NTNU
Det skapende universitet

Håndtering av replikeringskonflikter for multi-master dokumentdatabaser ved hjelp av metoder fra versjonskontrollsystemer

Kjell Rydningen Elstad

Master i datateknologi
Innlevert: juni 2017
Hovedveileder: Svein Erik Bratsberg, IDI
Medveileder: Axel Anders Kvale, Norsk Helsenett SF

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknologi og informatikk

Abstract

By allowing write operations on isolated nodes in a replicated database there is a risk of getting replication conflicts. These conflicts are the result of overlapping writes to the same data while there is a network partition. This study assumes that these conflicts cannot be resolved by simply choosing one version and discarding all other versions.

The goal of this study is to find a way to solve replication conflicts automatically in document databases. In this regard, methods from version control systems are suggested as a way to reduce user involvement in the process of resolving the conflict. Specifically, this study compares the components of both systems and the program flow for conflicts, then looks at practical aspects of implementing such a system.

This study suggests that methods from version control systems can be useful in resolving replication conflicts. However, these methods may not solve all conflicts automatically and in some cases this method may not yield the wanted result.

Two major limitations of this work are that the suggested solution has not been implemented and tested in practice; and that it focuses on text based merge, while syntactic or semantic merge may also prove useful. Implementation of such a system and researching the possibility of different methods for merge is suggested as further work.

Sammendrag

Ved å tillate skriveoperasjoner på isolerte noder i en replikert database er det en risiko for at det oppstår replikeringskonflikter. Slike konflikter er resultatet av overlappende endringer på data i en periode hvor det er nettverkspartisjoner. Det her antatt at disse konfliktene ikke kan løses ved å velge én versjon og forkaste alle andre.

Hensikten med dette studiet er å finne en måte å løse replikeringskonflikter automatisk i dokumentdatabaser. I den forbindelse er metoder fra versjonskontrollsystemer foreslått som en måte å redusere brukerinvolvering i prosessen. Dette studiet sammenligner komponenter og programflyt fra begge systemer, samt ser på praktiske hensyn ved implementeringen av et slikt system.

Studiet antyder at metoder fra versjonskontrollsystemer kan være nyttige i håndtering av replikeringskonflikter. Det ligger likevel an til at ikke alle konflikter kan løses automatisk med et slikt system og i noen tilfeller kan det resultere i andre løsninger enn ønsket.

To begrensende faktorer i dette studiet er at den foreslåtte løsningen ikke er implementert og testet i praksis; og at det fokuseres på tekstbasert merge mens syntaktisk- og semantisk merge også kan vise seg å være nyttig. Implementasjon av et slikt system, samt et studie med andre metoder for merge, foreslås som videre arbeid.

Forord

Takk til: Svein Erik Bratsberg, veileder ved NTNU, for veiledning og tilbakemelding underveis; Norsk Helsenett ved Axel Anders Kvale for oppgaven; ansatte og innleide ved avdeling for arkitektur og utvikling hos Norsk Helsenett.

Innhold

Figurer

Kapittel 1

Innledning

1.1 Presentasjon av problemet

Problemet oppgaven tar utgangspunkt i er konflikthåndtering i forbindelse med replikerte multi-master dokumentdatabaser hvor skriveoperasjoner er tillatt også på isolerte noder.

Grunnlaget for situasjonen er at det finnes flere dokumentdatabaser som replikerer data til hverandre slik at alle til enhver tid skal ha samme data. Problemet oppstår som et resultat av at det tillates skriveoperasjoner også til isolerte databaser. Flere databaser kan skrive til samme dokument tilsynelatende samtidig, noe som resulterer i at det kan finnes flere forskjellige utgaver av samme dokument. Når disse ved en senere anledning skal synkroniseres vil det oppstå en konflikt.

Interessen for håndtering av replikeringskonflikter oppsto i forbindelse med videreutvikling av et av Norsk Helsenett sine systemer til en høytilgjengelig løsning. Mer informasjon om dette prosjektet finnes i Vedlegg A.

1.2 Problemstilling

I hvilken grad kan metoder fra versjonskontrollsystemer brukes i forbindelse med håndtering av replikeringskonflikter?

Problemet oppstår i forbindelse med at det kan finnes flere utgaver av samme dokument samtidig. I versjonskontrollsystemer finnes mekanismer for å finne forskjeller mellom ulike versjoner av ei fil, slå sammen ulike versjoner til en, og løse problemer dersom det finnes overlapp i endringer. Kan slike verktøy brukes i forbindelse med replikeringskonflikter? Hvilke forutsetninger må være på plass for at det skal fungere? Hvilke utfordringer finnes i forbindelse med en slik utvikling?

Oppgaven tar utgangspunkt i at det er behov for å kombinere data fra flere dokumenter ved konflikt slik at en løsning som kun velger et dokument og forkaster alle andre ved konflikt ikke er aktuelt.

1.3 Metoder

I løpet av arbeidet med denne oppgaven ble det utviklet et testmiljø. Hensikten med dette testmiljøet var å teste funksjonalitet i RavenDB, samt programflyt i forbindelse med replikeringskonflikter. Det er gjort en sammenligning av programflyten som fører til en replikeringskonflikt (en konflikt i forbindelse med synkronisering av replikerte dokumenter) med arbeidsflyten som fører til en *merge conflict* (en konflikt i forbindelse med sammenfletting av filer) i versjonskontrollsystemer. Det er undersøkt hvorvidt metoder fra versjonskontrollsystemer kan fungere i replikerte dokumentdatabaser.

1.4 Tidligere arbeid

Kuznetsov [1] gir en oversikt over NoSQL databaser, inkludert dokumentdatabaser, og hvordan de fungerer. Den gir også en introduksjon til replikering av data og modeller for konsistens[1]. Terry [2] gjør et studie på replikerte data, samt multi-master datalagring hvor skriveoperasjoner er tillatt på isolerte noder, basert på Bayou, et system for lagring av data. Artikkelen ser også på applikasjonspesifikk konflikthåndtering. Baudiš [3] oppsummerer metoder innen versjonskontrollsystemer og Mens [4] gjør et studie på software merging – sammenfletting av kode for programvare.

1.5 Disposisjon

Resten av oppgaven er strukturert slik: Kapittel 2 og 3 gir en introduksjon til henholdsvis dokumentdatabaser og versjonskontrollsystemer. Kapittel 4 beskriver testmiljøet som ble utviklet for å undersøke replikering og funksjonalitet i RavenDB. Kapittel 5 beskriver muligheter for konflikthåndterere i RavenDB. Kapittel 6 gjør en sammenligning av versjonskontrollsystemer og dokumentdatabaser. Kapittel 7 ser på hvordan en løsning kan fungere i praksis. Til slutt kommer diskusjon i Kapittel 8, konklusjon i Kapittel 9, og videre arbeid i Kapittel 10.

[1] en: consistency

Del I

Bakgrunnsmateriale

Kapittel 2

Dokumentdatabaser

Dette kapittelet gir en kort introduksjon til dokumentdatabaser og komponenter som er relevante for oppgaven. I tillegg gir det en introduksjon til RavenDB som er dokumentdatabasen som ble brukt i arbeidet med denne oppgaven.

2.1 Introduksjon

I en dokumentdatabase blir informasjonen lagret som semantisk strukturerte dokumenter. Informasjonen er gjerne lagret i et format som JSON, XML, eller en variant av dette[1]. Hvert dokument har en primærnøkkel som brukes til oppslag. Dokumentdatabaser har gjerne også en sekundærindeks som kan brukes for å gjøre dokumentsoppslag på andre felter enn primærnøkkelen. [1, 7]

2.2 Json

Tekstformatet JSON [8], *JavaScript Object Notation*, er et av formatene som kan brukes for å lagre data i en dokumentdatabase. Listing 1 er et eksempel på hvordan et Json dokument kan se ut. Forskjellige typer parenteser (`{}[]`) brukes til å representere start og slutt på objekter og lister. Datafelter har først navnet på feltet etterfulgt av et kolon (`:`) og deretter data. Felter er separert med komma (`,`).

Slik informasjonen er formatert i eksempelet, med nye linjer for hvert felt, er ikke nødvendig, men byr på et par fordeler. Det gjør det lettere for et menneske å lese dokumentet og det er godt egnet til bruk mot versjonskontrollsystemer, introdusert i Kapittel 3, da disse ofte sjekker forskjeller mellom filer ved å sammenligne linje for linje.

Som vist i Listing 1 lagres navnet på feltet sammen med informasjonen, men det mangler detaljert typeinformasjon. For eksempel, er det ikke spesifisert at `IndustryCodes` er av typen `IList<Code>`.

[1] Et eksempel er dokumentdatabasen MongoDB [5] som bruker BSON [6], et format som ligner JSON.

```
{
    "Business": {
        "OrganizationNumber": 974749025,
        "Name": "ST OLAVS HOSPITAL",
        ...
        "LastChanged": "2017-05-04T10:10:21.6130000",
        ...
        "IndustryCodes": [
            {
                "OID": 0,
                "SimpleType": "naringskode",
                "CodeValue": "86.101",
                "CodeText": "Alminnelige somatiske sykehus",
                "Active": true
            }
        ],
        ...
    }
}
```

Listing 1: Eksempel på Json fil basert på `Htk/Business.cs` fra NhnDtoContracts [9]. Eksempelet er forkortet. Linjer med `...` indikerer hvor data er ekskludert.

2.3 Replikering

Replikering innebærer at flere databaser, gjerne i forskjellige datasentre på forskjellige geografiske steder, har den samme kopien av den samme informasjonen. På denne måten er informasjonen og tjenester som bruker den bedre beskyttet mot nettverksbrudd og lokale katastrofer. Samtidig forbedres responstiden på leseoperasjoner i systemet fordi lasten kan fordeles på flere databaser. [1]

Når skriveoperasjoner er tillatt i flere databaser som replikerer til hverandre brukes begrepet *multi-master* eller *master-master*. Dette er i kontrast til *master-slave* hvor slave databasene ikke tillater skriveoperasjoner og får all data ved at master replikerer den til slaven. [1] Denne oppgaven tar utgangspunkt i et master-master system.

