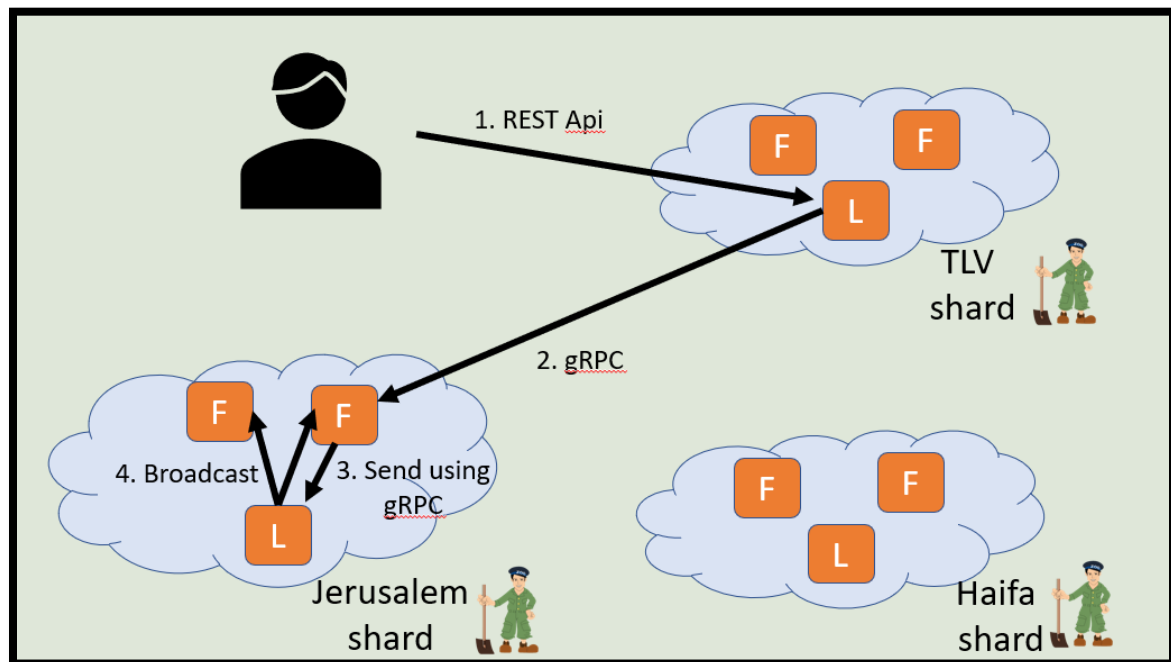


# תרגיל בית מבוזרות

**תומר וורנוב 315155465**

**עידן רז 209276013**

## Design:



### הסבר ארכיטקטורה:

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

כאשר לקוח רוצה לשלוח בקשה הוא יצור קשר בעזרת REST API עם אחד הleaders של אחת הערים במערכת. נניח שהוא פונה (במקרה המסובך) לleader של shard של העיר תל אביב אך הוא מבקש לנסוע לצאת מהעיר ירושלים לחיפה. השרת שקיבל את הבקשה ינתב את הבקשה לשרת כלשהו בshard של ירושלים בעזרת gRPC. נניח שהשרת התל אביבי מנתב את הבקשה (במקרה המסובך) לשרת X שהוא follower בירושלים, אז השרת הזה ישלח את הבקשה לשרת הleader הירושלמי שהוא יטפל בבקשה ואז יעשה broadcast לכל שאר השרתים בshard של ירושלים. לבסוף, הleader הירושלמי מחזיר את התשובה ללקוח דרך מסלול ההגעה: דרך followers X, לשרת הleader בתל אביב ועד הלקוח (או שהוא יחזיר את התשובה בעצמו במידה והוא היה שרת הREST ההתחלתי).

