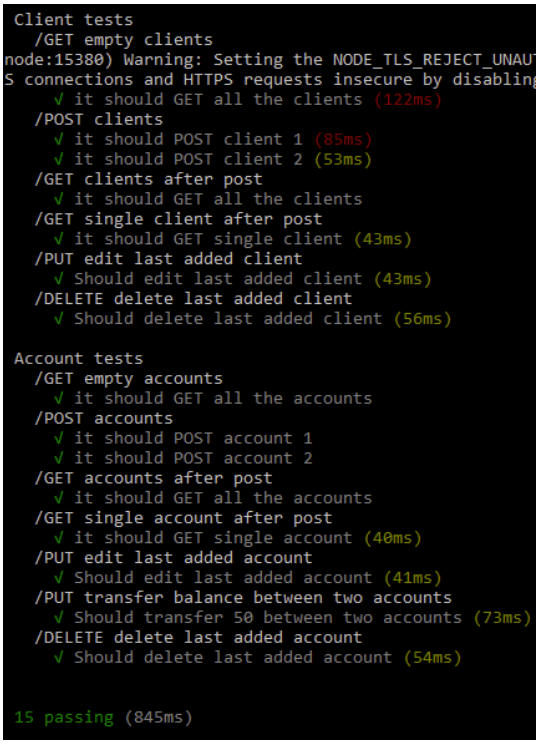
Fag: Distribuerede systemer (BINTO1067E) Studienummer :137643
Studieretning: Ha(it.) – Erhvervsøkonomi – Informationsteknologi

08/12/2020

Distribuerede Systemer

Mundtlig prøve pba skriftligt produkt

Statistics:	
Pages	15
Words	4.393
Characters (no spaces)	24.407
Characters (with spaces)	28.793
Paragraphs	147
Lines	412



Abstract

This report aims to explain the functionality of the developed system and some of the choices made regarding the implementation of the systems sub-elements. The first sections of the report explain the fundamental requirements of the system.

The system is a banking application, secured with SSL encryption and a load balancer which distributes the incoming traffic across the different servers. The system allows for multiple instances of the same server to be run at the same time. These servers all use the same underlying database. Because of the simplicity of the database structure a non-relational database – MongoDB, was chosen for the system.

The load balancer is constructed using the Round Robin Method, which distributes the traffic using a rotating list of servers. SSL is implemented between the server and the load balancer and the load balancer and the browser to ensure the protection of sensitive information. Seaport[1] is used to store the running instances of the server, which the load balancer then can redirect the incoming traffic to.

The system meets the requirements specifications (Bilag 1). The users of the system can interact with the system using the pre-determined endpoints (Bilag 1). There are elements of the system which can be improved on, but this system aims to represent at version of a real-world banking application, which it successfully does.

Christopher Maximilian John André
COPENHAGEN BUSINESS SCHOOL

Indholdsfortegnelse

1. Introduktion

Det bagvedliggende projekt, for denne rapport, var at implementere et banking system. Systemet er bygget op i JavaScript og suppleres med Nodejs, OpenSSL, samt MongoDB.

Rapporten har til formål at beskrive udviklingen, og begrunde nogle af de valg truffet i udviklingen af systemet. Systemet repræsenterer en forenklet version af et reelt banking system. Netværkstrafikken i systemet er SSL krypteret. Systemet understøttes af en load balancer og har muligheden for at oprette flere instanser af en server, som benytter sig af den samme underliggende database.

Rapporten vil berøre de funktionelle og ikke-funktionelle krav samt den underliggende datamodel for systemet. Derefter vil rapporten komme ind på implementeringen af systemets krav. Diskussionen tager udgangspunkt i implementationen og opsummerer nogle af de uhensigtsmæssige problemer med systemet. Til sidst vil rapporten omhandle en række teoretiske spørgsmål og en konklusion.

2. Kravspecifikation

I udarbejdelsen af systemet er der implementeret en række funktionelle og ikke-funktionelle krav. Dette afsnit vil belyse, hvordan disse krav er opfyldte ved at forklare programmets funktionalitet.

2.1 Funktionelle krav

Der er gjort brug af to datamodeller for at strukturere data i systemet. Datamodellerne bestemmer den logiske struktur af databasen, og dermed hvordan data kan gemmes, organiseres og manipuleres. Systemet indeholder to modeller: Accounts (konti) og Clients (kunder). De to datamodeller sikrer en ensformig struktur og understøtter funktionaliteten i systemet:

Clients:

- id
- firstname
- lastname
- streetAddress
- city

Accounts:

- id
- client_id
- balance
- alias

Alle Accounts og Clients bliver tilknyttet et id. MongoDB generer som standard et unikt ObjectID-id. Der tildeles et id-felt i de oprettede Accounts og Clients. Det unikke id bliver brugt i forbindelse med CRUD.

CRUD er et akronym, der henviser til fire funktioner, der generelt er nødvendige for at implementere vedvarende lagring: Opret, læs, opdater og slet.

Opret: Giver brugeren mulighed for at oprette en ny post (Client/Account) i databasen. *Læs*: Svarer til en søgefunktion i en database. Brugeren har muligheden for at søge

og hente specifikke poster gemt i databasen. *Opdatere*: Bruges til at ændre eksisterende poster, der allerede findes i databasen. *Slet*: Fjerner poster der ikke længere er relevante for databasen. F.eks. En kunde ønsker at lukke sine kontoer, og personens oplysninger skal derfor slettes fra databasen. Ovenstående funktionalitet understøttes i systemet ved brug af et API.

