגמנטים החביבים בתור "בטוחים"). בנוסף, המנגנון הזה מודוד אוטומטיות שכן לא נחוץ ללקוח הודעה חיובית/שלילית עד שכל הסגמנטים החזירו תשובה חיובית ואז נשלח אליהם `commit` או עד שלפחות אחד מהם החזיר תשובה שלילית ואז נשלח אליהם `cancel`.

:Snapshot

כאשר `leader` כלשהו מקבל בקשה REST לקלטת `snapshot` של המערכת הוא פונה אל שרת אחד מכל `shard` בעזרת `gRPC` על מנת לקבל את תמונה המצב שלהם של המערכת. כיוון שככל השרותים באוטו `shard` מסווגרים על המצב שלהם בחרנו לא להעביר את בקשות `snapshot` ל `leader` אלא לטפל בהן בשרת שקיבל את בקשה `gRPC`. אחרי שהוא מקבל את תמונה המצב מכל השרותים במערכת השרת שקיבל את הבקשת `snapshot` מחד את כל התמונות ביחיד על מנת לקבל את `state` הכלול של המערכת (כל שרת בעיר שומר רק את הנסיעות ואת המקטעים שהוזמנו עבור העיר שלו ולכך נדרש לאחד אותם ביחיד על מנת לבנות את תמונה הנסעה הכוללת). את מצב זה הוא מחייב ללקוח בפורמט JSON הנitinן לפרוטוור בקלות.

הבטחות המערכת:

- אוטומיות: אם לקוח מבצע בקשה אז או שהוא התקבל או שהוא תדחה. לא יכול להיות מצב ביןים שבו למשל נחזיר לו תשובה שקבענו עבורו טרמף אבל לא באמת קבענו.
- אמינות: אם אנחנו מחזירים שהבקשה שלו נחתה/התקבל אז בודאות זה אכן קרה.
- שידאג תמיד לעשות broadcast לשאר שירותים העיר. בנוסף, אנחנו דואגים להוציא שירותים שאינם מגיבים heartbeat לוודא membership. באופן זה, גם אם שירות ייפול, שאר שירותים בעיר עדין יהיו מסוכנים על הבקשות.
- Linearizability: בעזרת Zookeeper, לכל שירות בעיר יהיה בדיקות state באותו סדר הגעת בקשות.
- Availability: בכל רגע נתון יהיה קיים leader לכל עיר בזכות בדיקת membership של כל השירותים בעיר. במידה וleader נופל, שירות אחר יחליף אותו ויטפל בבקשתות העיר.
- עקביות: state של המערכת שלנו הוא עקיbij ולא תלוי בזמןן שבו נשלחת הבקשה. לכן, כאשר בקשה תישלח בתזמן שונה (ולא הייתה בקשה אחרת בין לביין) אנחנו נחזיר בדיקת את אותה תשובה.
- Scalability: המערכת שלנו מתאימה להיות סקליבילית בצורה קלה ונוחה. בעת התווספות שירותי הם פשוט מודיעים לzoookeeper על הצטרכות shard וממקום והלאה הם יחוּץ מהshard ויקבלו את הodata broadcast מהleader.
- יעילות: בחרנו למשם את המערכת שלנו בצורה מבוזרת ככל הנימן ולא למשם שירות יחיד שאליו הלקוח יכול לפנות שישמש כשרת redirection (ויהו בעצם bottleneck). במקום זאת, לquo יכול לפנות לכל שירות, והוא יהיה זה שבמקרה הצורך ינתב את הבקשה לleader המתאים (ולכן למעשה ביצרנו את העומס משרת redirection ייחיד לכל leaders).

שימוש בכלים של מדן:

:Zookeeper

השתמשנו בזookeeper כדי לנהל את התקשרות בין leader לfollowers בכל עיר מכיוון שהוא מספק הבטחות מאוד חזקות בנוגע לפועלה שלו. התכונות העיקריות עליו הסתמכו (עליהם אנחנו מתבססים בכל הסעיפים הבאים) הן Casual Order, Total Order, Reliable Delivery

- בחירת leader:

בחירת leader לכל עיר נעשית באופן דומה לאלגוריתם שראינו בcliffe – לכל shard במערכת קיימצומת האחראי על הבחירה. כל הצמתים באותו shard יוצרים צומת חדש תחת צומת זה בתהליך העלייה שלהם בעל התכונות ephemeral וsequential. בכל רגע נתון leader של השרת הוא השרת אשר יצא את הצומת עם המספר הסידורי הכי נמוך. שימוש זה מקיים את התכונות של leader election כיון שכאשר המנהיג מת הצומת שהוא יצר נמחק ושרת אחר ידע שהוא המנהיג החדש (כיון שהצומת הוא ephemeral) וכי בכל רגע יהיה קיים שרת אחד בלבד אשר יהיה בעל המספר הסידורי הכי נמוך (בגלל שהצומת הוא sequential).

- בדיקת membership :

באופן דומה לleader election, לכל shard במערכת קיימצומת אשר כל השירותים של אותו shard רושמים את עצםם בעזרת nodes ephemeral כאשר הם מצטרפים למערכת ובעזרתו ניתן לדעת בכל רגע איזה שירותים חיימים בshard המדובר.

- ביצוע broadcast על עדכונים שmaguiim (כמו טרמף חדש שמוצע או בקשה של טרמף שהתקבלה):

בחרנו לבצע את broadcast בין leader לfollowers ? בזכות התכונות שהוא מבטיח לנו שהזכרנו קודם. תכונות אלו מבטיחות לנו שהתקשרות בין leader לfollowers תהיה אמינה וכי הם יהיו מעודכנים בהתאם למצבי של leader.

: Tests+Dockers

השתמשנו בdockers כדי לבדוק את המערכת שלנו. כל שרת רץ בcontainer נפרד שמקבל בשורת הרצה שלו את הארגומנטים לרכיב (מספר פורט להאזין בgRPC, שם העיר שאליה השרת שייר והכתובת של zookeeper). השתמשנו בbuildmultistage buildimage המכיל את כל התלוויות של gradle ואת hkdk והסופי שלנו הכל רק את jar של המערכת שלנו ואת REJ על מנת שהוא יהיה קל ככל הניתן. בנוסף, בחרנו להשתמש בhost networking על מנת לתקשר בין containers שלנו, בין zookeeper ובין clients שורץ במכונה שלנו. את המערכת שלנו בדקנו על ידי הרצת ל Koho (שפה פיתון) שישלח בקשות REST API לשרתים שלנו בסacle גדול וישמש למעשה system test.