Oppdatering kan skje synkront slik at alle replikater får oppdateringen samtidig eller asynkront hvor endringer skyves ut til andre databaser etter at de er utført på en. [1] I denne oppgaven brukes asynkron oppdatering ettersom det baserer seg på et multi-master system hvor skriving tillates også på isolerte noder.

2.4 RavenDB

RavenDB [10], også omtalt som Raven i denne oppgaven, er en dokumentdatabase utviklet av Hibernating Rhinos. Ettersom RavenDB brukes i utviklingen av grunndata (beskrevet i Vedlegg A) vil også denne oppgaven bruke det som utgangspunkt. Raven bruker tekstformatet Json, introdusert i Delkapittel 2.2, for lagring av data. For å kunne replikere data i Raven er det nødvendig å aktivere `Replication Bundle`, replikeringspakken, for databasen. I denne oppgaven brukes versjon 3.5 av RavenDB.

2.4.1 Metadata

Raven lagrer metadata sammen med informasjonen den tar vare på. Blant metadata er det informasjon om når dokumentet sist ble oppdatert og hvilken datatype dokumentene som lagres er. Typeinformasjon lagres ved å oppgi navnet på datatypen, som for eksempel `NHN.DtoContracts.Htk.Business` [2] for eksempelet i Listing 1. Dette løser også problemet med at Json-formatet ikke lagrer typeinformasjon direkte. Metadata inkluderer også informasjon om replikering og replikeringshistorikk. [11]

2.4.2 Oppdage konflikter

RavenDB har innebygd deteksjon av konflikter som oppstår i forbindelse med replikering. Dette blir gjort ved å ta vare på replikeringshistorikk i form av versjon / kilde par [11].[3] Ettersom dette allerede er implementert i Raven har denne oppgaven fokusert på hvordan konflikten skal håndteres etter at den er oppdaget.

2.4.3 Historikk

RavenDB tilbyr muligheten til å lagre historikk for dokumenter. Dette gjøres ved at hele dokumentet lagres på nytt med en egen nøkkel[4]. For eksempel kan dokumentet i Listing 1 som har nøkkelen `htkBusiness/974749025` ha versjonene: `htkBusiness/974749025/revisions/1` og `htkBusiness/974749025/revisions/2`. Kapittel 6 og 7 ser nærmere på hvordan metoder fra versjonskontroll kan være interessant i forbindelse med replikerte dokumentdatabaser.

For å benytte historikk i RavenDB må `Versioning Bundle`, versjoneringspakken, aktiveres for databasen.

[2] HTK er forkortelse for *Helsetjenestekatalogen*

[3] Det er planlagt å erstatte dette med *vector clocks* i versjon 4.0 av Raven [12].

[4] Dette vil ikke være tilfellet i versjon 4.0 [13]

Kapittel 3

Versjonskontrollsystemer

Dette kapittelet gir en rask oversikt over noen sentrale konsepter bak versjonskontrollsystemer. Hensikten er å gi en bakgrunn til temaet før Kapittel 6 og 7 som ser nærmere på hvordan erfaring herfra kan være nyttig i forbindelse med utvikling av konflikthåndterere i dokumentdatabaser.

3.1 Introduksjon

Versjonskontrollsystemer, slik de omtales i denne oppgaven, er programvare som har i oppgave å ta vare på historikk for en eller flere filer. I tillegg tilbyr flere slike systemer muligheten for flere brukere å jobbe på samme prosjekt samtidig og senere samle og synkronisere endringene. Det er spesielt denne siste funksjonaliteten som er interessant i denne sammenhengen.

Selv om det finnes ulike implementasjoner av versjonskontrollsystemer, er de ofte basert på de samme grunnkomponentene. For denne oppgaven er det verktøy for *diff*, *branch*, *merge*, og versjonshistorikk, samt problemet med *merge conflicts* som er interessant. Resten av dette kapittelet ser kort på hvert tema. For en mer detaljert gjennomgang av versjonskontrollsystemer og deres virkemåte anbefales Baudiš [3].

Versjonskontrollsystemer grupperes gjerne inn i en av to grupper; distribuert eller sentralisert, hvor sistnevnte er basert på en sentral server [3]. Hensikten i denne oppgaven er å sammenligne versjonskontrollsystemer mot distribuerte multi-master dokumentdatabaser hvor det ikke er en sentral server. Derfor refererer begrepet *versjonskontrollsystemer* kun til distribuerte versjonskontrollsystemer i denne oppgaven. For å begrense omfanget vil Git [14], et distribuert versjonskontrollsystem, brukes som eksempel der det ikke diskuteres generelle prinsipper.[1]

[1] Git ble valgt fordi forfatteren kjenner det best.

3.2 Diff

En sentral del av versjonskontroll er verktøyet for å sammenligne filer. Dette er bakgrunnen for å kunne finne likheter og forskjeller mellom to filer – gjerne to utgaver av samme fil. For å finne forskjellene mellom filer brukes en delta-algoritme. Resultatet fra algoritmen kan deretter bli presentert for brukeren. Et slikt resultat blir heretter omtalt som en *diff*.

Listing 2 viser et eksempel på hvordan en diff kan se ut. I en slik diff er det informasjon om hvilke filer som er sammenlignet og hvor i filene forskjellene ligger. Ved å gjøre det på denne måten holder det å vise kun de delene av filene hvor det er ulikheter. Deretter kommer sammenligningen av filene. Første tegn på hver linje indikerer kildefil slik det er angitt i starten av diffen[2]. I tillegg til endringene er det noen linjer over og under som gir kontekst for endringene.

```
--- a/src/NHN.DtoContracts/NHN.DtoContracts/Flr/Service/IFlrExportOperations.cs
+++ b/src/NHN.DtoContracts/NHN.DtoContracts/Flr/Service/IFlrExportOperations.cs
@@ -22,6 +22,6 @@ namespace NHN.DtoContracts.Flr.Service
         /// </returns>
         [OperationContract]
         [FaultContract(typeof(GenericFault))]
-        Stream ExportGPDetails(QueryParameters searchParameters);
+        Stream ExportGPContracts(ContractsQueryParameters searchParameters);
     }
 }
```

Listing 2: Eksempel på diff hvor to utgaver av samme fil er sammenlignet. Det er ei linje som forskjellig mellom utgavene. Dette representeres ved at den gamle versjonen av linja er slettet og den nye lagt til. Eksempelet er hentet fra NhnDtoContracts [9].

Det finnes flere forskjellige delta-algoritmer. Baudiš [3] gir en detaljert forklaring av flere slike. I denne oppgaven oppsummeres tre av algoritmene. Det er valgt tre som er vanlig i bruk mot tekstfiler ettersom fokuset for oppgaven er tekstbaserte dokumenter i en dokumentdatabase.

Myers' Longest Common Subsequence
: Basert på å finne lengste felles delsekvens, linje for linje, mellom filene. Dette er standardvalget for `git diff`.

Patience Diff
: Denne er også basert på å finne lengste felles delsekvens, men den kjører først en runde med kun linjer som eksisterer én gang i begge filene. Resultatet utvides deretter med linjene som finnes flere steder i begge filene.

BDiff
: Søker først etter lengste felles delstreng. Deretter gjøres det samme, rekursivt, med det som kommer før og etter.

[2]Her er `-` gammel utgave og `+` er ny utgave. Linjer uten fortegn finnes i begge og brukes for å gi kontekst for endringene.

3.3 Historikk og logg

En annen sentral del av versjonskontroll er versjonshistorikk. Ved å ta vare på hver utgave i historikken kan man gå tilbake til en tidligere versjon og sammenligne ulike versjoner. Brukeren må selv bestemme når en revisjon skal lagres til historikken. I Git gjøres dette ved å opprette en *commit*.

For hver commit finnes det informasjon om hvilken commit som kom før den – foreldrenoden. Delkapitlene 3.4, 3.5, og 3.6 ser nærmere på konsekvensen av å tillate at et commit kan ha flere foreldrenoder og at flere commits kan ha samme foreldrenode.

Muligheten for å visualisere historikken kan være nyttig. Listing 3 viser et eksempel på en logg fra Git som viser grafen for endringer og beskrivelsen av endringene over tid. De første 7 alfanumeriske karakterene etter grafen er de første tegnene i SHA1-IDen til commiten.

```
*   9c88afd Merge pull request #19 from royfatla/UpdatedOn
|\
| * b8b7328 Added UpdatedOn for Data contracts
* | adffdda Add GetGPContractIdsOperatingInPostalCode operation
|/
*   06b6abb Merge pull request #17 from janerist/missingoperation
```

Listing 3: Eksempel på hvordan en logg fra Git kan se ut. Hver commit bygger på enhver commit som ligger under den i treet. Utdrag av historikken fra NhnDtoContracts [9].

3.4 Branch

Et annet nyttig verktøy i forbindelse med versjonskontroll er muligheten for at flere utgaver av samme fil kan eksistere samtidig. En måte å gjøre dette på er å opprette en forgrening i historikken – en *branch*. Deretter kan man bytte mellom forskjellige branches og endringene som er gjort i en branch vil ikke bli speilet i en annen med mindre det gjøres noe aktivt for å synkronisere dem. I distribuert versjonskontroll kan det også eksistere flere versjoner av samme fil uten å spesifikt opprette en ny branch fordi alle systemene som har fila ikke er synkronisert til enhver tid.

I eksempelet på en logg i Listing 3 er det en forgrening i loggen fordi det er opprettet en branch. Her har både `b8b7328` og `adffdda` samme foreldrenode: `06b6abb`. Mer om hvordan branching kan brukes i praksis er beskrevet av Driessen [15].

3.5 Merge

Muligheten til at flere utgaver av samme dokument eksisterer samtidig medfører et behov for å kunne synkronisere versjoner og slå dem sammen til ett dokument. For å flette sammen flere utgaver utføres en *merge*. Det er dette som er gjort i commiten `9c88afd`

i Listing 3. To branches er merget til én og det er opprettet en commit som har to foreldrenoder.

