Orkestrering af distribuerede systemer over store datamængder

Jesper Marrup

Kongens Lyngby 2013
IMM-M.Sc.-2013-35

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-35

Indholdsfortegnelse

Chapter 1

Introduktion

I det sidste årti, og i stigende grad i fremtiden, bliver distribuerede eller parallelle systemer mere og mere almindelige. Systemerne uddelegering de forskellige opgaver for at sikre et hurtigt respons. Derfor er de afhængige af distribuerede systemer, hvor data kan opdeles mellem forskellige servere. Biler nutildags, anvender elektroniske systemer, hvor systemerne består af distribuerede enheder, hvor sensorer og beregningsenheder er placeret forskellige steder på bilerne. Nogle af komponenterne bruges i sikkerhedskritiske systemer såsom ESP(elektronisk Stabiliseringsprogram), bremser, airbags og styretøjet. Det er yderst vigtigt at disse virker efter hensigten på alle tidspunkter. Nyere biler med automatisk indstilling af xenon har to sensorer; en foran og en bagved. De måler højde forskellen på for- og bag aksen for derefter automatisk, at kunne regulere på xenon lygterne, så modkørende ikke bliver blændet. At designe distribuerede systemer er dog ikke nogen triviel opgave.

Distribuerede system er oftest meget store og indeholder mange komponenter med forskellige opgaver. Det resulterer i komplekse systemer, som er svære at ræsonnere om på en uformel måde. Kan der opstå hårdknuder i systemet? Og hvad er tilstanden af hele systemet på et givent tidspunkt?

Kompleksiteten stiger når der skal udvikles, debugges og testes sådanne systemer. Det kan være nemt at teste en enkel komponent, men når de tilføjes til det overordnede system, bliver opgaven en del vanskeligere. Kommunikationen

mellem enhederne tager tid og tilstanden er delt mellem enhederne, hvor det er umuligt at kende den komplette tilstand af systemet til et givent tidspunkt. Selv simple opgaver bliver komplekse teoretiske problemer, når man har med distribuerede systemer at gøre, hvis man ser på synkronisering af tiden i systemet.

Derfor udvikles en model af systemet på et passende abstraktionsniveau før den aktuelle implementering, og det giver udviklerne mulighed for bedre at kende systemet. Modellerne kan både bruges til simulering, testning og for at bevise egenskaberne af modellen, ved at bruge statisk analyse, samt at give et mere sikkert og korrekt design.

Til denne opgave findes der forskellige metasprog - primært en række proces algebra(også kaldet proces calculi). Proces calculi er en forskelligartet familie af beslægtede metoder til formel modellering af parallelle systemer. Følgende sprog falder inden for denne kategori: Calculus of Communicating Systems (CSS), Communicating Sequential Processes (CSP) og π-calculus. Selvom sprogene er forskellige, deler de nogle typiske funktioner. Processer opbygges typisk ved, at bruge et begrænset antal primitiver og operatorer, hvilket defineres af de algebraiske love. Det giver mulighed for at manipulere processer ved brugen af ligninger. Kommunikationen sker gennem beskeder eller kanaler, i stedet for at manipulere en delt ressource.

Dog findes et relaterende koordineringssprog kaldet Linda. Her opererer primitiverne på indtastede data objekter, gemt i tupler. Processer kommunikerer ikke gennem beskeder eller kanaler, men i stedet ved at manipulere tupel spaces. Kernel Language for Agents Interaction and Mobility (KLAIM) er en formalisering, hvilket kombinerer ideer fra både proces calculi og Linda. Fra Linda anvendes tuple space koncepterne, og det kombineres med et set af operatorer, som er lånt fra CCS.

Denne afhandling er lavet i samarbejde med Henrik Pilegaard fra Kapow Software. Hos Kapow Software anvender Henrik Pilegaard KLAIM til at modulere distribuerede systemer, hvorfor der er stiftet bekendtskab med KLAIM og det anvendes i denne afhandling. Med KLAIM er det nemt at udtrække et subset, eller udvide syntaksen og det gør det nemt at arbejde med. Man kan starte med kun at arbejde med en begrænsning af KLAIM, og når systemet er udviklet er det nemt at udvide. Valget af KLAIM i denne afhandling er derfor bestemt af Kapow Software.

