Distributed Systems: UNO Assignment

De Lange Matthias
Sander Sienaert
KULeuven FIIW - MELICTIsw

December 19, 2017

1 Architectuur

De architectuur van onze UNO applicatie bestaat uit 4 instanties: de clients, applicatie servers, databases en dispatcher.

In ons architectuur proberen we fat-clients zoveel mogelijk te vermijden. Dit betekent dat de logica en data management vooral aan de applicatie servers wordt overgelaten. De clients gaan enkel de nodige informatie gaan opvragen. Hierbij proberen we te steunen op het principe van micro-services. Tijdens het UNO spel zelf zal de client meerdere RMI connecties simultaan gaan oproepen in zijn background threads. Dit bevordert de efficiëntie in het gebruik van het netwerk. Ook wordt in deze micro-services enkel de nodige informatie doorgestuurd. We maken dus lightweight objecten in ons model, die enkel bedoeld zijn voor de client. Als een client aan de beurt is in een UNO spel, wordt wel eerst aan de client-side gekeken of zijn move valid is. Is dit niet het geval, dan wordt ook niet onnodig data met de applicatie server uitgewisseld. Wel wordt een valid move, verzonden door de client, een tweede keer gecontroleerd op de server kwestie van valsspelerij tegen te gaan.

Daarnaast hebben we de dispatcher die initieel aan een client een bepaalde applicatie server zal toewijzen. De dispatcher gaat de verscheidene opgestarte appservers en databases gaan bijhouden en managen. Hierbij wordt er aan een simpele vorm van load balancing gedaan. Voor elke applicatie server wordt bijgehouden met hoeveel clients deze verbonden is. Zo kan de dispatcher een nieuwe client gaan redirecten naar de applicatie server die nog het meeste capaciteit over heeft. De capaciteit wordt initieel opgegeven aan de dispatcher bij opstart en wordt uitgedrukt in het aantal ondersteunde clients. Hiernaast staat de dispatcher ook in connectie met de applicatie servers om nieuwe database servers te kunnen toewijzen in het geval van connectieproblemen of crashes van databases.

De applicatie servers bevatten de logica van het UNO-spel, verzorgen de authenticatie en authorisatie, en ze beheren de lobby's en gamelobby's. Hier worden alle micro-services verwerkt die worden opgeroepen van de client uit. Het model verandert alleen als een speler een zet doet. Wij gaan dit proberen exploiten voor onze micro-services: bij een move wordt het model op de applicatie server geüpdatet, waarna data wordt verstuurd naar alle clients die nieuwe info nodig hebben. We sturen hier niet een heel Game-object door waarmee clients hun model kunnen updaten. We sturen enkel de minimieme data die nodig is voor de clients om een correcte UI te construeren. Deze UI is consistent met het spel-model op de applicatie server. De volgende micro-services worden gebruikt in het spel:

playMove Enkel de speler die aan beurt is roept deze micro-service aan. Deze notified de andere micro-services van het spel om te laten weten dat het model is geüpdatet.

fetch CurrentPlayer and Card Wanneer een move is gespeeld, wordt deze opgeroepen om de huidige speler en kaart door te sturen naar de clients.

fetch other players info Wanneer een move is gespeeld, wordt deze opgeroepen om het aantal kaarten van de andere spelers op te vragen.

fetch plus cards De client vraagt zijn kaarten maar 1 keer op in het begin van het spel. Daarna wordt enkel aan de hand van de laatste move gekeken wie er pluskaarten moet krijgen.

Als laatste component zijn er nog de databases. Deze worden in de volgende sectie uitvoerig beschreven.

2 Databank Model

2.1 Tabellen

In onze database worden de volgende tabellen bijgehouden:

User Iedere user heeft een unieke *id*, *salt*, *hash* en is gekoppeld aan een *player*.

Player Bevat de *name* van de user, zijn *highscore* en de *game_id* van het spel waarin hij zich momenteel bevindt.

Game Elk spel heeft een unieke *gameId* die de samenstelling is van de *gameName* en een timestamp. Vervolgens houdt het spel ook nog bij in welke richting het spel gaat met een boolean *clockwise*. Tot slot nog de maximale *gameSize* en het aantal *joinedPlayers*.

Move Hierin houden we de flow van een game bij. De *game_id*, *player_id* en *card_id* kunnen zo gebruikt worden om een spel te reconstrueren. Wanneer de card_id null is, wil het zeggen dat de speler een kaart getrokken heeft.

Card Ook elke kaart heeft unieke *id*. We houden ook bij tot welke *player_id* en welke *game_id* ze behoort. Uiteraard bepalen we ook over welke kaart uit het UNO-spel we het hebben m.b.v. *cardType*, *color* en *value*.

2.2 Beschrijving

De databases staan in voor het persisten en opvragen van data voor de applicatie servers. Maar ook dienen ze onderling consistent gehouden te worden.

Voor het consistentiemodel van het spel wordt er met Move-objecten gewerkt. Dit biedt de mogelijkheid om spellen te reconstrueren aan de hand van de gespeelde Moves.