## Implementation justification:

- ראשית, נתון לנו כי מספר לא ידוע של שרתים מכל shard יכול ליפול. על מנת להבטיח fault tolerance מקסימלי בחרנו לנהל כל עיר במודל leader followers ולא במודלים אחרים שראינו בקורס. במודל הleader followers המערכת יכולה להמשיך לתפקד גם כאשר קיים שרת אחד במערכת לכל shard ולכן הוא מתאים להתמודד עם נפילה של מספר לא ידוע של שרתים (לעומת מודלים אחרים מבוססי quorums בהם צריך שישארו quorum של שרתים בחיים על מנת שהמערכת תוכל לתפקד). בנוסף, מכיוון שהfollowers תמיד מעודכנים במצב של הleader ההתאוששות מנפילה היא מהירה.
- בחרנו שכל leader יהיה אחראי גם על שירות הREST על מנת להשתמש בfault tolerance שקיים לנו בכל shard על מנת להבטיח fault tolerance גם לשירותי ה REST בלי השקעה של משאבים נוספים עבור שרתים ייעודיים למטרה זו. בנוסף, כאשר הleader אחראי על שרת הREST, במקרה ולקוח פונה אליו עם בקשה שנוגעת לעיר שלו הוא יכול לטפל בה בקלות ובלי תקשורת נוספת מיותרת. שרתי הREST של כל עיר נמצאים תמיד באותה כתובת לכן הלקוחות יכולות לפנות אליהם בצורה נוחה. כפי שראינו בהרצאה ניתן לממש מבנה כזה במציאות ע"י שימוש בname servers או בvirtual ip שאליו הלקוחות יכולים לפנות.

## הסבר סוגי בקשות נתמכות:

האתר תומך בשלושה סוגי בקשות: offerRide, snapshot askRideI.

### **:OfferRide**

בofferRide המשתמש מספק json שמכיל את כל השדות הרלוונטים עבור הצעת נסיעה חדשה. הבקשה מנותבת כפי שהוסבר למעלה ומגיעה בסופו של דבר לשרת הleader של העיר שממנה המשתמש מציע לצאת. כל שרת של עיר מחזיק 2 מבני נתונים רלוונטים:

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

כאשר שרת הleader מקבל הצעה חדשה הוא בודק שהoffer לא קיים עד כה במערכת, ואם לא אז מוסיף את ההצעה למפת rides ומעדכן את מפת ridesVacancies להחזיק ערך חדש בתאריך של הטרמפ ועושה broadcast לכל הfollowers (שאר השרתים בעיר) בעזרת Zookeeper ולבסוף הleader מחזיר תשובה לשרת הREST הראשון דרך אותו מסלול (במידת הצורך).

### **:Implementation justification**

- בזכות מנגנון ההפצה של הleader לfollowers אנחנו משאירים את כל השרתים בעיר מסונכרנים בבקשות המגיעות ועל ידי כך אנחנו דואגים לfault tolerance (במידה ונצטרך להחליף leader אז הleader החדש יהיה מעודכן בבקשות) ואנחנו דואגים לLinearizability (מכיוון שלכל שרתי העיר יהיה state אחיד).
- התשובה ללקוח נעשית רק לאחר הוספת הrideOffer החדש למבנה הנתונים וההפצה שלו לשאר השרתים. פעולה זו מבטיחה לנו אטומיות במובן שהבקשה או שתתקבל או שתידחה (ואנחנו לא נחזיר ללקוח תשובה חיובית/שלילית טרם וידאנו שהoffer נכנס בלי בעיות).

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

## :AskRide

בaskRide המשתמש מספק json שמכיל את המסלול המבוקש ואת תאריך הנסיעה.

הבקשה מגיעה לleader של עיר אקראית. כעת השרת הזה יחלק את הבקשה לפי הסגמנטים בpath וישלח כל בקשת סגמנט לשרת הleader האחראי על עיר המוצא של הסגמנט. הוא שולח את הבקשה לכל השרתים ומחכה לתשובה. במידה וקיבל תשובה חיובית מכל השרתים הוא שולח בקשה נוספת לביצוע commit. אם אפילו אחד מהשרתים החזיר תשובה שלילית, הוא יבצע בקשה נוספת של cancel.