De før definerede endpoints, der skal håndtere funktionaliteten i systemet, kan ses i *Use Case Diagrammet (3.2 Use-Case Diagram)*

2.2 Ikke-funktionelle krav

Systemet indeholder også en række ikke-funktionelle krav:

SSL: Systemet gør brug af SSL (Secure Socket Layer) for at sikre, at kommunikationen mellem brugeren og systemet er sikker. Det er især vigtigt, når der skal udveksles information såsom; personlige oplysninger og betalingsoplysninger.
SSL – Secure Sockets Layer er en protokol, der bruges til at etablere en sikker krypteret forbindelse mellem serveren og hjemmesiden. Implementationen af SSL sikrer at uvedkommende ikke kan opsnappe informationen som bearbejdes af systemet.

Load balancer: Systemet understøttes af en load balancer. Load balancing er en metodisk og effektiv distribution af netværkstrafik på tværs mellem flere servere. En load balancer befinder sig mellem klienten og backend-servere, den modtager og distribuerer de indgående anmodninger til de tilgængelige server.

Der er implementeret en one-click løsning til systemet. En one-click run sikrer at systemet starter korrekt, uden at manuelt skulle starte de forskellige delsystemer og services, såsom seaport.

3. Data model

Systemet bruger en NoSQL database – MongoDB. Forskellen mellem at bruge en ikke relationel database og en relationel database er minimal for dette system. Databasestrukturen er simpel og gør ikke brug af mange komplicerede relationer.

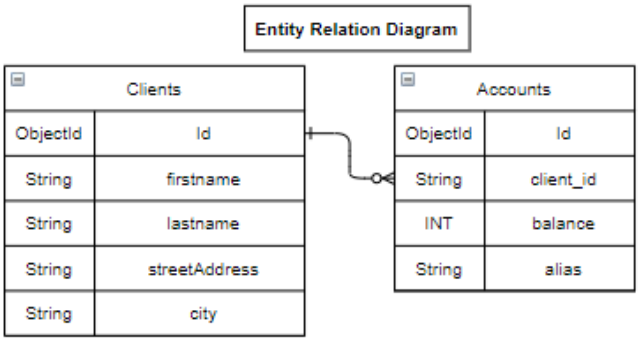
Systemet indeholder to MongoDB Schemas[1]: Accounts og Clients. Et schema er et json-objekt, der giver muligheden for at definere form og indehold på dokumenterne i en samling.

[1] Bilag 2 – MongoDB schemas

Et Entity Relationship Diagram (ER-Diagram) kan med fordel benyttes for at illustrere de to MongoDB Schema og deres relationer til hinanden.

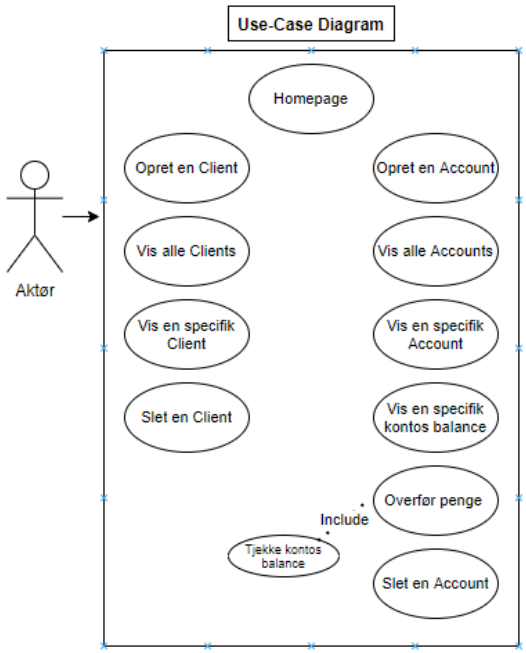
3.1 Entity-Relationship Diagram

ER-Diagrammet[2] beskriver de to MongoDB schema i databasen med deres tilhørende attributter. Tidligere i rapporten blev der redegjort for, at MongoDB selv opretter et ObjectID-id. Derfor er id heller ikke inkluderet i koden, hvor de to MongoDB schema defineres, /models/account.js og /models/client.js. De generede id er stadig en del af databasen og er derfor inkluderet i ER-diagrammet. Diagrammet beskriver en one-to-many relation mellem Clients og Accounts. Det er muligt for en Client at have mange Accounts, men en Account kan kun tilhøre en specifik Client.



3.2 Use-Case Diagram

Use-case diagrammet[3] viser de funktionelle krav for systemet og giver et overblik over, hvilke funktioner aktøren har mulighed for at interagere med i systemet. En aktør har muligheden for at benytte sig af funktionerne ved at tilgå de tilhørende endpoints. Diagrammet er en illustration af, hvordan systemet opfylder de funktionelle krav for endpoints, der er givet for projektet (Bilag 1)[4].



[2] Udarbejdet i draw.io
[3] Udarbejdet i draw.io
[4] Bilag 1 – Endpoints – Kravspecifikation

4. Implementation

Dette afsnit forklarer, hvordan systemet er bygget op for at opfylde kravspecifikationen og beskriver, hvordan aktøren kan interagere med systemet.