De assumptie die we volgens de opdracht mochten aannemen is dat al de database servers initieel weten wie de andere database servers zijn. Wij hebben het zo geïmplementeerd dat de dispatcher deze info meegeeft bij de opstart van een database.

De databases zijn in SQLite in-memory-databases. Als abstractielagen hierbovenop gebruiken we ORMLite, dit is een object-relational-mapping framework.

3 Ontwerpbeslissingen

3.1 Consistentie en Replicatie

De client komt als eerste in connectie met de dispatcher. Deze geeft door aan de client met welke applicatie server hij kan verbinden.

Initieel wordt een Game samen met zijn Players en starter deck gepersist naar de database. Tijdens het spel zelf worden enkel de Moves nog doorgestuurd. Eerst en vooral gaat de database de data zelf opslaan als hij deze binnenkrijgt. Onmiddellijk hierna stuurt hij een acknowledgement naar de applicatie server, zodat deze verder kan. Pas hierna gaan we de update propageren naar de andere databases, dit om te voorkomen dat de applicatie server traag responsie krijgt.

Bij het propageren van deze updates wordt geprobeerd naar alle andere peer databases door te sturen. Als er geen connectie is met een database bvb. door een crash, dan worden de updates bewaard in een queue voor die database. Als er later wel connectie zou zijn, worden deze updates eerst doorgestuurd. Dit wordt tot stand gebracht doordat een database bij opstart al zijn updates aan zijn peer databases gaat opvragen. De update-queue's worden geleegd bij het doorsturen van de updates. Op deze manier houden we ons databases consistent.

3.2 Caching

De afbeeldingen van onze Card-objecten worden niet telkens naar onze client heen en weer gestuurd zodat veel nutteloze transmissie kan vermeden worden. Zeker vanuit gebruikersperspectief is het niet aangenaam als elke zet enige merkbare verwerkingstijd nodig heeft. Daarom zullen we de afbeeldingen van de kaarten bij de client gaan cachen. Wanneer de client zich succesvol inlogt, zal er een background service opgestart worden die de kaartafbeeldingen zal binnenhalen op de applicatie server. De applicatie server gaat op zijn beurt de database server aanspreken om de

afbeeldingen op te halen. Zo kunnen we er ook eenvoudig voor zorgen dat op speciale feestdagen een andere set van kaarten zal binnengehaald worden.

Wanneer een spel gestart wordt, zal er ook steeds gecontroleerd worden of iedere speler reeds alle afbeeldingen heeft binnengehaald. Omdat we niet willen dat een speler moet spelen zonder afbeeldingen, kan er geen spel gestart worden zolang niet iedereen alle afbeeldingen heeft. Maar aangezien er tussen de login en het joinen van een spel ruim veel tijd aanwezig is, kan de backgroundthread hoogstwaarschijnlijk alle afbeeldingen binnen die tijd ophalen. Een nog betere oplossing, vanuit gebruikersstandpunt, zou zijn om een client die nog niet al zijn kaarten heeft ontvangen, de toegang tot het creëren/joinen van games te ontnemen.

De client cachet zelf ook gegevens over het spel, zoals bijvoorbeeld zijn eigen kaarten, om zo weinig mogelijk data uitwisseling met de server te voorzien.

3.3 Beveiliging

3.3.1 Authentication

Nieuwe gebruikers dienen zich te registreren met een wachtwoord dat voldoet aan minstens de volgende vereisten:

- 6 karakters lang
- 1 hoofdletter
- 1 kleine letter
- 1 van volgende karakters: @#$%
- 1 getal

Deze vereisten worden client side al gevalideerd, vooraleer de registratie door te sturen naar de applicatie server.

Vervolgens kijken we in ons database of de usernaam uniek is. Is dit het geval, dan genereert de applicatie server een salt voor deze gebruiker. Deze wordt samen met het wachtwoord gehasht. De hash en de salt worden bijgehouden in onze database. Wanneer een gebruiker zich vervolgens wil aanmelden, zullen zijn ingegeven wachtwoord en username doorgestuurd worden naar de applicatie server. Het wachtwoord zal samen met de salt uit de database opnieuw gehasht worden en zo vergeleken worden met de hash uit de database. Wanneer dit succesvol lukt, zal de gebruiken ingelogd worden en een JWT ontvangen voor zijn sessie.

3.3.2 Authorization

Voor de autorisatie maken we dus gebruik van *JSON Web Tokens (JWT)*. Wanneer de client zich succesvol inlogt, krijgt hij van de server een JWT die hij later nodig heeft om zich te gaan authoriseren op de server. Deze JWT wordt door de client bij het aanroepen van elke methode op de server meegegeven en wordt door de server gevalideerd. Zo zullen er geen derden toegang kunnen krijgen tot onze servers zonder geldige token. De geldigheidsduur van de token is 24h. Het grote voordeel van JWT's is dat er geen tokens op de applicatie server moeten worden opgeslagen. De applicatie server heeft enkel een secret nodig, waarmee hij de JWT kan signen. Het signen gebeurt bij ons via het HMAC SHA256 algoritme. Zo weet de server dat de token niet is aangepast eens hij die ontvangt.