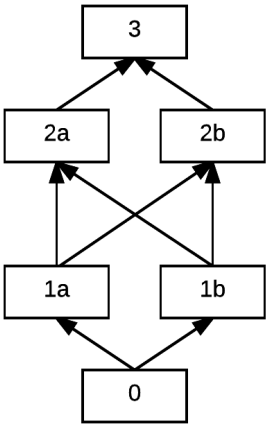
Det finnes flere forskjellige strategier for å utføre en merge. Nedenfor oppsummeres fire strategier. Baudiš [3] gir en dypere forklaring av *Fast-forward merge*, *three-way merge*, og *recursive merge*. Mens [4] forklarer og sammenligner *two-way merge* og *three-way merge*.

Fast-forward Merge
Dette er i praksis kun en oppdatering. En eldre versjon blir oppdatert slik at den blir lik den nye og får samme historikk som den nye. Dette krever at den eldre versjonen er en del av historikken til den nye.

Two-way Merge
To filer sammenlignes med hverandre. Tar ikke hensyn til om det finnes en felles forgjenger – en *common ancestor*.

Three-way Merge
I tillegg til å sammenligne to utgaver, slik som two-way merge, sammenligner three-way merge også begge utgaver med en common ancestor slik den vet hvilke endringer hver utgave har gjort.

Recursive Merge
I noen tilfeller vil det ikke være trivielt å finne en felles forgjenger. Et eksempel på dette er *criss-cross merge*-eksempelet vist i Figur 3.1. Her kan en rekursiv tilnærming av three-way merge brukes.



Figur 3.1: Criss-cross merge problemet. Både `1a` og `1b` er en felles forgjenger for `2a` og `2b`. Dette er problematisk fordi resultatet i `3` kan bli ulikt avhengig av hvilken som blir valgt og hvordan merge ble håndtert i `2a` og `2b`.

3.5.1 Argumentet for Three-way merge

Two-way merge har ikke tilstrekkelig informasjon for å kunne gjøre merge riktig i flere tilfeller. I Figur 3.1 vil dette kunne bli et problem når `1a` og `1b` skal merges til `2a` eller `2b`. Årsaken til at dette er et problem er at two-way merge ikke klarer å skille mellom hva

som er nytt, slettet, og modifisert – og om det er gjort i kun én fil eller i flere samtidig. Fordi Three-way merge kan sammenligne hver utgave med en common ancestor har den ikke dette problemet [4].

3.6 Merge conflict

I situasjoner hvor versjonskontrollsystemet ikke kan garantere at en merge blir løst riktig oppstår en *merge conflict*. Dette er en konflikt som oppstår fordi det er gjort endringer på samme sted i samme fil i begge utgavene. Dersom det oppstår konflikter under en merge må alle løses av brukeren før merge kan fullføres.

Dette er et problem som er vanskelig å løse automatisk fordi det ikke er opplagt hva som er riktig resultat. Det er en risiko for at kode kan gå tapt eller at det oppstår syntaktisk eller semantiske feil dersom konflikten forsøkes løst av et automatisk system.

Terry [4] ser på andre metoder for merge, blant annet syntaktisk og semantisk. Slike verktøy for merge kan finne andre typer konflikter ettersom de tar hensyn til strukturen og innholdet i filene.

Del II

Analyse / Evaluering

Kapittel 4

Testmiljø

For å lage et testmiljø som kan simulere problemet er det utviklet programvare for klient og server. Dette kapittelet ser på virkemåten til denne programvaren og hvilke gevinster implementasjonen gav.

4.1 Funksjonalitet

Testmiljøet består av en klient og to servere hvor følgende funksjonalitet er implementert:

Skriv data
: Klienten sender et dokument med en gitt ID til en server som deretter lagrer det i sin database. Dersom replikering er slått på vil dokumentet automatisk bli replikert til den andre serverens database.

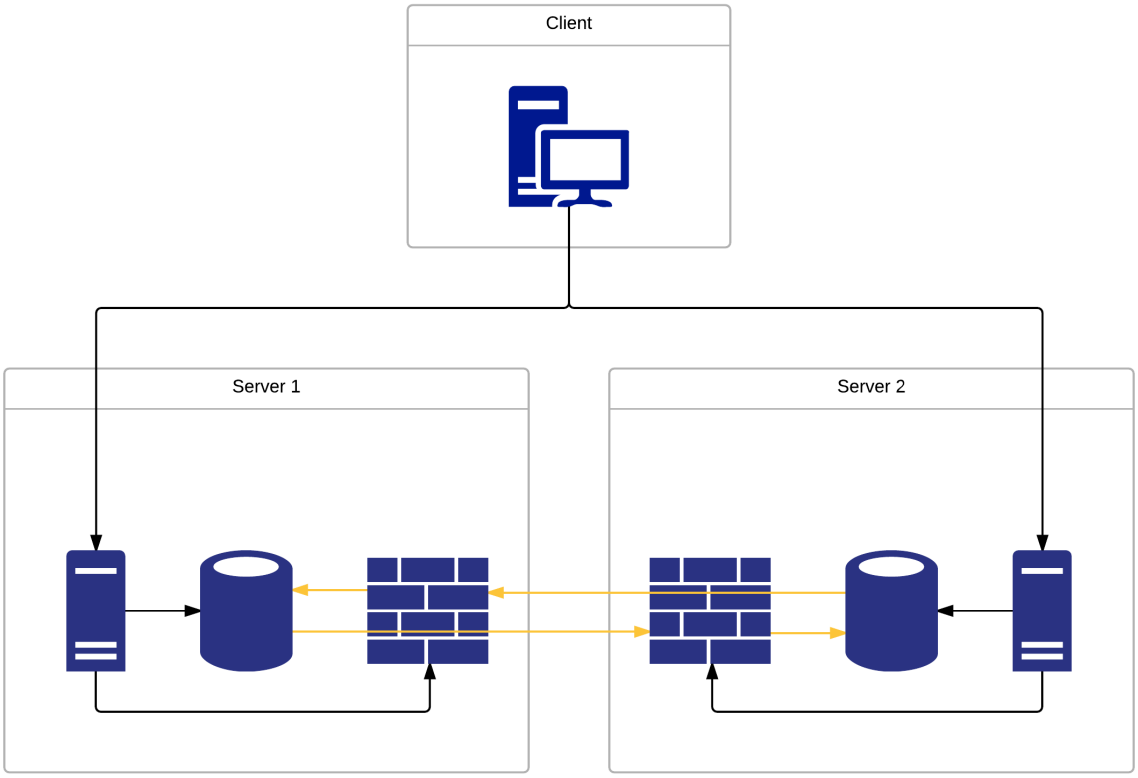
Les data
: Klienten ber en server om å få et dokument med en gitt ID. Dersom det er en konflikt på dokumentet vil denne bli forsøkt løst før dokumentet sendes.

Start replikering til *server*
: Klienten gir en server nødvendig kontaktinformasjon slik at serveren heretter kan replikere dokumenter til den andre serveren.

Blokker replikering
: Klienten ber serveren om å slutte å motta replikeringsdata. Dette gjøres ved at brannmuren på serveren ikke slipper slik data gjennom.

Åpne for replikering
: Klienten ber serveren om å godta replikeringsdata. Dette gjøres ved at brannmuren på serveren slipper gjennom slike data.

Nettverksdiagrammet i Figur 4.1 viser hvordan testmiljøet er strukturert. Klientprogrammet har en kobling til begge serverprogrammene. Disse skriver data til hver sin database og styrer en lokal brannmur med regler for blokkering av replikeringsdata.



Figur 4.1: Nettverksdiagram for et testmiljø med en klient og to servere. Serverikonet representerer i denne sammenhengen serverprogramvaren. Serverprogrammet leser og skriver data til en RavenDB database. I tillegg styrer serverprogrammet en brannmur som kan stoppe innkommende replikeringsdata.