4.1 Eksterne system og interface (API tilgangen)

En API (Application Programming Interface) er en softwaregrænseflade, der definerer interaktionen mellem software. En API definerer en række brugbare funktioner, som andre eksterne programmer kan bruge i forbindelse med at udføre et request. Når du skriver en hjemmeside i søgefeltet i din browser, går en request (anmodning) ud til en fjernserver. Din browser modtager et svar, som den fortolker og viser til dig.

F.eks. hver gang du besøger Facebook, interagerer du med deres fjernservers API. En API er ikke det samme som en ekstern server. API er snarere en del af serveren, som håndterer requests og sender et svar tilbage.

Forestil dig, at du sidder ved et bord på en valgfri restaurant. Køkkenet er en del af deres system, men du mangler et forbindelsesled for at sende din anmodning (bestilling af mad), og modtage dit svar (mad). Det er her API'en kommer ind. API'en agerer som en tjener, der modtager en request (bestilling) og formidler denne videre til køkkenet. Derefter leverer tjeneren (API) svaret tilbage til dig, i dette tilfælde er det maden.

Systemet gør brug af et API for at udføre de metoder[5] opstillet i kravspecifikationen. Hvis en aktør ønsker at se deres balance for en account, er det opbygget en API der håndterer requesten. En Mongoose metode henter en accounts balance i databasen, og API'en returnerer et svar. Måden hvorpå aktøren bestemmer, hvilken request de ønsker at sende er ved at gå ind på en række forudbestemte endpoints. Disse endpoints og deres tilknyttede metoder kan ses i Bilag 1[6].

Systemets API befinder sig i filen app.js. Det er i denne fil, der skabes en forbindelse til databasen, samt importerer routes.

```
16  // Require DB
17  const db = require("./db.js");
18  // Import Routes
19  const accountRoute = require("./routes/accounts");
20  const clientRoute = require("./routes/clients");
21  // Define Routes
22  app.use("/accounts", accountRoute);
23  app.use("/clients", clientRoute);
```

```
42    db.getConnection().then(() => {
43      console.log("Database connected");
```

Systemet benytter sig af Express. Express er et Node js-webapplikationsserver framework, som er designet til opbygning af webapplikationer. Koden viser, hvordan routes bliver importeret og tildelt i systemet. Når en

[5] Get, Post, Put, Delete
[6] Bilag 1 – Endpoints – Kravspecifikation

aktør prøver at tilgå et endpoint, der gør brug af /accounts, skal de routes opsat for accounts benyttes. Linje 16, samt 42-43, importerer og gør brug af *getConnection()*[7] metoden for at oprette en forbindelse til databasen.

Der blev tidligere redegjort for, at en API er en softwaregrænseflade. Der er gjort brug af en REST API i systemet. REST bestemmer strukturen af API'en, og hvordan den interagerer med aktøren. REST står for *Representational State Transfer*. REST API tillader aktøren at få hentet data ved at tilgå en bestemt URL. Hver URL håndterer anmodninger og returnerer et svar i form af json. Det er hensigtsmæssigt at bruge en REST API, da vores server er stateless (der lagres ikke nogen data). Dette betyder at hver klientanmodning behandles uafhængigt og ikke som en del af en ny eller eksisterende session.

For at etablere kommunikation mellem databasen benyttes *Mongoose*, den agerer som en ekstern API, der fungerer som et interface mellem databasen og systemet. Mongoose er et ODM-bibliotek til MongoDB. Det styrer forholdet mellem data samt giver skemavalidering og bruges til at oversætte mellem objekter i koden og disse objekter i MongoDB. Mongoose er brugt til at konstruere de to MongoDB Schemas, der er omtalt i datamodel. Mongoose giver muligheden for at arbejde med en lang række Mongoose-metoder. Disse metoder er med til at understøtte funktionaliteten i systemet og er brugt flittigt i kodningen af hver endpoint.

```
35  // Endpoint for showing a specific Client by id
36  routerClient.get("/:id", async (req, res) => {
37    try {
38      let oneClient = await clientModel.findById(req.params.id).exec();
39      res.json(oneClient);
40    } catch (err) {
41      res.status(400).json("Error " + err);
42    }
43  });
```

Mongoose-metoden *findById* benyttes i dette endpoint (/clients/id – Get request) for at finde en specifik bruger i databasen på baggrund af det medsendte id. (routes/clients – linje 35-43)

Der er implementeret en række statements i systemet, der sikrer at metoden returnerer et svar. Alle endpoints gør brug af en *await* og en *exec()*. await spiller en væsentlig rolle, fordi systemet gør brug af et eksternt framework Mongoose, sikrer await at Mongoose metoden har eksekveret før resultatet bliver returneret. exec() benyttes til at returnerer resultatet som et promise til den asynkrone funktion.

4.2 Sikkerhed (kryptering)

Systemet kører på en server, som modtager request og sender responses. Systemet interagerer med person- og kreditkort oplysninger som kan være følsomme. Det er derfor nødvendigt at implementere en form for kryptering.

[7] Bilag 3 – db.js

SSL benytter sig af symmetrisk og asymmetrisk kryptering. Symmetrisk kryptering kendetegnes ved brug af en key, der kan kryptere/dekryptere data. Asymmetrisk kryptering kendetegnes ved brug af public/private keys. Alle parter har adgang til din public key. Data krypteret med din public key, kan kun dekrypteres med din private key. Disse keys bliver generet ved brug af en algoritme, det er gjort med OpenSSL[8]

SSL-certifikater er digitale pas, der giver godkendelse for at beskytte integriteten af et websteds kommunikation med browsere. SSL-certifikaternes job er at starte en sikre session med aktørens browser via SSL-protokollen.

