Sammendrag

Moderne kommersielle programvaresystemer leverer ofte tjenester til opptil flere hundre tusen brukere over Internett, det vil si ved hjelp av HTTP - applikasjonsprotokollen. Det er på slike systemer at NoSQL - databaser gjerne tas i bruk da de i vesentlig grad er i stand til å lagre stadig større datavolum som oppstår i et stadig raskere tempo mer effektivt. NoSQL - databaser er også mer horisontalt skalerbare enn de tradisjonelle relasjonsdatabasene, det vil si at de egner seg bedre til å dele ut og kopiere dataelementer over en klynge av databaseprosesser.

En av de største utfordringene innen drift av moderne kommersielle programvaresystemer som bankapplikasjoner, sosiale medier og netthandelssytemer er kunsten å minimalisere nedetid som følger av oppdatering av systemet. For mange store bedrifter som eier og admistrerer slike systemer er det totalt uaktuelt å dekommisjonere hele eller deler av systemet for å installere en liten programvareoppdatering eller resirkulere minne. Til det vil nedetiden til systemet medføre utålelige inntektstap. Derfor oppgraderer mange bedrifter systemene sine ”online”, det vil si at opppgraderingen gjøres uten å slå av en eneste datamaskin, og uten å forstyrre behandlingen av forespørsler fra brukere. Erfaringer fra industriene tilsier at slike levende oppgraderinger er lettere sagt enn gjort, især når det kommer til oppgraderinger av applikasjonens datamodell mens den opererer i et produksjonsmiljø.

Moderne programvaresystemer utvikles gjerne under en smidig utviklingssykel der nye versjoner, eller oppdateringer, publiseres til bruk opptil flere ganger om dagen. Slike oppdateringer kan endre programvaresystemets datamodell, eller ”skjema” som det heter i relasjonelle databaser. For å utføre slike opppgraderinger tryggest mulig blir systemet oppgradert på rullerende vis. Denne masteroppgaven setter som mål å realisere støtte for levende oppgradering av data-modeller i høytilgjengelige systemer uten nedetid ved å utvikle et eget administrasjonsverktøy til databasehandteringssystemet Voldemort. Dette verktøyet tillater applikasjonsutviklere å legge inn transformasjonsfunksjoner som kalles på ”lazy” vis når hver enkelt datatuppel aksesseres i databasen.

Forord

Min masteroppgave presenterer et modulært programvarebibliotek som automatiserer oppdatering av semistrukturerte datamodeller i distribuerte, aggregatorienterte databasesystemer. Rapporten utgjør min besvarelse som vurderes i emnet TDT4900 - Datateknologi, masteroppgave, og utgjør samtidig mitt siste innleveringsarbeid i studieprogrammet MTDT - Datateknologi ved Norges Teknisk - Naturvitenskapelige Universitet i Trondheim. Oppgaven er basert på vitenskaplige kilder funnet og diskutert i løpet av fordypningsprosjektet jeg gjennomførte høsten 2017.

Formålet med oppgaven er å utforske hvordan prosessen med å oppgradere moderne webapplikasjoner som allerede kjører i et fungerende, aktivt produksjonsmiljø uten å slå av tjenesten. En egen løsning for denne problemstillingen er blitt implementert og testet i et realistisk oppgraderingsscenario for en typisk datamodell i en ekommersiell setting.

Den enkelte leser behøver ikke ha noen dype forkunnskaper om datamaskinvare eller operativsystemer. Det antas imidlertid at leseren er kjent med fenomenet ”prosess” i kontekst av operativsystemer, samt mønsteret for fjernt prosedyrekall, tradisjonelle databasesystemer, transaksjonsmønsteret og dets fire kvalitative egenskaper.

En stor, personlig takk rettes til min veileder Svein Erik, for gode, motiverende svar på mine spørsmål og usikkerheter rundt dette prosjektet, samt frie tøyler til å forme masteroppgaven etter eget ønske.

Rapporten er skrevet i LaTeX, og benytter en mal laget av Agus Ismail Hasan. [1] Takket være hans arbeid med denne malen sparte jeg mye tid på å sette opp dokumentets tekniske struktur, og det er derfor forfatteren krediteres i dette forordet.

Jeg vil også takke min tante, forhenværende lærer og utdannet logoped Nella Lovise Bugge, for hjelp med korrekturlesing av denne prosjektrapporten.