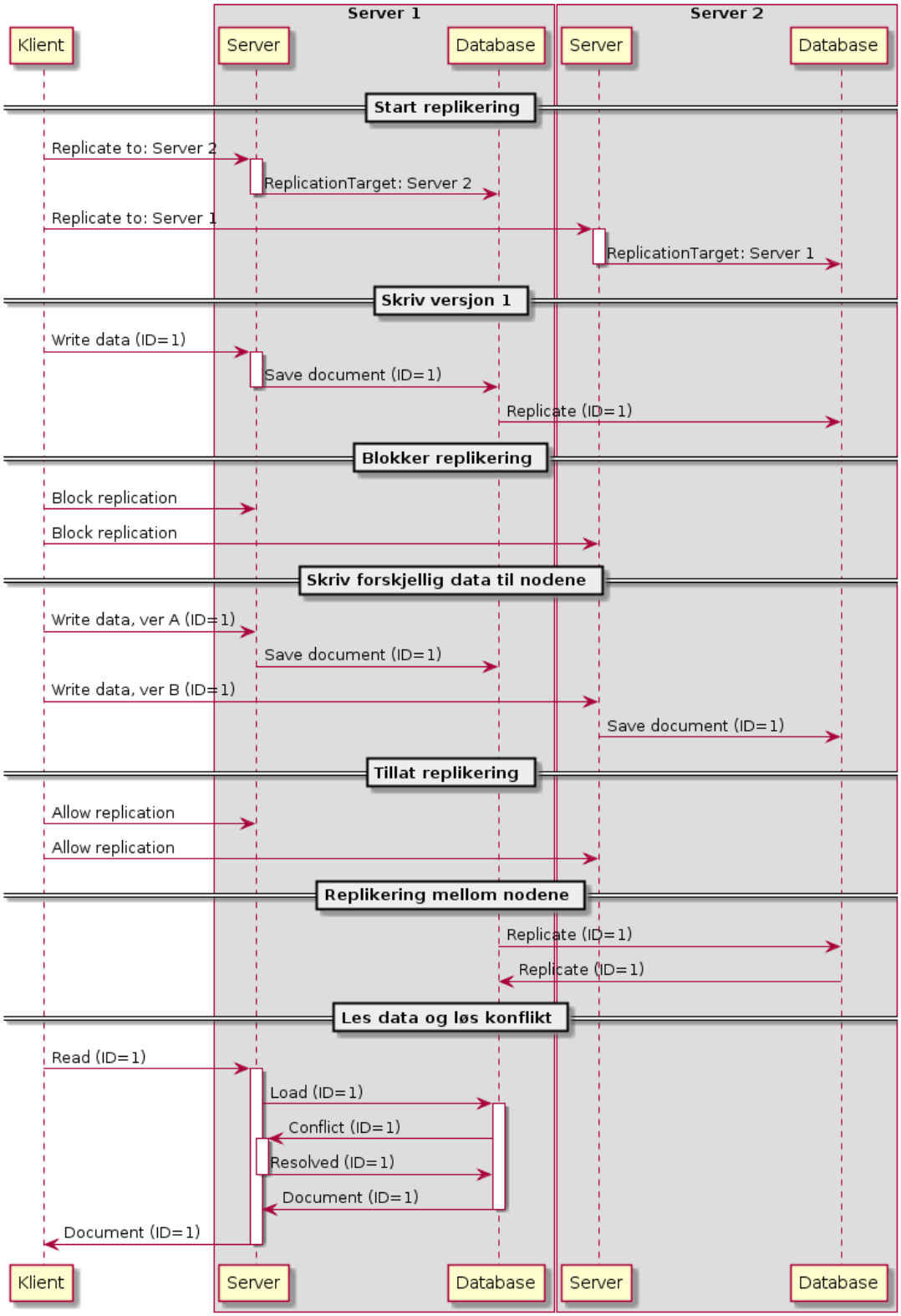
4.2 Programflyt i testmiljøet

I dette testmiljøet er det mulig å tvinge frem konflikter for replikerte Raven databaser. Sekvensdiagrammet i Figur 4.2 viser en programflyt som tvinger frem en slik konflikt.

Først settes miljøet opp slik at det alltid har samme utgangspunkt uavhengig av om programmet har kjørt før eller ikke. Klienten starter med å gi begge serverne beskjed om at Raven skal replikere til den andre serveren. Et dokument med ID=1 skrives til Server 1 og blir deretter replikert til databasen på Server 2. Dette gjøres slik at begge serverne i utgangspunktet har samme utgave av dokumentet. Klienten ber så begge serverne om å blokkere replikering.

Neste steg er å oppdatere dokumentet med ID=1 med forskjellig data til hver server. Ved å sende forskjellig oppdatering til serverne uten at de kan kommunisere seg i mellom vil dette forårsake en konflikt neste gang de kan kommunisere. Etter at begge serverne har fått sin oppdatering åpnes brannmurene og begge sender sin utgave av dokumentet til den andre serveren. Nå finnes begge versjonene i begge serverne, men ingen av dem vet hva som er riktig.

Til slutt ber klienten Server 1 om å lese dokumentet med ID=1. Databasen oppdager at



Figur 4.2: Sekvensdiagram for en programflyt som forårsaker konflikt

dokumentet har en konflikt og ber serveren løse den. Etter at serveren har løst konflikten blir dokumentet sendt fra databasen til serveren som sender det videre til klienten.

Kapittel 5 og Vedlegg B viser forskjellige metoder for å løse slike konflikter i Raven.

4.3 Resultater fra testmiljøet

I tillegg til å gi innsikt i hvordan RavenDB implementerer replikering, har implementasjon og testing av testmiljøet hjulpet med å identifisere et par potensielle problemtilfeller. Resten av delkapittelet ser nærmere på resultatene fra testmiljøet.

4.3.1 Implementasjon i Raven

Testmiljøet gav en innsikt i hvordan replikering og konflikthåndtering fungerer i RavenDB. Hvilke alternativer som er bygd inn i Raven og hvordan applikasjonstilpasssede konflikthåndterere implementeres mot Raven diskuteres ikke nærmere her, da dette er temaet i Kapittel 5. I tillegg viser Vedlegg B eksempler på hvordan en tilpasset løsning kan programmeres mot RavenDB.

4.3.2 Historikk

Som nevnt i Delkapittel 2.4.3, kan Raven lagre historikk for dokumenter. Dette øker likhetstrekkene mellom NoSQL problemet og merge-conflict-problemet i versjonskontroll. Samtidig introduserer dette også et interessant problem i forbindelse med replikering. På samme måte som det kan oppstå replikeringskonflikter for vanlige dokumenter kan det også oppstå konflikter på historiske dokumenter. Dersom historikken skal brukes vil dette innebære at konfliktene i historikken også må håndteres.[1]

4.3.3 Annen server, annen løsning

Ved å blokkere replikering mellom servere etter at konflikten er replikert, men før den er løst, oppstår et annet problem. Dersom begge nodene, som nå er isolert fra hverandre, løser konflikten er det mulig at de løser den på forskjellig vis. Dermed vil konflikten oppstå på nytt med de to forskjellige løsningene når det åpnes for replikering igjen. Dette kan unngås ved å bruke en konfliktløser som løser konflikter på samme måte uavhengig av hvilken server den er på.

4.3.4 Datatyper

Hvordan en konflikt løses kan være avhengig av hvordan data er representert (dataformat) og hvilken informasjon som er representert (informasjonstype). Noen typer informasjon kan kreve ekstra hensyn i hvordan de håndteres i forbindelse med konflikthåndtering. Dette kan derfor være nyttig å ta hensyn til før implementasjon av en konflikthåndterer. For eksempel skal ei sortert liste fortsatt være sortert etter konflikthåndtering.

[1] Dette gjelder for versjon 3.5. Implementasjonen av versjonering i RavenDB endres i versjon 4.0 [13].

Kapittel 5

Konfliktløsing i RavenDB

RavenDB tilbyr tre forskjellige typer konfliktløsing. Disse krever forskjellig grad av menneskelig involvering og utviklingstid. Den første er en simpel, innebygd automatisk løsning; den andre er et innebygd webgrensesnitt for manuell håndtering; den siste er muligheten for å programmere en automatisk (eller manuell) modell selv.

5.1 Innebygd automatisk

Den første typen konfliktløsing tilbudt i RavenDB er en simpel, automatisk løsning. I denne løsningen kan man velge en av tre strategier:

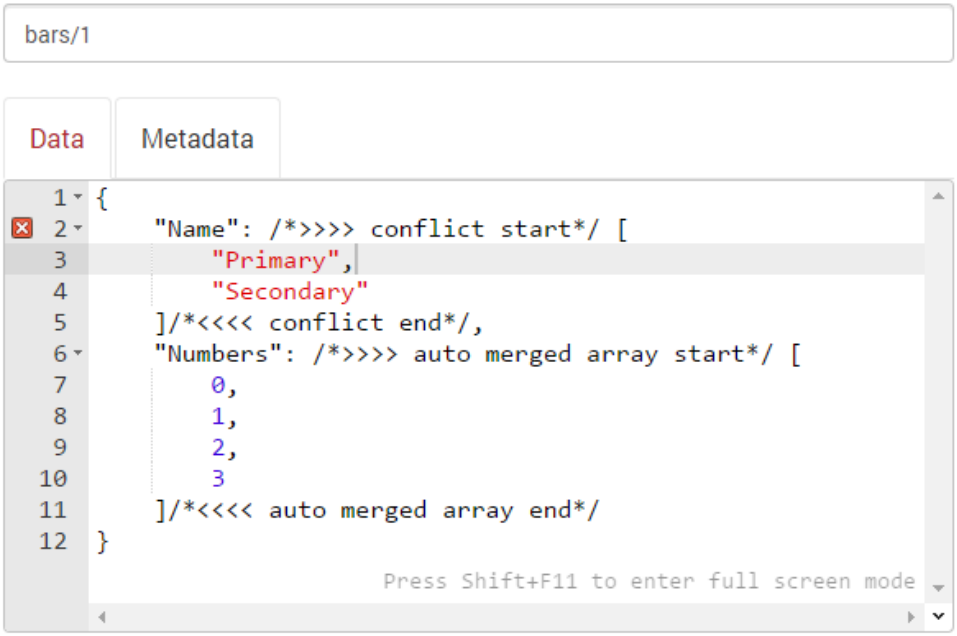
1. Velg utgaven fra **lokal** server
2. Velg utgaven fra **ekstern** server
3. Velg utgaven som ble **endret sist**

Etter at en av disse reglene er valgt kreves ikke mer menneskelig involvering med mindre det skal velges en annen strategi. Dette krever heller ikke ekstra tid til utvikling. Ulempen med denne løsningen er at man har ingen kontroll over resultatet utover den simple regelen som er valgt.

5.2 Raven Studio

RavenDB tilbyr også en løsning for å løse konflikter manuelt gjennom et webgrensesnitt i *Raven Studio*, vist i Figur 5.1. Her får man presentert et tekstområde med dokumentet, samt markører som viser hvor i dokumentet det har oppstått konflikter. For å løse konflikten redigeres dokumentet slik at data i dokumentet blir riktig. Deretter slettes konfliktmarkørene før dokumentet lagres og konflikten er løst.

Dette er en metode som ikke krever ekstra utviklingstid, men det krever stor grad av menneskelig involvering for hver konflikt som oppstår. Fordelen er at man får god kontroll på hvordan konflikter løses.



Figur 5.1: Grensesnittet for konfliktløsning i Raven Studio. Her vises data fra begge kildene og Raven har gjort et forsøk på å slå sammen to lister med forskjellig data.