F.eks. en browser prøver at oprette en forbindelse til en webserver, der er beskyttet med SSL. Browseren anmoder om, at webserveren identificerer sig. Webserveren sender browseren en kopi af sit SSL-certifikat. Browseren tjekker, om den stoler på dette SSL-certifikat eller ej, i tilfælde at browseren stoler på SSL-certifikatet sender den en besked tilbage til serven, som indeholder en key. Webserveren sender en digital bekræftelse tilbage og en SSL krypteret session startes. Browseren og webserveren kan derefter dele data sikkert.

Normalt vil brugeren af systemet, ved modtagelse af et certifikat fra en server, få dette godkendt af en tredje part for at sikre, at serven er troværdig. For simplificering af systemet self-signer systemet certifikaterne. Dette er ikke optimalt, men selvom systemet gør brug af et self-signed certifikat, er fuld kryptering implementeret. Aktøren vil blot få en advarsel om at certifikatet ikke er autoriseret af en tredje part.

SSL og HTTPS er implementeret i filen app.js (app.js – linje 25-32). Der bliver oprettet en SSL server, der indeholder en genereret key og certifikat. Denne server bliver senere sat til at *listen* ved brug af seaport[9].

```
25  // Starting ssl server
26  const sslServer = https.createServer(
27    {
28      key: fs.readFileSync(path.join(__dirname, "cert", "key.pem")),
29      cert: fs.readFileSync(path.join(__dirname, "cert", "cert.pem")),
30    },
31    app
32  );
```

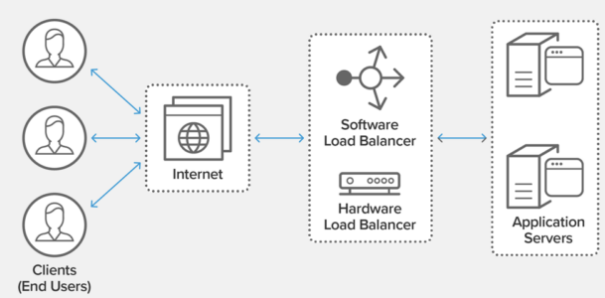
[8] Open Source command line took, som bruges til at generere keys
[9] Service Registry and port assignment for clusters

4.3 Skalering (load balancer)

Beskrevet tidligere i kravspecifikation understøtter systemet en load balancer, der distribuerer netværkstrafikken på tværs af serverne. Load balancern befinder sig mellem klienten og backend-serveren. Fordelen ved at bruge en load balancer er, at den kan øge kapaciteten af aktører og pålideligheden af applikationen. Den mindsker byrden for hver enkelt server, ved at administrere byrden ligeligt på de forskellige server, dermed gør den også systemet mere skalerbart.

En load balancer bruger en algoritme til at bestemme, hvordan requests skal fordeles på de tilgængelige servere, dette er gjort ved at implementere Round Robin Metoden.

Round Robin er en relativt simpel teknik for at sikre, at hver klientanmodning bliver videresendt til en anden server baseret på en roterende liste.[10]



En load balancer agerer som en trafikbetjent, der sidder foran serverfarmen og dirigerer anmodninger på tværs af serverne. Dette maksimerer hastigheden og kapacitetsudnyttelsen, samt sikrer at ingen server er overbelastede. Load balancer er fleksible, da der kan tilføjes eller fratrækkes servere efter behov, og derfor nemt skaleres.

Systemet har muligheden for at starte flere instanser af en server. Seaport benyttes til at gemme instanserne af tilgængelige servere. Seaport[11] er et middleware, der benyttes til service register og port tildeling for cluster (serverfarm). Systemets load balancer gør også brug af http-proxy[12]. http-proxy er et http programmerbart proxy-bibliotek. Det er velegnet til implementeret af load balancers, da det sørger for at den instans af serveren seaport har gemt, håndterer den request som aktøren har sendt. Ligesom vores app.js bruger sig af SSL, er det også vigtigt at kryptere kommunikationen til og fra load balancern, der oprettes derfor en SSL-server i load balancern.

[10] Billede - https://www.nginx.com/resources/glossary/load-balancing/
[11] https://www.npmjs.com/package/seaport
[12] https://www.npmjs.com/package/http-proxy

```
15 var server = https.createServer(
16   {
17     key: fs.readFileSync(path.join(__dirname, "../cert", "key.pem")),
18     cert: fs.readFileSync(path.join(__dirname, "../cert", "cert.pem")),
19   },
20   function (req, res) {
21     var addresses = sp.query();
22     if (addresses.length == 0) {
23       res.end("No Servers Available");
24     }
```

Funktionen (linje 20) gemmer alle instanser af serverne i en variabel *addresses* ved brug af en *query()*. q*uery()* metoden skaber et array der indeholder alle de tilgængelige servere. Der er inkluderet et *if statement* for at sikre, at der er tilgængelige servere.