Trondheim, 28. februar 2018

Vegard Bjerkli Bugge

[1] Malen er tilgjengelig fra DAIM sin FAQ, `https://daim.idi.ntnu.no/faq_innlevering.php`

Innhold

Forkortelser

Forkortelse	=	Definisjon
SQL	=	Structured Query Language
DDL	=	Data Definition Language
SLA	=	Service Layer Agreement

Kapittel 1

Introduksjon

Dette kapitlet introduserer problemstillingen som oppgaven skal besvare og motivasjonen som ligger bak. Videre skisseres målene for løsningen av problemstillingen, og et eget delkapittel beskriver rammebetingelsene og gyldighetsområdet for denne løsningen. Siste delkapittel beskriver strukturen på oppgaven.

Dette kapitlet introduserer problemstillingen som oppgaven skal besvare og dets industrielle/vitenskaplige bakgrunn. Problemstillingens tema, dens overordnede mål og mening, begrensinger av det presenterte resultatets gyldighet, og historikk presenteres her.

1.1 Bakgrunn

Moderne nettbutikker, offentlige nettbaserte tjenester, og nettbankapplikasjoner stilles svært strenge krav til tilgjengelighet. Aller helst skal en hvilken som helst kunde av en populær nettbutikk som Amazon kunne se på og legge ting i handlekurven, for deretter å betale for dem **når som helst, til alle døgnets tider**. At en vare blir lagt i handlekurven to ganger eller at en kunde leser utdatert informasjon om en vare like etter at den er blitt oppdatert har ikke så mye å si, for den slags småfeil lar seg alltid rettes opp i etterkant.

I tjenestenivåavtalen [1] (eng. "Service Level Agreement") til Amazon EC2 oppgis en tilgjengelighetsgaranti på 99,95 prosent (Bass et al., 2013). Til tross for denne høye prosenten tilgjengeligheten må programvarearkitekter som vil gjeste sine systemer på EC2 ta

[1] Tilgjengelig på url `https://aws.amazon.com/ec2/sla/`

høyde for den halve prosentandelen der plattformen ikke er tilgjengelig for tjenesteleveranse.

Hvert sekund nedetid teller når det kommer til høyt-traffikerte tjenester på Internett som det sosiale mediet Facebook og tidligere nevnte Amazon sine skytjenester. Den 21. april 2011 hadde skyplattformen Amazon EC2 en periode med nedetid på fire dager (Bass et al., 2013). Dette tjenesteavbruddet rammet mange oppstartsselskaper som benyttet skyplattformen, inklusive Reddit, Quora og FourSquare. Schiller (2011) ved Information Today rapporterer at årsaken til hendelsen kom av en planlagt konfigurasjonsoppdatering som medførte at mange tjenestenoder mistet kontakten med backuptjenerne. Den samlede effekten av at alle nodene automatisk prøvde å gjenetablere forbindelsen førte til en overbelastning av forespørsler mot disse tjenerne.

En interressant bemerkelse fra denne episoden er at Netflix også var en hyppig bruker av plattformen på det tidspunktet webtjenesten gikk ned, uten at det gikk utover strømme - tjenestens egen tilgjengelighet. Forklaringen var at Netflix sine ingeniører tok høyde for den halve promillen som EC2 sin tjenestegaranti ikke dekket, blant annet ved å spre flere instanser av sine tilstandsløse tjenester utover flere av Amazon sine tilgjengelighetssoner (Bass et al., 2013).

Nedetid, den forventede tiden en plattform eller et programvaresystem ikke kan utføre dets definerte arbeidsoppgaver for dets brukere i løpet av en definert tidsperiode, er sterkt knyttet til systemets tjenestenivågaranti. Slike tilgjengelighetsgarantier baseres på beregninger med stokastiske modeller, for eksempel Markoff-analyse eller feil-tre (Bass et al., 2013). Ved hjelp av nevnte verktøy kan man anslå en forventningsverdi for hvor lang tid det vil gå mellom hvert feilscenario som rammer systemet slik at det blir totalt utilgjengelig for bruk.

Man kan også estimere en forventningsverdi for hvor lang det tar å reparere eller maskere nevnte feil slik at tjenester kan leveres av systemet som normalt. I lys av programvare som for eksempel databasesystemer er den førstnevnte verdien i praksis tiden fra en instans slås av til en ny startes opp, for eksempel ved en programvareoppdatering. Verdien til den andre variabelen påpeker tilsvarende hvor lang tid en programvarerestart tar.

Ut ifra en studie av flere dusin feilscenarier i storskala internettsystemer gjorde Oppenheimer et al. (2003) følgende konkluderende observasjoner: (1) operatørfeil er den hovedsaklige feilkilden i to av tre tilfeller; (2) operatørfeil har størst innvirkning på reparasjonstiden i to av tre internett-tjenester; (3) blant operatørfeil er konfigurasjonsfeil (feil syntaks, inkompatible argumenter) vanligst.