5.3 Egenprodusert

Raven tilbyr også et grensesnitt for konfliktløsning gjennom egenprodusert kode. På denne måten kan man enten lage et eget grensesnitt for manuell håndtering av konflikter, eller lage egenprodusert automatisk konfliktløsing. Ved å utvikle en egen automatisk løsning har man muligheten til å lage en smartere konfliktløser basert på egne datatyper og logikk uten menneskelig involvering. En slik automatisk løsning har gjerne litt mindre kontroll enn i et fullstendig manuelt miljø. Dersom det er behov for større kontroll kan det utvikles en løsning som tillater manuell konflikthåndtering i stedet for eller i tillegg til automatisk håndtering.

Vedlegg B ser nærmere på hvordan man kan lage forskjellige typer automatiske konflikthåndterere gjennom det innebygde grensesnittet for konflikthåndtering i Raven.

Kapittel 6

Versjonskontroll og dokumentdatabaser

Flere av komponentene fra versjonskontrollsystemer, beskrevet i Kapittel 3 løser problemer som ligner på problemene som oppstår i forbindelse med replikerte dokumentdatabaser. Dette kapittelet ser på hvilke lærdommer fra versjonskontrollsystemer som kan være nyttige i forbindelse med replikerte dokumentdatabaser. Hvilke likhetstrekk finnes; på hvilken måte er de forskjellige; og hvilke problemer som oppstår i forbindelse med å implementere idéene fra versjonskontroll i dokumentdatabaser.

6.1 Sammenligning av systemer

Argumentet for at replikerte dokumentdatabaser kan sammenlignes med versjonskontrollsystemer er som følger:

Tekstformat og datastruktur
: I begge tilfeller lagres data i et tekstformat hvor strukturen i tekstformatet er viktig. Versjonskontroll knyttes som regel mot håndtering av programkode, noe som følger et gitt programmeringsspråk. Dokumentdatabaser bruker gjerne et format som Json. Dersom strukturen eller rekkefølgen på linjer brytes er det en høy risiko for at data eller kode blir korrupt.

Branching og isolerte noder
: Flere versjoner av samme data kan eksistere samtidig. I versjonskontroll kan dette skje dersom det opprettes branches og i distribuerte systemer. I dokumentdatabaser skjer dette når en node isoleres og skriveoperasjoner er tillatt også på isolerte noder.

Diff – Finne forskjeller
: Når forskjellige versjoner skal synkroniseres og slås sammen er det et behov for å finne ut hva som er forskjellen mellom dem.

Merge
: Når forskjellene er funnet er neste steg å flette sammen utgavene, utføre merge, slik at endringene fra hver utgave reflekteres riktig i den samlede utgaven. Det er

ønskelig med en løsning som gjør dette automatisk uten å introdusere feil for å minske involvering av brukere.

Merge conflicts
Noen ganger kan ikke automatisk merge gjøres på en måte som garanterer at resultatet er uten feil.

Selv om det finnes likheter mellom de to systemene er det også noen forskjeller mellom dem:

Datatype
Typen informasjon som lagres er som regel forskjellig. Versjonskontrollsystemer forbindes oftest med kode, mens dokumentdatabaser brukes til å lagre data.

Historikk
Historikk er en sentral del av versjonskontroll, men det er i utgangspunktet ikke brukt i RavenDB. En form for historikk kan aktiveres med Versioning pakken i Raven, men dette fungerer på en annen måte enn historikken i Git. Se 2.4.3 for mer om versjonering i Raven. For mer om utfordringen rundt historikk i Raven, se 4.3.2.

Brukerinnvolvering ved konflikthåndtering
Brukeren er mer tilgjengelig i et versjonskontrollsystem. Ved bruk av versjonskontroll har en utvikler tilgang til filer og program direkte. Dokumenter og metoder mot dokumentdatabaser er ofte gjemt bak et tilpasset brukergrensesnitt og programlogikk.

Tidspunkt for konflikthåndtering
I dokumentdatabasen RavenDB håndteres ikke konflikten før neste leseoperasjon til dokumentet med konflikt. Versjonskontrollsystemet Git ber brukeren håndtere konflikten når konflikten oppdages.

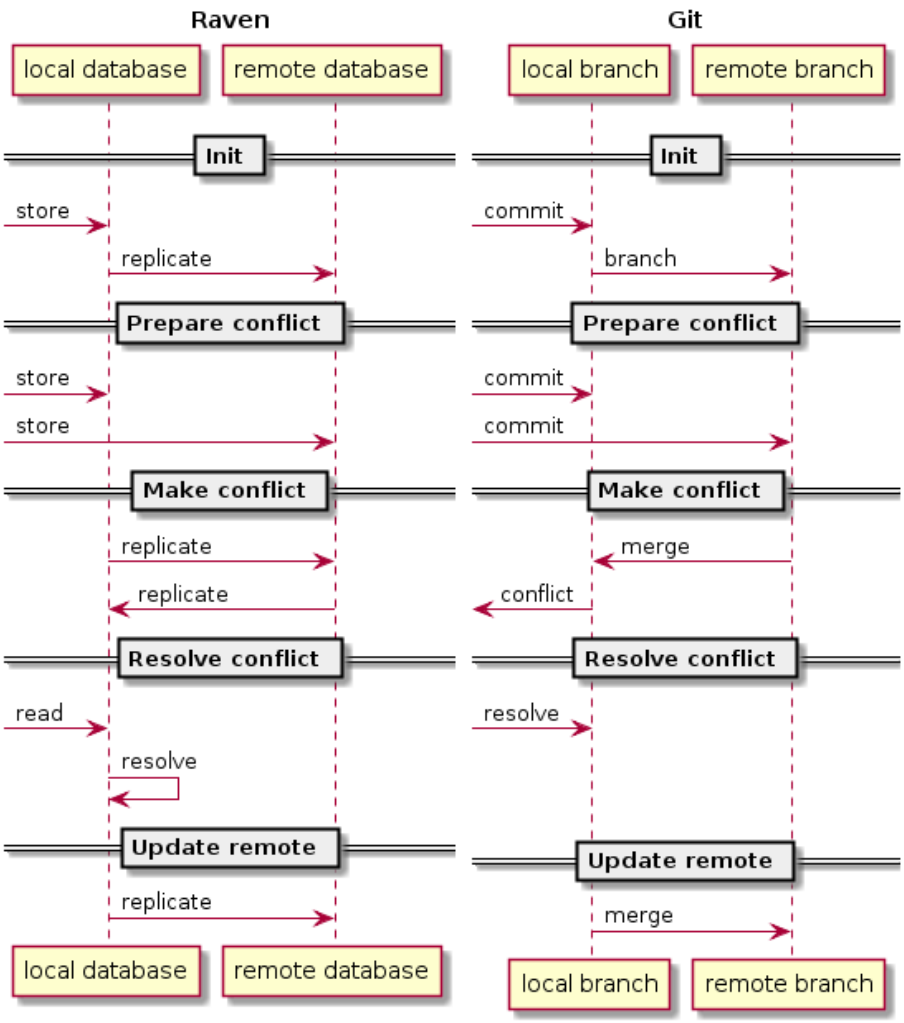
6.2 Sammenligning av programflyt

Dette delkapittelet ser på likheter og ulikheter mellom programflyten som leder til en replikeringskonflikt og arbeidsflyten som leder til en merge conflict. Sekvensdiagrammet i Figur 6.1 brukes som bakgrunn for sammenligningen.

6.2.1 Init

Den første fasen, *Init*, er med for å gi kontekst for utgangspunktet i begge situasjoner. Ved slutten av fasen regnes begge systemer å være i god helsemessig stand. For å gjøre eksempelet enkelt finnes kun én utgave i forkant av neste fase her. Det kunne ha vært en lengre historikk i forkant av fasen så lenge lokal og ekstern versjon er like og har samme historikk.

Init for Raven er her gjort ved først å opprette to databaser. En på en lokal server og en på en ekstern server. Begge databasene er konfigurert til å replikere til den andre. Deretter lagres et dokument i den lokale databasen. Når dokumentet er replikert til den eksterne databasen er init-fasen ferdig.



Figur 6.1: Sammenligning av programflyten som leder til en replikeringskonflikt (venstre) og arbeidsflyten som leder til en merge conflict (høyre).

Init for Git er her gjort ved å opprette en ny Git repository. Det er opprettet en commit med ei fil. Deretter er det opprettet en forgrening ved å publisere repository til en ekstern server. En alternativ måte å oppnå en tilsvarende situasjon er å opprette en branch fra samme commit lokalt på samme maskin.

Fremgangsmåten er forskjellig i Raven og Git, men resultatet er tilsvarende: det finnes én utgave og den eksisterer både lokalt og eksternt.

6.2.2 Prepare conflict

I den andre fasen, *Prepare conflict*, legges grunnlaget for konflikten. Det gjøres en samtidig endring lokalt og eksternt. Endringene som gjøres er ikke like og dermed vil det oppstå en konflikt når de to skal synkroniseres.

Fordi Raven i utgangspunktet vil replikere endringer automatisk mellom de to databasene blir det her antatt at forbindelsen mellom de to er brutt i en periode.

6.2.3 Make conflict

Det er i tredje fase konflikten oppstår. Dette er punktet hvor lokalt og ekstern blir synkronisert. Her er også flere av forskjellene mellom Git og Raven.

Det er brukeren selv som må starte synkronisering i Git, mens i Raven vil synkroniseringen skje automatisk.

Raven vil synkronisere begge veier – lokal til ekstern og ekstern til lokal – mens i Git synkroniseres kun en vei om gangen. Det er mulig å manuelt gjøre en ekstra synkronisering i Git slik at det vil oppføre seg som om det ble synkronisert begge veier samtidig. Da oppstår samme problemstilling som i Raven dersom løsningene av konfliktene er forskjellig.

I Git vil brukeren umiddelbart få beskjed om at en konflikt har oppstått, som vist i Listing 4. Områdene hvor det har oppstått konflikt vil bli markert og begge utgaver er inkludert i fila. Konflikten må løses før synkroniseringen kan fullføres. Situasjonen er en annen i Raven. Der skjer synkroniseringen automatisk når kontakten gjenopprettes mellom lokal og ekstern database, gjerne lenge etter at brukeren lagret dokumentet. Dermed ligger begge versjonene på begge steder. Raven har på dette punktet oppdaget konflikten, men vil ikke gjøre et forsøk på å håndtere den før i neste fase.