I tilfælde af at der er tilgængelige servere, skal load balancern nu distribuere trafikken ligeligt mellem serverne:

```
9  // Reference til step 4 (modulus)
10 var i = -1;
```

En variabel *i* defineres i til -1. Hver gang load balancern modtager en request stiger værdien af i med + 1, det er et essentielt step for Round Robin metoden. Afhængigt af værdien af i bestemmer metoden, hvilken server requesten skal videresendes til. Modulus (%) benyttes i linje 26 til at bestemme hvilken server der modtager næste request.

```
25     // Reference til step 4
26     i = (i + 1) % addresses.length;
27     var host = addresses[i].host.split(":").reverse()[0];
28     var port = addresses[i].port;
29     console.log(port);
30     proxy.web(req, res, {
31       target: "https://" + host + ":" + port,
32       secure: false,
33     });
34   }
35 );
```

Til højre er der opstillet et scenarie, hvor systemet kører 2 instanser af serveren. Variablen addresses indeholder derfor 2 instanser af servere, hvilket medfører addresses.length = 2.

Modulus virker ved at tjekke for en rest. Koden på linje 26 undersøger, hvor mange gange 2 går op i 0 – hvilket den gør 0 gange. To variable *host* og *port* defineres ud fra hvilket indeks i arrayet requesten skal sendes til. Ved brug af proxy.web videresendes requesten til den rigtige servere.

Request 1:
i = - 1
i = (-1 +1) % addresses.length
0 % 2 = 0

Request 2
i = 0
i = (0 + 1) % addresses.length
1 % 2 = 1

Request 3
i = 1
i = (1 + 1) % addresses.length
2 % 2 = 0

Ved næste request kører load balancern den samme kode, den eneste ændring er at nu at i = 0, i stedet for -1, hvilket giver en anden rest værdi og dermed andet index i arrayet med tilgængelige servere.

4.4 Interessant stykke kode - /clients/:id – Put Request

Dette endpoint har til formål at ændre en clients informationer. Mongoose metoden *.findByIdAndUpdate* benyttes til at finde en client på baggrund af deres specifikke id, og ændre deres informationer.

```
58  routerClient.put("/:id", async (req, res) => {
59    try {
60      let updateClient = await clientModel
61        .findByIdAndUpdate(
62          req.params.id,
63          {
64            $set: req.body,
65          },
66          { new: true }
67        )
68        .exec();
69      res.json(updateClient);
70    } catch (err) {
71      res.status(400).json("Error " + err);
72    }
73  });
```

Koden kan findes i routes/client.js – linje 58 - 73

Der er to måder at sende data til systemet. Et json-objekt med en body der indeholder informationer. Aktøren kan også medsende en parameter direkte i URL'en. I dette endpoint henter metoden det specifikke id fra URL'en – linje 62. Systemet henter dette id ved at request parameteren *id* fra URL'en.

Endpointet er specielt, fordi det ikke er forudbestemt, hvilke informationer der skal ændres. Aktøren har mulighed for at ændre en eller flere informationer. Det formidles til systemet i json-objektet, der indeholder de nye oplysninger.

I systemet bruges *$set* operatoren til at erstatte værdier i et felt i databasen. Hvis et felt ikke findes i databasen, tilføjer $set operatoren feltet til den angivne værdi. Årsagen til at $set er implementeret er, at den kun overskriver de værdier defineret i json-objektet. Hvis $set ikke var implementeret, ville MongoDB sætte værdien af alle de ikke ændrede felter til null. De nye informationer hentes fra den medsendte body, og de nye data sættes til at være true (linje 66), ellers vil MongoDB ikke gemme ændringerne i databasen.

Hvis systemet modtager denne body, vil det kun ændre balancen og alias for en client. De andre felter vil forblive uændret.

```
31  Content-Type: application/json
32
33  {
34      "balance": 123456789,
35      "alias":"Altered Alias"
36  }
```

Eksemplet viser et mere fleksibelt og skalerbart endpoint. De andre endpoints i systemet der tilføjer data, håndterer kun den data fra body der er defineret i endpointet. Når vi opretter en ny client, skal det følge det oppesatte skema for en client, der kan derfor ikke oprettes nye felter. $set tillader aktøren at medsende flere parametre i json-objektet, og $set vil automatisk oprette et felt i MongoDB. Dette modvirker dog hensigten om at have en ensartet struktur i systemet.

5. Diskussion

Der er en del uhensigtsmæssige elementer i systemet, de vil i dette afsnit forklares, og jeg vil argumentere for hvordan, de muligvis kunne forbedres.

5.1 Rest API / Webinterface

Med hensigten om at implementere et brugbart system for en rigtig aktør i virkeligheden, er systemet ikke særligt brugervenligt. Endpoints, der har til formål at læse/manipulere data, kræver at der medsendes et json objekt eller specifikt id i URL'en.

Json-objekter sendes ved brug af REST Client[13] extension i Visual Studio Code, andre tredje part programmer kan også benyttes, såsom *postman*[14]. I den virkelige verden ville der optimalt være implementeret en form for webinterface som aktøren nemt og overskueligt kan interagere med. F.eks. ville det være optimalt, hvis en aktør kan finde informationer ved at søge på et kundenummer i et søgefelt, og ikke skulle sende et autogenereret id med i URL'en som en parameter. Aktøren ville også få returneret informationen overskueligt og ikke i json format, som systemet gør i dag.

5.2 Self-Signed Certifikater

Et self-signed certifikat er et certifikat, der ikke er godkendt af en tredje part, såsom certifikatmyndigheden (CA). Det er gratis at lave et self-signed certifikat, de leverer dog ikke den samme funktionalitet som et godkendt certifikat. En aktør, der prøver at tilgå en hjemmeside, der bruger et self-signed certifikat, vil få en advarsel i deres browser. Aktører, der ignorer denne advarsel og stadig benytter hjemmesiden kan muligvis udsætte sig selv for en risiko, at en anden kan opfange deres trafik. Selv om systemet certifikat ikke er autoriseret, er kryptering implementeret, det er bare ikke godkendt af certifikatmyndigheden.