De token zou echter wel gebruikt kunnen worden in replay attacks. Maar in de opdracht mogen we veronderstellen dat er over een veilig kannaal wordt gecommuniceerd, waardoor we hiervan abstractie kunnen maken.

3.3.3 Model

In ons model werken we met verschillende ojecten: Player en User. Het User-object wordt gebruikt voor authenticatie en authorisatie doeleinden. Hier worden de hash en salt gekoppeld aan een bepaalde Player. Deze architectuur zorgt ervoor dat Player-objecten geen confidentiele informatie bevatten idien ze aan Games toebehoren of voor doeleinden buiten security gebruikt worden.

3.3.4 SQL Injections

Op de database servers gebruiken we ORMLite. ORM's hebben hun voordelen bij security indien er gebruik gemaakt wordt van parameterized queries. Hierbij wordt eerst de query zelf geconstrueerd zonder parameters. De parameters worden pas later toegevoegd, en kunnen niet als SQL geinterpreteerd worden doordat de string-escape karakters eruit worden gefilterd.

3.4 Recovery

Onze applicatie ondersteunt het uitvallen van een database server en ondersteunt architecturaal de mogelijkheid om Games te reconstrueren door gebruik van Move-objecten.

Als een applicatie server wil persisten naar zijn toegewezen database, en deze reageert niet of is gecrashed, dan vraagt de applicatie server een nieuwe database aan de dispatcher. De dispatcher kijkt welke database het minst toegewezen applicatieservers heeft en wijst deze toe. De applicatie server kan zijn normale functionaliteit gewoon hervatten alsof er niks gebeurd is.

Elk Game object bevate een lijst met de gespeelde moves. Eeen Move-object bevat een Player en een Card. Hierdoor kunnen we de flow van het hele spelverloop bijhouden. Dit biedt mogelijkheden voor uitbereidingen zoals reconstructies van Games.

Ook na de heropstart van een database na enige downtime, gaat deze initieel al zijn gemiste updates ophalen waardoor de data gerecovered kan worden op deze database server.

4 Documentatie

Zie JavaDoc in Github repo.

5 Reflectie

Ik denk we met trots mogen terugkijken naar ons project van de voorbije maanden, waar we intensief aan gewerkt hebben. Hieronder zetten we nog eens onze sterktes in de verf. Vervolgens kijken we ook eens kritisch naar onze zwaktes en voorzien we ook ruimte voor mogelijke uitbreidingen.

5.1 Sterktes

We voorzien zo weinig mogelijk overhead om zo de gebruikerservaring te maximaliseren. Om dat te bereiken, versturen we enkel de hoogst noodzakelijke informatie, met behulp van micro-services, tussen client en applicatie server.

Voorts voorziet onze applicatie ook security die aan de industrie vereisten voldoet. Zowel op het gebied van authenticatie als autorisatie gebruiken we industriële security standaarden die reeds hun waarde in het veld bewezen hebben. Daarbovenop zijn de JWT's nog eens heel compact.

De dispatcher gaat aan load balancing doen voor nieuwe clients die applicatie servers aanvragen. Ook is er robuustheid tegen database servers die crashen. De databases zelf gaan de updates bijhouden voor de andere databases en de applicatie server vraagt een nieuwe, actieve database aan de dispatcher.

Een grote sterkte is ook onze bestendigheid tegen database crashes. Applicatie servers kunnen gewoon een nieuwe database server aan de dispatcher opvragen en doorgaan zonder dat de eindgebruiker er iets van merkt. Hiernaast worden de updates voor deze database ook bijgehouden door de andere databases. Eens de database terug up-and-running is, kan deze gewoon al zijn gemiste updates gaan opvragen aan zijn peer databases.

5.2 Zwaktes

Ons database model steunt op unieke usernames. Wegens tijdgebrek hebben we echter niet de edge-case kunnen behandelen waarbij net op hetzelfde moment twee users registreren met identieke naam. Hiervoor hebben we echter wel een oplossing uitgedacht waarbij een database eerst een ACK krijgt van alle andere databases dat hij een nieuwe user mag aanmaken. De databases die een ACK sturen locken daarbij operaties op de user-table tot ze een ACK terug krijgen van de aanvragende database dat de operatie voltooid is om een nieuwe user aan te maken.

We veronderstellen dat clients telkens op een correcte manier afsluiten. We voorzien geen functionaliteit die een spel correct afsluit wanneer een speler middenin het spel zijn applicatie bruusk sluit.

Hiernaast zou ook de usecase kunnen behandeld worden waarbij een speler kan meekijken naar een spel zonder zelf deel te nemen. Ook wegens tijdgebrek was het niet mogelijk deze functie er nog bij te voegen. Het gebruik van Move-objecten ondersteunt deze mogelijkheid echter wel, waarbij users zelfs heen en terug in een spel zouden kunnen navigeren.

5.3 Uitbreidingen

- Verschillende deck types met bijvoorbeeld extra veel speciale kaarten
- Mogelijkheid voorzien om een actief spel te verlaten
- Voorzien van een toeschouwersfunctie waarbij spelers stap voor stap de moves kunnen afspelen.
- Mogelijkheid voorzien om account te wijzigen/verwijderen