6.2.4 Resolve conflict

I fjerde fase løses konflikten. Her er den største forskjellen mellom Raven og Git.

Git Som nevnt tidligere vil Git gi beskjed om konflikten med en gang den oppstår og vil ikke fullføre merge før den er løst. Brukeren kan løse konflikten enten ved å manuelt redigere fila eller ved å bruke et verktøy. Her kan brukeren velge å ta vare på endringer fra begge utgaver eller velge bare én. I tillegg kan brukeren velge å gjøre endringer som involverer mer enn bare området hvor det var en konflikt. Deretter opprettes en commit og konflikten er regnet som løst.

I Git finnes detaljert endringshistorikk fra begge sider, inkludert hvem som gjorde hva og når det ble gjort. Dersom commit-meldinger, beskrivelser av arbeidet i hver commit, er brukt på en god måte finnes også forklaring på hva som ble gjort og hvorfor. Dette gir den som skal løse konflikten tilgang til informasjon om endringene som ble gjort – og dersom det ikke er nok vet vedkommende hvem som var involvert i arbeidet som førte til at problemet oppsto og kan kontakte dem. Fordi konflikten oppstår med en gang er det også mulig at endringene og jobben som ble gjort fortsatt er fersk i minnet for involverte parter.

Raven I Raven er situasjonen en annen. Her blir ikke konflikten forsøkt løst før ved neste leseoperasjon. Dermed kan en betydelig mengde tid ha passert fra konflikten oppsto til den blir forsøkt løst. Dette kommer i tillegg til tiden det tok fra dokumentet ble skrevet til replikering hvor konflikten oppsto. Det er mulig at det er en tredjepart som utfører leseoperasjonen. Dersom konflikten i denne situasjonen skal løses manuelt kan det medføre at en som er ukjent med endringene kan få i oppgave å løse den.

I utgangspunktet har ikke Raven historikk. Dermed er diff og merge begrenset til toveismetoder som ikke er like kraftfulle som treveismetoder. Det er mulig å introdusere historikk i Raven, men det har egne utfordringer beskrevet i Delkapittel 4.3.2. Historikken i Raven inneholder ikke informasjon om hvem som gjorde endringen og hvorfor, slik historikken i Git har.

6.2.5 Update remote

Til slutt skal det eksterne systemet oppdateres slik at resultatet etter at konflikten er løst også finnes der. Hvis alt går vel vil ikke dette forårsake problemer og begge systemer vil da være i god helsemessig stand. Dersom det er gjort endringer på det eksterne systemet er det en risiko for at det oppstår en ny konflikt som må løses.

6.3 Oppsummering

Det finnes likheter og ulikheter mellom komponenter og programflyt i de to systemene. Begge brukes primært mot tekst som er i et strukturert format og de har sammenlignbare komponenter og programflyt.

De to største forskjellene i programflyt mellom systemene er i fasene `Make conflict` og `Resolve conflict`. Dette er knyttet til når konflikter oppdages og når den må løses. Git oppdager konflikten før synkroniseringen er ferdig og vil ikke la synkroniseringen fullføre før den er løst. Raven vil ikke oppdage konflikten før det gjøres en leseoperasjon mot dokumentet med konflikten. Dermed er det først når dokumentet leses at konflikten kan løses. I et system hvor Raven ber brukeren løse konflikten kan dette medføre at en bruker som ikke var involvert i å lage konflikten blir bedt om å løse den.

Kapittel 7

Praktiske hensyn

Dette kapittelet ser på hvordan en løsning for konflikthåndtering basert på metoder fra versjonskontrollsystemer kan fungere i praksis for master-master dokumentdatabaser.

7.1 Diff

Som nevnt i Delkapittel 2.2, bruker RavenDB tekstformatet Json til å lagre data. Dermed kan endringer i data representeres ved at linjer legges til, slettes, eller endres. For å finne forskjellene, diff, mellom ulike utgaver i en dokumentdatabase kan man derfor benytte seg av de samme metodene som versjonskontroll, beskrevet i Delkapittel 3.2.

Ettersom det i utgangspunktet ikke finnes historikk i en RavenDB dokumentdatabase er det ikke mulig å bruke en treveis diff. Som et resultat av dette er man begrenset til en toveis diff. Av dette følger det samme dilemmaet beskrevet i Delkapittel 3.5.1 – det er umulig å vite hva hver enkelt versjon endret.

Man kan bruke treveis diff ved å benytte `Versioning` pakken i Raven som introduserer historikk for data. Da blir det også nødvendig å finne en løsning på problemet med at historikken også får konflikter i RavenDB, slik beskrevet i Delkapittel 4.3.2. Et annet alternativ er å lage en tilpasset implementasjon av dokumenthistorikk.

7.2 Merge

Ved å ta vare på historikk for dokumentene og bruke samme form for diff åpnes det for bruk av samme metode for merge som brukes i versjonskontrollsystemer. I både dokumentdatabaser og i versjonskontrollsystemer som holder kode er det viktig å ikke ende opp med korrupt data. Dette innebærer at man ikke bryter syntaksreglene som gjelder for formatet informasjonene lagret i eller språket koden er skrevet i. Ved å bruke tekstbasert merge ved en replikeringskonflikt vil systemet ha samme evne til å ta ta vare på syntaks som versjonskontroll vil ha for samme type dokument.

7.3 Merge conflicts

Den største forskjellen mellom dokumentdatabaser og versjonskontrollsystemer kan ligge rundt håndtering av merge conflicts. I et versjonskontrollsystem får man umiddelbart beskjed om at en merge conflict har oppstått, som vist Listing 4. Git vil ikke la brukeren fullføre merge operasjonen før konflikten er markert som løst ved å opprette en commit.

```
$ git merge other-branch
Auto-merging Business.cs
CONFLICT (content): Merge conflict in Business.cs
Automatic merge failed; fix conflicts and then commit the result.
```

Listing 4: Eksempel på melding etter forsøkt merge som resulterer i merge conflict.

Her oppstår problemet for dokumentdatabaser. Det er ingen garanti for hvor lang tid det tar før en isolert node får kontakt med andre noder. Dermed oppstår konflikten mye senere, potensielt på et tidspunkt hvor ingen av brukerne som var involvert i skriveoperasjonene som laget konflikten er tilgjengelig.

Dette er en av faktorene som motiverer automatisk konflikthåndtering, slik at en tredjepart ikke blir gjort ansvarlig for å håndtere konflikten. Dersom det er oppstått mange konflikter vil det også medføre mye ekstra arbeid for brukere, noe som også motiverer automatisk konflikthåndtering. En annen faktor er at brukeren vil ikke ha direkte tilgang til dokumentdatabasen, da den ligger bak en applikasjon. Eventuelt kan manuell konflikthåndtering gjøres gjennom applikasjonen, men det krever at applikasjonen tilpasses dette. Et alternativ er en kombinert løsning som utfører merge, men involverer brukeren til å håndtere konflikter. Avhengig av ønsket oppførsel kan brukeren også involveres for å verifisere at resultatet av en automatisk merge ble riktig.

Et spesielt tilfelle i Raven er at konflikten eksisterer både lokalt og eksternt. Dette kan resultere i problemet beskrevet i Delkapittel 4.3.3. Dersom konflikten løses på begge steder i et tidsrom hvor databasene igjen er isolert fra hverandre er det en risiko for at konflikten løses forskjellig. Dette vil medføre at en ny konflikt når kontakten mellom databasene gjenopprettes.

7.4 Merge eksempler

Dette delkapittelet gir eksempler på forventet oppførsel ved håndtering av replikeringskonflikt basert på metoder fra versjonskontrollsystemer. Alle eksemplene bruker Jsondokumentet i Listing 1 som utgangspunkt og antar at det finnes historikk for dokumentene. Det er to parter som gjør hver sin endring på dokumentet mot to forskjellige databaser i et tidsrom hvor de to databasene er isolert fra hverandre.

Forskjellige felter

I dette eksempelet gjør partene endringer på forskjellige deler av dokumentet. Den ene endrer `OrganizationNumber` og den andre parten endrer `Name`. Når databasene senere synkroniseres vil konflikthåndtereren kunne bruke begge endringene uten problemer fordi det ikke er overlapp mellom endringene.

Samme felt, lik verdi

I situasjonen hvor begge parter har endret samme felt til samme verdi vil det ikke oppstå konflikt i sammenligningen og problemet er løst.

Samme felt, ulik verdi

I dette eksempelet gjør begge partene en endring på samme felt i dokumentet. Begge endrer `Name`, men ikke til samme verdi. Her er det overlapp i endringene og dermed vil *merge* resultere i en *merge conflict* og konflikten er ikke løst automatisk.

LastChanged

Eksemplene over ignorerer feltet `LastChanged` – tidspunktet for siste endring av dokumentet. Dette feltet introduserer et interessant problem. Fordi dette feltet representerer tidspunktet da den siste endringen ble gjort skal det alltid bli oppdatert i alle utgaver. Det vil medføre overlapp i endringene.

For et slikt felt foreslås det to forskjellige automatiske løsninger som kan velges avhengig av ønsket oppførsel. Den ene løsningen er å velge det seneste tidspunktet, den andre løsningen som foreslås er å sette denne verdien til tidspunktet konflikten ble løst. Begge disse løsningene avhenger av en semantisk forståelse av innholdet i feltet og en evne til å ta valg basert på den forståelsen, noe en ren tekstbasert merge ikke har. Dermed er ikke dette forventet oppførsel fra en tekstbasert merge og det er et eksempel på et område hvor en slik løsning ikke kan garantere riktig resultat.

Del III

Diskusjon og Konklusjon

Kapittel 8

Diskusjon

Denne oppgaven har sett på replikeringskonflikter i dokumentdatabaser og har i den forbindelse sett på versjonskontrollsystemer. Målet har vært å finne ut i hvilken grad metoder derfra kan brukes til konflikthåndtering i dokumentdatabaser.