5.3 Round Robin Load Balancer

Round Robin metoden er valgt for dette system. Den distribuerer trafikken ligeligt mellem de tilgængelige servere. Det er dog ikke svært at forestille sig en situation, hvor serverne har forskellige kapaciteter. Den fleksible natur af load balancers tillader os at tilføje flere evt. bedre servere i fremtiden. Det er derfor ikke optimalt at netværkstrafikken distribueres ligeligt, når nogle servere kan være bedre/hurtigere til at håndtere requests end andre. Round

[13] REST Client, Huachao Mao , v0.24.4
[14] https://www.postman.com/

Robin vil i sådan et tilfælde ikke udnytte servernes fulde kapacitet. Der findes flere alternative metoder som bedre kan håndtere sådan en situation;

Weightet load balancing tillader udvikleren at tildele hver enkelt server en vægt (en andel af requests), hvor man vil tildele de bedre servere en større andel, for at bedre udnytte servernes fulde kapacitet.

Der findes mere sofistikerede metoder, såsom *Least Response Time* metoden. Denne metode distribuerer netværkstrafikken afhængigt af, den tid det tager en server at svare på en anmodning om sundhedsovervågning. Tiden, det tager serveren at svare, er en god indikator for, hvor belastet serveren er.

6. Teoretiske opgaver

De givne teoretiske spørgsmål kan findes i Bilag 4[15]

Hvad er fordelen ved at have en protokol-stack som er inddelt i lag (fem-lags modellen) i forhold til en uopdelt en-lags model?[16]

Ved at inddele protokollerne i lag giver vi dem en struktur. En protokol-stack inddelt i flere lag, gør det muligt at udvikle standarder, men det gør det også nemmere at tilpasse sig ny hardware og software over tid. Protokollerne for TCP/IP fem-lags modellen blev udviklet tilbage i 70'erne. På grund af opdelingen af protokollerne har det været muligt for udvikler at ændre og opdatere enkelte protokoller, uden at påvirke de resterende modeller.
F.eks. flytningen fra IPv4-adressering til IPv6-adressering påvirker kun protokollerne for netværkslager, de andre lag forbliver upåvirket. Det gør det muligt at foretage forbedringer uden at skulle omdefinere hele TCP/IP-protokollen.

Standarder gør det også nemmere for udvikler at bestemme hvilke protokoller de skal benytte sig af. Ved at bruge TCP/IP modellen, skal de ikke selve konstruere et transport-lag, men kan vælge mellem standarder såsom TCP eller UDP, afhængigt af deres krav til transport-laget.

[15] Bilag 4 – Teoretiske spørgsmål
[16] Bilag 4 – Teoretiske spørgsmål – 1

Forklar forskellen på TCP og UDP med særligt fokus på, hvordan TCP opnår leveringsgaranti på lange byte sekvenser, og hvordan UDP opnår dette? [17]

Transmission Control Protocol (TCP) er en forbindelsesorienteret protokol til forskel for UDP. En TCP-forbindelse oprettes ved hjælp af et *three-way-handshake*. Det er en proces, der starter en forbindelse og anerkender forbindelsen. Først når forbindelsen er oprettet, begynder dataoverførelsen. TCP sender pakker i en specifik rækkefølge og benytter sig af *error-checking* på disse pakker, samt *error-recovery*. TCP opnår leveringsgaranti ved at tildele et sekvensnummer til hver oktet, det sender, og kræve en positiv bekræftelse fra modtageren. Hvis denne bekræftelse ikke modtages før timeout-intervallet, sendes dataene igen. Den sikkerhed TCP tilbyder betyder, at hastigheden for TCP er betydelig langsommere end UDP. TCP benyttes derfor i scenarie, hvor det er essentielt at modtageren modtager hele pakken.

Datagram Oriented Protocol (UDP) er en forbindelsesløst protokol. UDP bruger en simpel transmissionsmetode uden underforståede *handshake*. UDP sender datapakker ud på netværket og accepterer indgående datagrammer hos modtageren. UDP garanterer ikke levering eller den korrekte rækkefølge på pakkerne. Det giver dog muligheden for brug af checksum, til at kontrollere integriteten af dataene hos modtagere. Selvom UDP giver denne verifikation af integritet, giver den ingen garanti for levering. UDP bliver dog stadig brugt, når der fokuseres på at sende mange pakker hurtigt. UDP benyttes derfor i scenariet, hvor det er essentielt, at hastigheden er hurtigt, og det ikke gør noget, hvis der mistes pakker undervejs. Den bliver generelt brugt meget til computerspil og streaming-tjenester.

Hvorfor er der brug for både MAC- og IP-adresser? Hvor er MAC-adresser alene ikke nok til at route internettrafikken? [18]

Både Mac-adresser og IP-adresser bruges til at identificere en maskine på internettet. Mac-adresser leveres af chipproducenten, mens IP-adressen leveres af internetudbyderen. Den afgørende forskel mellem Mac- og IP-adresser er, at Mac-adresser bruges til at definere computerens fysiske adresse - enheden på et netværk. En IP-adresse bruges til at identificere et netværket.