I en annen undersøkelse, der totalt 51 databaseadministratorer med varierende fartstid i yrket ble intervjuet, identifiserer Oliveira et al. (2006) i alt åtte kategorier feilscenarier som oppstår i et databasesystem som kjører i et produksjonsmiljø: leveranse til produksjonsmiljø (deployment), ytelse (performance), strukturer i databasen (structure), tilgangsrettigheter (access-privilege), vedlikehold (maintenance), diskplass (space), feil i programvare (DBMS), og feil i maskinvare (hardware). I de fem førstnevnte er det databaseadministratoren som er den typiske hovedårsaken (i over 50 prosent av problemene som ble oppgitt under intervjuene) til at feil av disse typene oppstår.

Observerte trender innen flere forskjellige typer næringsvirksomhet, deriblant kundestøtte, industriell produksjon, e-kommers, finans, og banktjenester (Dumitraş et al., 2010; Choi, 2009) tilsier at det er et sterkt behov for distribuerte systemarkitekturer som støtter online-oppgraderinger. Oppgraderingsrutiner for kjørende databaseapplikasjoner som fordrer eller påtvinger nedetid er ikke lengre forsvarlige i lys av tjenestenivåavtalene som deres flerfoldige tusen klienter tilbys.

Den mest sentrale karakteristikken ved online-oppgradering, programvareoppgradere uten stopp i systemet, er at den gamle versjonen av applikasjonen må kjøre samtidig som den nye installeres, slik at tjenestene applikasjonen leverer ikke blir utilgjengelig for dets brukere. Choi (2009) kaller denne rutinen for ”hot rollover”. I tillegg må installasjonsoperasjonen ikke forstyrre applikasjonens leveranse av tjenester, e.g. behandling av innkommende HTTP-forespørsler.

Et sentralt problem innen online-oppgradering er kunsten å holde styr på pakkeavhengigheter. Dette må gjøres for å oppdage om den gamle og nye versjonen har delte avhengigheter, det vil si at begge avhenger av samme programvarepakke, men ikke nødvendigvis samme versjon av denne pakken. For at tjeneren skal unngå å miste data eller å gå ned må begge versjonene av en og samme pakke installeres på tjeneren. I praksis benytter oppdateringsprogrammet som handterer avhengigheter en form for manuelt skrevet konfigurasjonsfil der alle avhengigheter listes i form av par av unike pakkenavn og påkrevd versjon. For eksempel leser pakkehandtereren til NodeJS inn avhengigheter fra en JSON-fil med navn ”packages.json”, som vedlikeholdes av utviklerne selv.

Disse inputfilene er altså kilder til menneskelige feil, som for eksempel syntaksfeil, eller deprekeringsadvarsler. Det er bevist at problemet med å løse opp avhengigheter er NP-hardt ved å utføre en reduksjon (transformering av problemet og dets input) fra **3SAT** – problemet (Dumitraş and Narasimhan, 2009). Dermed er det grenser for hvor mange og store avhengigheter et programvaresystem kan ha før kjøretidskostnaden for avhengighetsbehandling (i for eksempel APT-registeret) vokser seg altfor stor.

Derfor har store aktører i industrien i de senere år innført prossessen *rullerende oppgradering*, der programvaren på én etter én tjener i klyngen av tjenere blir oppdatert. Ved en automatisert rullerende oppgradering kan man i utgangspunktet kun gjennomføre patching av programvare, det vil si at brukergrensesnittet som applikasjonen tjener må i den nye versjonen være bakoverkompatibel med den gamle. Eventuelle konflikter må løses manuelt.

Opp igjennom det siste tiåret har det vært vanlig å oppgradere programvare som kjører i et system av flere instanser, eller prosesser, på rullerende vis. I denne manuelt kontrollerte oppgraderingsmetoden blir én etter én instans av den gamle versjonen av programmet avsluttet og erstattet med en instans av den nye versjonen. Et vesentlig problem med denne metoden er at applikasjonens datamodell er som regel realisert i et databasesystem som er instansiert i en separat prosess fra webapplikasjonsprosessen på en og samme fysiske tjenerdatamaskin. Dermed oppgraderes datamodellen til hver applikasjonsinstans på et annet tidspunkt enn koden til selve applikasjonen. Dette medfører til at det dsitribuerte produksjonsmiljøet befinner seg i en mikset tilstand - en uoppgradert applikasjonsposess kan potensielt interagere med en oppgradert datamodell og vice versa, noe som kan introdusere uante feilkilder til applikasjonen.