Metoder fra versjonskontroll ser ut til å ha en nytteverdi. Dette er gitt at det finnes historikk for dokumentene slik at *three-way merge* kan brukes. Metodene fra versjonskontroll er ikke en fullgod løsning for å løse problemet automatisk. Det finnes situasjoner hvor versjonskontrollsystemer vil kreve brukerinvolvering for å håndtere konflikter og en tilsvarende løsning for replikeringskonflikter vil ha samme problem. Problemet ligger spesielt i dokumenter hvor forskjellige utgaver har overlappende endringer.

RavenDB lagrer dokumenter i tekstformatet Json og tilbyr muligheten for å lese dette formatet i programkode. Metodikker fra versjonskontrollsystemer kan brukes mot slike Json-dokumenter.

Oppgaven beskriver en programflyt for å gjenskape problemet på en pålitelig måte. Den sammenligner også program- og arbeidsflyt for konflikter i dokumentdatabaser og versjonskontrollsystemer, men har ikke implementert og testet hvorvidt det vil fungere i praksis. Den største forskjellen mellom de to systemene ligger rundt det kritiske punktet – håndtering av konflikter – som byr på utfordringer i sammenligningen.

Oppgaven refererer til tidligere forskning rundt dokumentdatabaser og versjonskontrollsystemer separat ettersom forfatteren ikke er kjent med forskning som knytter disse sammen. Baudiš [3] sammenligner ulike tekstbaserte metoder i versjonskontrollsystemer. Mens [4] beskriver ulike måter for å finne og løse konflikter i versjonskontroll. Dette inkluderer syntaktiske og semantiske metoder som kan oppdage og håndtere konflikter som tekstbasert versjonskontroll ikke kan. Terry [2] foreslår en annen løsning for konflikthåndtering for datalagring hvor hver skriveoperasjon har en egen tilpasset sjekk for konflikt og metode for konflikthåndtering, men også der vil det i noen tilfeller være nødvendig med menneskelig input. Det er spesielt interessant å nevne at denne også ser på konflikter som kan oppstå på tvers av databaseinnslag, noe som ikke kan oppdages med for eksempel tekstbasert diff.

Kapittel 9

Konklusjon

Metoder fra versjonskontrollsystemer kan være nyttig i forbindelse med konflikthåndtering i tekstbaserte dokumentdatabaser dersom det finnes historikk. Delta-algoritmer kan være nyttig for å finne diff – forskjeller mellom dokumenter. Merge kan deretter brukes til å slå sammen ulike versjoner til ett resulterende dokument. Dersom det er overlapp i endringer vil ikke en tekstbasert merge-operasjon kunne løse denne automatisk da det vil oppstå en merge conflict. Dersom endringer i dokumentene ikke overlapper vil dette løses automatisk. Det er likevel en risiko for at det kan oppstå en datakonflikt i resultatet ettersom tekstbasert merge ikke sjekker informasjonen for feil internt i dokumentet eller på tvers av dokumenter.

Det kan også være ulike hensyn som må tas avhengig av hvilke data som lagres i dokumentene, som nevnt i Delkapittel 4.3.4. En ren tekstbasert løsning kan ha problemer med å løse konflikter riktig i slike situasjoner.

Kapittel 10

Videre arbeid

Denne oppgaven påstår at tekstbaserte metoder fra versjonskontrollsystemer kan ha en nytteverdi i dokumentdatabaser basert på en sammenligning av funksjonalitet og programflyt. Videre arbeid vil innebære implementasjon av slike metoder mot en dokumentdatabase for å teste et slikt system i praksis.

Denne oppgaven har fokusert på tekstbaserte metoder fra systemer for versjonskontroll. Det vil også være interessant å undersøke hvorvidt metoder for *syntactic*- og *semantic* merge, som beskrevet i Mens [4], kan fungere bedre enn tekstbaserte metoder. Mens [4] diskuterer også i hvilken grad merge kan automatiseres i versjonskontrollsystemer. Kombinerte systemer med ulike grader av automasjon og brukerinvolvering kan også være interessant i forbindelse med dokumentdatabaser.

Ettersom RavenDB har innebygd deteksjon av replikeringskonflikter har ikke dette vært et fokus i denne oppgaven. Hvordan dette kan gjøres på en effektiv måte, i samarbeid med metodene for håndtering av replikeringskonflikter beskrevet i denne oppgaven, er et annet mulig tema for videre studie.

I denne oppgaven er det gjort en antagelse om at det er et behov for applikasjonstilpasset konflikthåndtering. Det vil være interessant å undersøke i hvilke situasjoner det er nødvendig med et slikt konflikthåndteringssystem.

Del IV

Vedlegg

Tillegg A

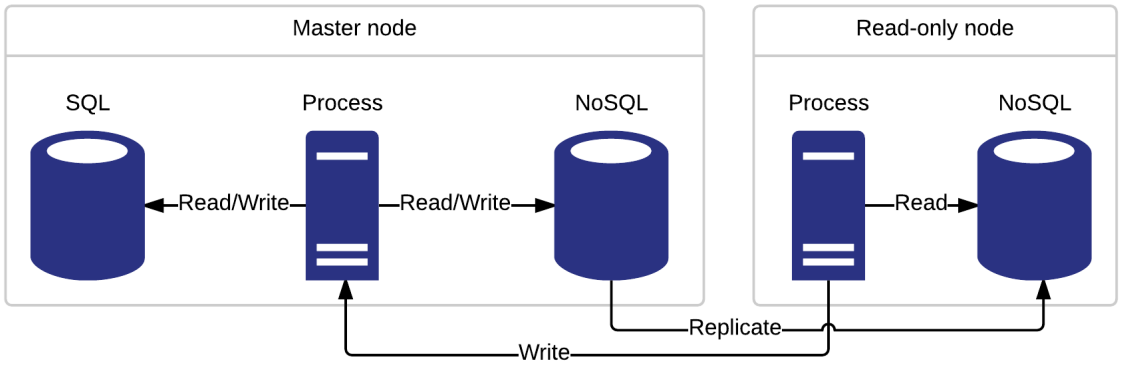
Grunndata i Norsk Helsenett

Blant oppgavene til Norsk Helsenett (NHN) er utvikling og drift av grunndataplatformen i helsesektoren [16]. Grunndataplatformen er en nasjonal tjeneste som inkluderer blant annet helseadministrative registre. Disse registrene inneholder informasjon om organisasjoner og personell i helse- og omsorgssektoren.

De helseadministrative registrene består av flere enkeltregister. Dette inkluderer blant annet Adresseregisteret – et felles nasjonalt register for presis adressering ved utveksling av helseopplysninger [17, 18]; Bedriftsregisteret – helse- og omsorgssektorens kopi av Bedrifts- og foretaksregisteret [17]; Helsepersonellregisteret – helsemyndighetenes sentrale register over alt helsepersonell med autorisasjon eller lisens [17,18]; Helsetjenestekatalogen – oversikt over behandlingssteder, tilbudte undersøkelser og behandlinger samt forventede ventetider [19]; Legestillingsregisteret – helsemyndighetenes register over ansatte leger og legestillinger i primær- og spesialisthelsetjenesten [17, 18]; og Personregisteret – helse- og omsorgssektorens kopi av det sentrale folkeregisteret [18].

I skrivende stund lagres grunndata på en SQL server. For å gjøre tjenesten høytilgjengelig skal data denormaliseres og lagres i replikerte NoSQL databaser som skal fungere som cache. Til dette brukes dokumentdatabasen RavenDB [10]. Grunndataplatformen vil da ha to forskjellige typer noder. Master-noder kan lese fra både SQL og NoSQL, samt stå ansvarlig for å oppdatere NoSQL. Read-only-noder kan kun lese fra NoSQL – og alle skriveoperasjoner, samt leseoperasjoner som ikke kan gjøres mot NoSQL, blir videresendt til en master-node.

I utgangspunktet vil kun leseoperasjoner være tilgjengelig på rene NoSQL-noder, som vist i Figur A.1. Muligheten for å tillate skriveoperasjoner på NoSQL-noder ved en senere anledning, for å gjøre også skriveoperasjoner tilgjengelig ved brudd eller nedetid på master-noder, vil også vurderes.



Figur A.1: Nettverksdiagram for høytilgjengelig grunndataplatform med én Read-Only (RO) node. Skriveoperasjoner på en RO node blir videresendt til en master node. Skriveoperasjoner på en master node blir først utført mot SQL og deretter blir Raven oppdatert og data replikeres til RO noder. Leseoperasjoner mot en master node vil først forsøke å lese mot Raven. Leseoperasjoner som Raven ikke kan svare på vil bli utført mot SQL. Fra en RO node blir slike leseoperasjoner videresendt til en master node som kan lese fra SQL.

Tillegg B

Oppsett av konflikthåndterere i RavenDB

Dette vedlegget ser på hvordan en konflikthåndterer kan implementeres mot RavenDB. Det ser først på grunnlaget for konflikthåndtering. Deretter følger tre eksempler på forskjellige fremgangsmåter hvor det velges et dokument basert på informasjon om eller i dokumentene. Til slutt introduseres konseptet med sammenfletting av dokumenter.

B.1 Grunnlag for håndtering av konflikter

Når serveren starter setter den opp koblingen til databasen og knytter på en eller flere konflikthåndterere. Når Raven finner en konflikt vil serveren dermed få beskjed og bli gitt en mulighet til å løse konflikten. En konflikthåndterer kan knyttes til Raven som vist i Listing 5. Her blir `HtkBusinessResolver` som implementerer interfacet `IDocumentConflictListener` knyttet til `documentStore`. Når Raven oppdager en konflikt i denne DocumentStore-instansen vil konflikten bli forsøkt løst med `HtkBusinessResolver`.

```
var documentStore = new DocumentStore
{
    Url = "localhost:8080"
}.RegisterListener(new HtkBusinessResolver()) as DocumentStore;
```

Listing 5: Kodesnutt som viser hvordan Raven instansieres til å bruke konflikthåndtering.