[17] Bilag 4 – Teoretiske spørgsmål – 2
[18] Bilag 4 – Teoretiske spørgsmål – 3

Det vil sige, IP-adressen bruges til at identificere et netværk, og Mac-adressen bruges til at identificere en specifik enhed på det netværk. Det er derfor nødvendigt at bruge både Mac- og IP-adresser for at oprette en forbindelse. Ved lokale små netværk er det nok at kende en Mac-adresse, men internettet er stort, derfor benytter vi os af IP-adresser for at definere kommunikation på tværs af forskellige netværk. Hvis Mac-adressen for en computer er kendt, men computeren befinder sig på et helt andet netværk, vil det ikke alene være nok for at oprette en forbindelse. Netværket skal først identificeres ved brug af IP-adressen, derefter kan enheden identificeres ved brug af Mac-adressen.

En analogi hertil er postlevering. En IP-adresse kan betragtes som en hjemmeadresse. Mac-adressen er din fysiske identitet, dit navn, alder, køn osv. Uden en IP-adresse vil postvæsnet ikke vide, hvilket hus de skal levere posten til, og uden en Mac-adresse vil postvæsnet ikke vide, hvilken person på adressen, de skal levere pakken til. Derfor er både en Mac- og IP-adresse nødvendigt for at route internettrafik.

Hvad er fordelen ved Distribuerede routing algoritmer? Hvorfor konfigurere vi ikke bare alle routere manuelt?[19]

Når en aktør ønsker at oprette en forbindelse på tværs af netværk, ønsker man at gøre det så hurtigt som muligt. Det vil sige den korteste vej gennem netværket. Det bliver gjort ved at benytte routing algoritmer, som betragter omkostningen ved at hoppe fra node til node (router) i netværket. I dag er det svært at konfigurerer routing manuelt, udelukkende på grund af den enorme størrelse af netværker. For at nå en bestemt server i dag, foretages mange hop fra node til node. Det er umuligt at bestemme den bedste/hurtigste vej på forhånd, derfor benyttes routing algoritmer, som udregner den relative bedste vej, alt efter hvor enheden befinder sig i netværket.

Dynamiske routingprotokoller fungerer bedst i enhver form for netværk, der består af flere routere. De er skalerbare og bestemmer dynamisk den bedste vej, hvis der er en ændring i netværket - f.eks. kan en router stoppe med at virke. En statisk routingprotokol vil fejle, hvis en router stopper med at virke og vil kræve manuel re-routing. Statisk routingprotokol bliver derfor mere kompliceret i takt med at netværket vokser og er ikke særligt skalerbart.

[19] Bilag 4 – Teoretiske spørgsmål – 4

Hvilken salgs kryptering bruges til data som "står stille" / er lagret? Og hvilken slags bruges på data i bevægelse (data der sendes over et netværk? Hvorfor er der en forskel på disse?[20]

Data i bevægelse er data, der aktivt bevæger sig fra en lokation til en anden. Beskyttelse af disse data, mens de rejser fra netværk til netværk, er kritisk, da data ofte er betragtes som mindre sikker under bevægelse. Data i bevægelse kan beskyttes ved brug af keys, blandt andet SSL (beskrevet i 4.2 - SSL), hvor det krypteres ved hjælpe af private/public keys.

Data i hvile (står stille) er data, der ikke aktivt bevæger fra et netværk til et andet, såsom data gemt på en harddisk, flashdrev eller database. Data i hvile kan benytte sig af en hashing-algoritme til at kryptere den lagrede data, inklusiv nøgleordet. Når aktøren vil have adgang til det lagrede data, benyttes nøgleordet til at dekryptere data.

7. Konklusion

Systemet overholder de generelle krav oplyst i kravspecifikationen. Netværkstrafikken mellem både server og load balancer samt load balancer og browseren er SSL krypteret. Systemet understøttes af en funktionel load balancer, der gør brug af Round Robin Metoden, som distribuerer den indkommende trafik ved hjælp af en roterende liste af tilgængelige servere. Systemet har muligheden for at oprette flere instanser af en server, som benytter den samme underliggende database.

Funktionaliteten vist i Use-Case diagrammet kan tilgås af aktøren, ved brug af de før definerede URL[21] - det fremgår også af den inkluderede video.

Der er nogle delelementer af systemet, der kunne forbedres på. Round Robin Metoden er en effektiv måde at distribuere netværkstrafikken i systemet, men det har også sine begrænsninger. Selv om systemet self-signer SSL certifikaterne er kryptering implementeret i systemet. Aktøren vil blive afvarslet om, at certifikatet ikke er autoriseret af en tredje part.

Der er elementer af systemet, som kan videreudvikles, men systemets formål er at repræsentere en fungerende version af et banking systemet, hvilket det gør med succes.

[20] Bilag 4 – Teoretiske spørgsmål – 5
[21] Bilag 1 – Endpoints – Kravspecifikation

8. Litteraturliste

8.1 Bøger

- Kurose & Ross, Computer Networking: A Topdown Approach 7th ed.

8.2 Videoer

- [WebConcepts]. (2014, 14. Juli). *REST API concepts and examples.* [Youtube]. Lokaliseret på;

https://www.youtube.com/watch?v=7YcW25PHnAA&t=324s.