1.2 Oppgavens problemstilling og mål

Denne masteroppgaven opererer med en konkret definert problemstilling. Her presesnteres denne definisjonen, hvorpå et sett med konkrete, oppnåelige, og tidsbestemte målpunkter forbundet til definisjonen av problemstillingen også blir listet opp.

Problemstilling: Formålet med dette prosjektet er å implementere et høytilgjengelig programvaresystem bygget med en nøkkel-verdi-datamodell realisert med databasehandteringssystemet Project Voldemort, der oppgraderinger som involverer endringer

Oppgaven har hatt følgende overordnede mål:

1. Beskrive sammenhengen mellom kontinuerlig programvareleveranse og levende oppgradering av datamodeller

2. Modellere en modulær løsning der semistrukturerte datamodeller, også referert til som NoSQL-datamodeller, kan oppgraderes synkront med applikasjonstjenerne

1.3 Oppgavens struktur

Denne rapporten har følgende struktur. Kapittel 1 er introduksjonskapitlet, som illustrerer problemstilllingen som undersøkes og kort hvordan. Det beskrives hvordan litteratursøket ble gjennomført, samt hvordan analysen av de ulike oppgraderingsmetodene utføres.

Kapittel 2 omtaler relevant teori om NoSQL - datamodeller, en kort innføring i databasearkitektur, distibuerte systemer, tilgjengelighetskvaliteten til et distribuert programvaresystem, oppgradering av programvaren som utgjør noder i distribuerte systemer og nedetid i systemer som oppstår i forbindelse med programvareoppgradering av dem. Konseptet "levende oppgradering av programvaresystemer" defineres og forklares her. I teorikapitlet presenteres også NoSQL - DBMSet Project Voldemort, som vedlikeholdes av et dedikert utviklingslag hos LinkedIn.

Det tredje kapitlet beskriver et knippe oppgraderingsverktøy som på hvert sitt eget vis kommer i bukt med versjonmiks - problemet beskrevet i kapittel 1.1. I kapittel 3 sammenliknes løsningsforslagene for online oppgradering av distribuerte databasesystem, som ble lest om . Denne sammenliknignen er "fortrinnsvis" kvalitativ (det vil si at det har vist seg komplisert å produsere et optimalt datagrunnlag for en kvantitativ test) der vurderingskriteriene bunner i hvor tilgjengelig løsningen er for det allmenne marked, hvor forståelig publikasjonene som presenterer er for rapportens forfatter og popularitet i industrien - det er jo tross alt et problem av industriell undertone som besvares her.

Kapittel 4 kalles for *Levende oppgradering av datamodeller realisert med Project Voldemort*. Dette kapitlet vil presentere systemdesignet rundt en kommersiell webapplikasjon kalt "DBUpgradinator", som kan inndeles i en frontend-del med presentasjonslogikk formet av CSS - og JS - filer og en backend-del med kontroll-logikk skrevet i Java som ved hjelp av serialisering med binærdatakodingsbiblioteket Apache Avro snakker med datalageret, en instans av den distribuerte oppslagstabellen Voldemort. I kapittel 4 blir også en kontinuerlig oppdateringsleveranseløsning for hele applikasjonen, inklusive dets implisitte dataskjema, skissert.

I kapittel 6 konkluderes evalueringen og i det beskrives forslag til videre kvantitativt feltarbeid som kan bygge på denne analysen.

Bibliografi

Bass, L., Clements, P., Kazman, R., 2013. Software architecture in practice.

Choi, A., 2009. Online application upgrade using edition-based redefinition. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades. ACM, p. 4.

Dumitraş, T., Narasimhan, P., 2009. Why do upgrades fail and what can we do about it? toward dependable, online upgrades in enterprise system. In: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware. Springer-Verlag New York, Inc., p. 18.

Dumitraş, T., Narasimhan, P., Tilevich, E., 2010. To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. In: ACM Sigplan Notices. Vol. 45. ACM, pp. 865–876.

Oliveira, F., Nagaraja, K., Bachwani, R., Bianchini, R., Martin, R. P., Nguyen, T. D., 2006. Understanding and validating database system administration. In: USENIX Annual Technical Conference, General Track. Boston, MA, pp. 213–228.

Oppenheimer, D., Ganapathi, A., Patterson, D. A., 2003. Why do internet services fail, and what can be done about it? In: USENIX symposium on internet technologies and systems. Vol. 67. Seattle, WA, pp. 11–25.

Schiller, K., 06 2011. Amazon ec2 outage highlights risks. Information Today 28 (6), 10, name - Amazon.com Inc; Copyright - Copyright Information Today, Inc. Jun 2011; Document feature - Photographs; Last updated - 2013-06-27.