Det er en del kode som vil være for det meste lik mellom forskjellige konfliktløsere. Denne koden er samlet i Listing 6. Eksemplene som følger arver fra klassen `AbstractHtkBusinessResolver` som løser konflikter ved å implementere metoden `ResolveConflict`.

Det er metoden `TryResolveConflict` fra `IDocumentConflictListener` som gjør jobben med å løse konflikten. Den sjekker her først at dokumentet er av riktig type ved å sjekke at identifikatoren (`key`) starter med riktig prefiks. Dersom dokumentet er av feil type gir `HtkBusinessResolver` beskjed om at konflikten ikke er løst slik at Raven kan forsøke å finne riktig konfliktløser. Etter at det er kontrollert at dokumentet er av riktig type

```
using Raven.Abstractions.Data;
using Raven.Client.Listeners;

public abstract class AbstractHtkBusinessResolver : IDocumentConflictListener
{
    public bool TryResolveConflict(string key,
        JsonDocument[] conflictedDocs, out JsonDocument resolvedDocument)
    {
        // This is the resolver for HtkBusiness. Return false for other objects.
        // Raven will then try other resolvers (if available)
        if (!key.StartsWith("htkBusiness/"))
        {
            resolvedDocument = null;
            return false;
        }

        // Resolve conflict in the abstract method `ResolveConflict`
        resolvedDocument = ResolveConflict(conflictedDocs);

        // Remove metadata used by Raven for the conflicted document
        resolvedDocument.Metadata.Remove("Raven-Replication-Conflict-Document");
        resolvedDocument.Metadata.Remove("Raven-Replication-Conflict");
        resolvedDocument.Metadata.Remove("@id");
        resolvedDocument.Metadata.Remove("@etag");
        return true;
    }

    protected abstract JsonDocument ResolveConflict(JsonDocument[] conflictedDocs);
}
```

Listing 6: Mal for `HtkBusinessResolver`. Senere eksempler bruker denne malen ved å implementere metoden `ResolveConflict`.

løses konflikten. Her gjøres et kall til den abstrakte metoden som vil bli implementert i eksemplene som følger. Til slutt slettes markørene som fortalte Raven at dokumentet hadde en konflikt før metoden gir beskjed om at konflikten er løst.

B.2 Konflikthåndterer basert på metadata

Listing 7 viser et eksempel på hvordan en simpel konfliktløser kan implementeres. Her blir dokumentene sortert etter `LastModified` feltet fra dokumentets metadata og det nyeste dokumentet velges.

B.3 Konflikthåndterer basert på Json

Eksempelet over brukte metadata som er tilgjengelig fra `JsonDocument`-datatypen for å velge hvilket dokument som vinner konflikten. Det er også mulig å bruke data direkte fra Json-dokumentet for å gjøre det samme. Ved å bruke `DataAsJson` fra `JsonDocument` kan

```
public class SimpleHtkBusinessResolver : AbstractHtkBusinessResolver
{
    protected override JsonDocument ResolveConflict(JsonDocument[] conflictedDocs)
    {
        // Get the document that was last updated
        // using LastModified from metadata
        var resolvedDocument = conflictedDocs
            .OrderByDescending(d => d.LastModified)
            .First();

        return resolvedDocument;
    }
}
```

Listing 7: Konflikthåndterer som velger dokumentet som sist ble oppdatert.

man både lese og skrive data uten å tolke Json-formatet selv.

```
public class JsonHtkBusinessResolver : AbstractHtkBusinessResolver
{
    protected override JsonDocument ResolveConflict(JsonDocument[] conflictedDocs)
    {
        // Get the document that was last updated
        // using LastChanged from the JsonDoc
        var resolvedDocument = conflictedDocs
            .OrderByDescending(LastChanged)
            .First();

        return resolvedDocument;
    }

    private static string LastChanged(JsonDocument doc)
    {
        return doc.DataAsJson["LastChanged"].ToString();
    }
}
```

Listing 8: Eksempel på konfliktløser som bruker `DataAsJson` fra `JsonDocument`. Her sorteres dokumentene etter verdien i feltet `LastChanged` fra Json-dokumentet og deretter velges dokumentet hvor dette feltet har høyest verdi.

Listing 8 er et eksempel på hvordan dette kan brukes. Her hentes verdien i feltet `LastChanged` fra Json-dokumentet og dokumentet med senest dato blir valgt. Ulempen med denne fremgangsmåten er at hvert enkelt felt hentes – og eventuelt deserialiseres etter behov – separat før bruk. Det kan by på ekstra arbeid dersom flere felter skal brukes eller dersom informasjonen er på et mer komplekst format.

B.4 Konflikthåndterer basert på POCO

Et Json-dokument kan deserialiseres til POCO, *Plain Old C# Object*, et objekt i programmeringsspråket C#. Dette åpner for muligheten til å gjøre operasjoner på objekter med statisk typing og struktur på samme måte som i andre deler av programmet. Dermed kan jobben med å løse konflikten bli enklere i mange sammenhenger fordi man kan bruke datatypen og medfølgende metoder som dokumentet er basert på. Denne fremgangsmåten kan ta hensyn til innholdet av hvert dokument og det er mulig å bruke tilpasset logikk for å håndtere konflikten.

Listing 9 er et eksempel på en konflikthåndterer som bruker denne fremgangsmåten. Hvert Json-document i `conflictedDocs` blir deserialisert til en `Business`. Business objektene kan deretter manipuleres etter ønske og behov. I dette tilfellet brukes kun `LastChanged` feltet til sortering slik at eksempelet skal ha tilsvarende funksjonalitet som tidligere eksempler i vedlegget. Til slutt serialiseres det endelige Business objektet til Json og skrives inn i data feltet til `resolvedDocument` som returneres.

```
public class PocoHtkBusinessResolver : AbstractHtkBusinessResolver
{
    protected override JsonDocument ResolveConflict(JsonDocument[] conflictedDocs)
    {
        // Get a resolveDocument with the correct metadata
        var resolvedDocument = conflictedDocs.First();

        // Deserialize the docuemnts to Business (POCO)
        // then get the htkBusiness that was last updated
        // using LastChanged from Business
        var htkBusiness = conflictedDocs
            .Select(Deserialize)
            .OrderByDescending(b => b.LastChanged)
            .First();

        // Convert POCO to JSON and save to resolvedDocument
        resolvedDocument.DataAsJson = RavenJObject.FromObject(htkBusiness);

        return resolvedDocument;
    }

    private static Business Deserialize(JsonDocument json)
    {
        return JsonConvert.DeserializeObject<Business>(json.DataAsJson.ToString());
    }
}
```

Listing 9: En konfliktløser som deserialiserer Json-dokumentene til POCO for å løse konflikten.

B.5 Kombinere dokumenter

Eksemplene i dette vedlegget bruker informasjon fra dokumentene til å velge ut et enkelt dokument og forkaste de andre. Målet i denne oppgaven, derimot, er å finne en løsning som kombinerer endringene fra alle utgavene til å bygge et dokument som tar hensyn til data fra alle. Kapittel 6 og 7 ser på muligheten for å bruke metoder fra versjonskontrollsystemer til å gjøre nettopp dette. I stedet for å deserialisere Json-dokumentene, slik som i Listing 9, brukes de serialiserte Json-dokumentene.

Bibliografi

[1] S. D. Kuznetsov and A. V. Poskonin, "Nosql data management systems," *Programming and Computer Software*, vol. 40, no. 6, pp. 323–332, 2014.

[2] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, *Managing update conflicts in Bayou, a weakly connected replicated storage system*, vol. 29. ACM, 1995.

[3] P. Baudiš, "Current concepts in version control systems," *arXiv preprint arXiv:1405.3496*, 2014.

[4] T. Mens, "A state-of-the-art survey on software merging," *IEEE transactions on software engineering*, vol. 28, no. 5, pp. 449–462, 2002.

[5] "MongoDB." URL https://www.mongodb.com. Dato: 05.05.2017.

[6] "BSON." URL http://bsonspec.org. Dato: 05.05.2017.

[7] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pp. 363–366, IEEE, 2011.

[8] "JSON." URL http://www.json.org. Dato: 24.04.2017.

[9] "NHNDtoContracts." URL https://github.com/NorskHelsenett/NHNDtoContracts. Dato: 27.04.2017.

[10] "RavenDB." URL https://ravendb.net. Dato: 24.04.2017.

[11] "Replication: Advanced replication details." URL https://ravendb.net/docs/article-page/3.5/csharp/server/kb/advanced-replication-details. Dato: 30.05.2017.

[12] O. Eini, "The design of RavenDB 4.0, Replication from server side." URL https://ayende.com/blog/174113/the-design-of-ravendb-4-0-replication-from-server-side. Blogg fra 24.05.2016.

[13] O. Eini, "RavenDB 4.0 Features: Document Versioning." URL https://ayende.com/blog/177665/ravendb-4-0-features-document-versioning. Blogg fra 07.04.2017.

[14] "Git." URL https://git-scm.com. Dato: 24.04.2017.

[15] V. Driessen, "A successful git branching model," *URL http://nvie.com/posts/a-successful-git-branching-model*, 2010.

[16] "Norsk Helsenett, Oppgaver og prosjekter." URL https://www.nhn.no/oppgaver-og-prosjekter/Sider/default.aspx. Dato: 02.05.2017.

[17] "Helseadministrative registre (Norsk Helsenett)." URL https://www.nhn.no/helsenettet/helseadministrative-registre/Sider/default.aspx. Dato: 02.05.2017.

[18] "Helseadministrative registre (Direktoratet for e-helse)." URL https://ehelse.no/helseadministrative-registre. Dato: 02.05.2017.

[19] "Oppdaterte ventetider innenfor somatisk helsetjeneste." URL https://helsenorge.no/Kvalitetsindikatorer/kvalitetsindikator-sykehusopphold/oppdaterte-ventetider-somatisk-helsetjeneste. Dato: 07.06.2017.