- [Canvas], *BINTO1067U.LA (Distribuerede systemer).* Lokaliseret på;

https://cbscanvas.instructure.com/courses/11500

8.3 Hjemmesider

- [MongoDB]. *The MondoDB 4.4 Manual.* Lokaliseret på;

https://docs.mongodb.com/manual/

- [ENTRUST]. *HOW DOES SSL WORK?* Lokaliseret på;

https://www.entrust.com/resources/certificate-solutions/learn/how-does-ssl-work#:~:text=The%20browser%2Fserver%20requests%20that,copy%20of%20its%20SSL%20certificate.&text=The%20web%20server%20sends%20back,server%20and%20the%20web%20server

- [CITRIX]. *What is load balancing?* Lokaliseret på;

https://www.citrix.com/en-in/glossary/load-balancing.html

- [NGINX]. *What is load balancing?* Lokaliseret på;

https://www.nginx.com/resources/glossary/load-balancing/

- [Slack]. *CBS DIS 2020.* Lokaliseret på;

https://app.slack.com/client/T019RSP59BL/C01EZ9R3FCY

- [Cisco Press]. *Cisco Networking Academy's Introduction to Routing Dynamically*. Lokaliseret på;

https://www.ciscopress.com/articles/article.asp?p=2180210&seqNum=5

- Zell, L. (2018, 17. januar). *Understanding And Using REST APIs.* Lokaliseret på;

https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/

- [Guru99]. *TCP vs UDP: What's the difference?* Lokaliseret på;

https://www.guru99.com/tcp-vs-udp-understanding-the-difference.html#:~:text=TCP%20is%20a%20connection%2Doriented,speed%20of%20UDP%20is%20faster&text=TCP%20does%20error%20checking%20and,but%20it%20discards%20erroneous%20packets.

- [MuleSoft]. *What is an API? (Application Programming Interface)* Lokaliseret på;

https://www.mulesoft.com/resources/api/what-is-an-api#:~:text=What%20Is%20an%20Example%20of%20an%20API%3F&text=That's%20where%20the%20waiter%20or,case%2C%20it%20is%20the%20food.

9. Bilag

9.1 Bilag 1 – Endpoints – Kravspecifikation

endpoint	Metode	body parameters name: Type: req/opt	Beskrivelse
/accounts	GET	-	Returnerer et array af alle konti
	POST	client_id: String: req balance: Number: opt	Opretter en ny konto
/accounts/:id	GET	-	Returnerer en specific konto
	PUT	balance: Number: opt alias: String: opt	Ændrer en konto. .
	DELETE	-	Sletter en konto med det specifikke id
/accounts/:id/balance	GET	-	Returnerer en specific kontos balance i formen {balance: 200}
/accounts/transfer	PUT	fromAccount: String: req toAccount: String: req amount: Number: req	Overfører penge fra en konto til en anden
/clients	GET	-	Returnere et array af alle kunder
	POST	firstname: String: req lastname: String: req streetAddress: String: req city: String: req	Opretter en ny kunde
/clients/:id	GET	-	Returnerer en specific kunde
	PUT	firstname: String: opt lastname: String: opt streetAddress: String: opt city: String: opt	Opdaterer en kundes oplysninger
	DELETE	-	Sletter kunden med det specifikke id

9.2 Bilag 2 – MongoDB schema

```
// Creating Account Schema
const AccountSchema = new mongoose.Schema(
  {
    client_id: {
      type: String,
      required: true,
    },
    balance: {
      type: Number,
      required: true,
    },
    alias: {
      type: String,
      required: true,
    },
  },
  { versionKey: false }
);

// Exporting Account Schema
const model = mongoose.model("Account", AccountSchema);

module.exports = model;
```

```
// Creating Client Schema
const ClientSchema = new mongoose.Schema(
  {
    firstname: {
      type: String,
      required: true,
    },
    lastname: {
      type: String,
      required: true,
    },
    streetAddress: {
      type: String,
      required: true,
    },
    city: {
      type: String,
      required: true,
    },
  },
  { versionKey: false }
);

// Exporting Client Schema
const modelClient = mongoose.model("Client", ClientSchema);

module.exports = modelClient;
```

9.3 Bilag 3 – db.js

```
const getConnection = async () => {
  if (!connection) {
    // 2. Insert the correct db url
    // Your URL should be mongodb://localhost/<database name>, ie. mongodb://localhost/<database name>
    connection = await mongoose.connect("mongodb://localhost/Banking", {
      useNewUrlParser: true,
      useCreateIndex: true,
      useUnifiedTopology: true,
      useFindAndModify: false,
    });
  }
  return connection;
};

module.exports = {
  getConnection: getConnection,
};
```

9.4 Bilag 4 – Teoretiske spørgsmål

1. Hvad er fordelen ved at have en protokol-stack som er inddelt i lag (fem-lags modellen) i forhold til en uopdelt en-lags model?
2. Forklar forskellen på TCP og UDP med særligt fokus på, hvordan TCP opnår leveringsgaranti på lange byte sekvenser, og hvordan UDP opnår dette?
3. Hvorfor er der brug for både MAC- og IP-adresser? Hvor er MAC-adresser alene ikke nok til at route internettrafikken?
4. Hvad er fordelen ved distribuerede routing algoritmer? Hvorfor konfigurerer vi ikke bare alle routere manuelt?
5. Hvilken slags kryptering bruges til data som "står stille" / er lagret? Og hvilken slags bruges på data i bevægelse (data der sendes over et netværk)? Hvorfor er der en forskel på disse?