כל שרת של עיר מחזיק מפה בשם rideRequests שמחזיקה מפה שהמפתח שלה הוא ה RideRequestObject שהתקבל והערך שלה הוא <Pair<PotentialRideOfferObject, Timestamp>. כאשר leader מקבל את הבקשה (הראשונית) לסגמנט, הוא עובר על כל הטרמפים האפשריים הקיימים בתאריך המיוחל (בעזרת מעבר על מפת rides), ולכל ride הוא בודק כמה מקומות פנויים יש בו (בעזרת מפת ridesVacancies) ואם מצא טרמפ רלוונטי- הוא יזווג בין המציע למבקש. אחרת, הוא יפנה בעזרת gRPC לכל הערים האחרות (קרי לשרתים אקראיים בכל עיר שיעבירו את הבקשה לleader שלהם והוא יפיץ את הבקשה בbroadcast לכל השרתים בעיר במידה ותתקבל) וישאל האם מישהו מאחת הערים האחרות יכול לספק את הבקשה. כל אחד מהשרתים יבצע את התנאי הבא:

- אם עיר ההתחלה של מציע הטרמפ (זו גם העיר של השרת) == לעיר ההתחלה של מבקש הטרמפ ← אז נציע את הטרמפ הנ"ל כאפשרי, נשריין מקום בטרמפ על ידי הורדת counter של המקומות הפנויים ב1, הוספה של הזיווג עם timestamp הנוכחי למפת rideRequests, וביצוע broadcast בעזרת Zookeeper לכל שאר שרתי העיר. timestamp הנ"ל יבוא לידי שימוש לצורך בדיקות timeout של בקשות שעבר זמנן.
- אם עיר ההתחלה של מציע הטרמפ (זו גם העיר של השרת) != לעיר ההתחלה של מבקש הטרמפ ← נחשב את המרחק בין עיר ההתחלה של מבקש הטרמפ לבין המסלול של

מציע הטרמפ, ואם המרחק הנ"ל קטן/שווה למרחק ה PermittedDeviation של מציע הטרמפ – הידד! נבצע זיווג כפי שמתואר למעלה. אחרת, נמשיך הלאה.

אם השרת יקבל בקשה ל commit של סגמנט הוא יעדכן את ה rideRequests ויסמן את ה timestamp בתור null (=כלומר לבקשה אין deadline להיות timed-out). אם השרת יקבל בקשה ל cancel הוא יעשה revert לכל מה שעשה (ימחק את ה rideRequest מהאובייקטים ויעלה את ה ridesVacancies של הטרמפ המוצע חזרה באחד). בכל פעולה של השרת מופעלת קריאה לפונקציה שעוברת על rideRequests ומנקה בקשות שעברו יותר מ 30 שניות מאז קבלתן.

### Implementation justification:

- בחרנו להשתמש במנגנון של cancel commit כדי למנוע מצב לא יציב של המערכת (למשל אם חלק מהסגמנטים אישרו טרמפ וחלק אחר החזירו תשובה שלילית אז לא נרצה לסמן את הסגמנטים החיוביים בתור "בטוחים"). בנוסף, המנגנון הזה מוודא אטומיות שכן לא נחזיר 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 בקשת REST לקבלת snapshot של המערכת הוא פונה אל שרת אחד מכל shard בעזרת gRPC על מנת לקבל את תמונת המצב שלהם של המערכת. כיוון שכל השרתים באותו shard מסונכרנים על המצב שלהם בחרנו לא להעביר את בקשות ה snapshot ל leader אלא לטפל בהן בשרת שקיבל את בקשת ה gRPC. אחרי שהוא מקבל את תמונת המצב מכל השרתים במערכת השרת שקיבל את הבקשה מאחד את כל התמונות ביחד על מנת לקבל את ה staten הכולל של המערכת (כל שרת בכל עיר שומר רק את הנסיעות ואת המקטעים שהוזמנו עבור העיר שלו ולכן צריך לאחד אותם ביחד על מנת לבנות את תמונת הנסיעה הכוללת). את מצב זה הוא מחזיר ללקוח בפורמט JSON הניתן לפרסור בקלות.

## הבטחות המערכת:

- אטומיות: אם לקוח מבצע בקשה אז או שהיא תתקבל או שהיא תדחה. לא יכול להיות מצב ביניים שבו למשל נחזיר לו תשובה שקבענו עבורו טרמפ אבל לא באמת קבענו.
- אמינות: אם אנחנו מחזירים שהבקשה שלו נדחתה/התקבלה אז בודאות זה אכן קרה.
- Fault tolerance and Reliability: בעזרת Zookeeper אנחנו מקפידים לבחור leader חי שידאג תמיד לעשות broadcast לשאר שרתי העיר. בנוסף, אנחנו דואגים להוציא שרתים שאינם מגיבים לheartbeat שדואג לוודא membership. באופן הזה, גם אם שרת יפול, שאר שרתי העיר עדיין יהיו מסונכרנים על הבקשות.
- Linearizability: בעזרת Zookeeper, לכל שרתי העיר יהיה בדיוק אותו state באותו סדר הגעת בקשות.
- Availability: בכל רגע נתון יהיה קיים leader לכל עיר בזכות בדיקת membership של כל השרתים בעיר. במידה leader נופל, שרת אחר יחליף אותו ויטפל בבקשות העיר.
- עקביות: הstate של המערכת שלנו הוא עקבי ולא תלוי בתזמון שבו נשלחת הבקשה. לכן, כאשר בקשה תישלח בתזמון שונה (ולא הייתה בקשה אחרת בין לבין) אנחנו נחזיר בדיוק את אותה תשובה.
- Scalability: המערכת שלנו מתאימה להיות סקלבילית בצורה קלה ונוחה. בעת התווספות שרתים הם פשוט מודיעים לzookeeper על הצטרפותם לshard ומכאן והלאה הם יהוו חלק מהshard ויקבלו את הודעות הbroadcast מהleader.
- יעילות: בחרנו לממש את המערכת שלנו בצורה מבוזרת ככל הניתן ולא לממש שרת יחיד שאליו הלקוח יכול לפנות שישמש כשרת redirection (ויהווה בעצם bottleneck). במקום זאת, לקוח יכול לפנות לכל שרת, והוא יהיה זה שבמקרה הצורך ינתב את הבקשה לleader המתאים (ולכן למעשה ביזרנו את העומס משרת redirection יחיד לכל הleaders).

## שימוש בכלים שלמדנו:

### **Zookeeper:**

השתמשנו בZookeeper כדי לנהל את התקשורת בין leader לfollowers בכל עיר מכיוון שהוא מספק הבטחות מאוד חזקות בנוגע לאופן הפעולה שלו. התכונות העיקריות עליהן הסתמנו (ועליהן אנחנו מתבססים בכל הסעיפים הבאים) הן Casual Order, Total Order, Reliable Delivery :

- בחירת leader:

בחירת leader לכל עיר נעשית באופן דומה לאלגוריתם שראינו בכיתה – לכל shard במערכת קיים צומת האחראי על הבחירות. כל הצמתים באותו shard יוצרים צומת חדש תחת צומת זה בתהליך העלייה שלהם בעל התכונות sequential וephemeral. בכל רגע נתון הleader של השרת הוא השרת אשר יצא את הצומת עם המספר הסידורי הכי נמוך. מימוש זה מקיים את התכונות של leader election כיוון שכאשר המנהיג מת הצומת שהוא יצר נמחק ושרת אחר ידע שהוא המנהיג החדש (כיוון 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 קיים צומת אשר כל השרתים של אותו shard רושמים את עצמם בעזרת ephemeral nodes כאשר הם מצטרפים למערכת ובעזרתו ניתן לדעת בכל רגע איזה שרתים חיים בshard המדובר.

- ביצוע broadcast על עדכונים שמגיעים (כמו טרמפ חדש שמוצע או בקשה של טרמפ שהתקבלה):

בחרנו לבצע את הbroadcast בין הleader לfollowers בזכות התכונות שהוא מבטיח לנו שהזכרנו מקודם. תכונות אלו מבטיחות לנו שהתקשורת בין הleader לfollowers תהיה אמינה וכי הם יהיו מעודכנים בנוגע למצב של leader.



## : Tests+Dockers

השתמשנו בdocker כדי לבדוק את המערכת שלנו. כל שרת רץ בcontainer נפרד שמקבל בשורת ההרצה שלו את הארגומנטים לריצה (מספר פורט להאזין בgRPC, שם העיר שאליה השרת שייך והכתובת של zookeeper). השתמשנו בmultistage build, התחלנו מimage המכיל את כל התלויות של gradle ואת jdk והimage הסופי שלנו הכיל רק את את jar של המערכת שלנו ואת JRE על מנת שהוא יהיה קל ככל הניתן. בנוסף, בחרנו להשתמש בhost networking על מנת לתקשר בין הcontainers שלנו, בין הzookeeper ובין client שרץ במכונה שלנו. את המערכת שלנו בדקנו על ידי הרצת לקוח (בשפת פייתון) שישלח בקשות REST API לשרתים שלנו בscale גדול וישמש למעשה כsystem tests.