1.1 Afhandlingens formål

Formålet med denne afhandling, er et ønske fra Kapow Software, hvor der først udvikles en variant af KLAIM til at beskrive processer. KLAIM bruges som det fortolkende domænespecifikke sprog. Dernæst skal det undersøges, hvilken type system transaktionsbegreberne kan implementeres på ryggen af, for at give det udviklede system robusthed og integritet. Det udviklede system skal garantere at en sekvens af behandlinger kun bliver udført én og netop én gang, og hvor ingen behandlinger går tabt. Da der er mulighed for sideeffekter i systemet, er det en vigtig egenskab. Jeg tror, at det kan hjælpe Kapow Software, hvis man flytter transaktionsbegreberne over i det system der hedder Storm.

Storm er et open source distribueret tidstro beregningssystem, som garanterer at data altid bliver eksekveret. Ydermere er Storm fejl-tolerant, dvs. at hvis en knude går ned, så vil Storm automatisk prøve at genstarte den. Hvis knuden ikke kommer op, så vil arbejdet blive overført til en anden knude. Fokusset er blandt andet, at undersøge om Storm vil opfylde Kapows ønsker, og se hvilke muligheder Storm kan give.

Henrik Pilegaard har til at starte med udleveret en KLAIM implementation, som han gerne vil have at jeg inkorporerer i mit system. Dvs. at den udleverede kode vil blive modificeret og tilpasset. Derfor er det ikke nødvendigt, at sætte mig ind i, hvordan KLAIM programmer skal parses. Det giver mulighed for, at arbejde mere med Storm, og finde ud af hvad det kan bruges til og om det overhovedet kan bruges.

Der vil blive udarbejdet to forskellige modeller. Begge modeller skal behandle tupler som sendes ind i systemet. Model 1 vil anvendes, som det første skridt, til at teste at tupler altid bliver behandlet. Selv hvis en tupel fejles i systemet skal der sørges for at genudsende den fejlede tupel, indtil den er behandlet. I model 1 vil der dog være mulighed for, at en tupel bliver behandlet mere en én gang, men det skal garanteres at den altid behandles. Model 1 bliver kun et lille skridt på vejen mod den endelige løsning.

Henrik fra Kapow har ønsket at systemet skal have mulighed for at indeholde en service. En service findes dog kun i Storm regi, og den har til opgave at manipulere de tupels der bliver outputtet til et tupel space med en service. Det betyder at en tupel der sendes til et tupel space med en service, først manipuleres af den tilhørende service, før den bliver tilføjet til tupelspacet. En service håndteres derfor i en individuel bolt.

Nu har vi sikret at data altid bliver behandlet og det giver anledning til model 2. I model 2 garanteres det at en tupel altid behandles én og netop kun én gang.

Selvom en tupel fejler, vil systemet sørge for at den bliver genudsendt igen, og der sikres at den kun udføres én og netop én gang.

Der præsenteres en use case, hvor det trinvist gennemgås, hvordan et KLAIM program oversættes til det sidst kører i Storm. Der er udviklet en variant af KLAIM , som kan oversættes og denne variant kaldes StormKLAIM - forkortet til sKLAIM. Til sidst er lavet en diskussion af robustheden for Storm, samt en test af ydeevnen og skalerbarheden for at se om det lever op til forventningerne.

I begge modeller sker interaktionen mellem de forskellige komponenter igennem multiple distribuerede tupel spaces. En tupel er anonym og udtrækkes fra et tupel spaces vha. en pattern-matching mekanisme. Implementeringen der anvendes til at håndtere tupel spaces er en SQL-space implementering. Flaskehalsen i systemet er SQL-space implementeringen, og derfor ønskes den fjernet fra systemet, men pga. manglende tid er det ikke nået. Derimod er der fremlagt ideer til, hvordan en model uden brug af tupel spaces kan implementeres.

Det udførte arbejde består af følgende dele:

- Forståelse af KLAIM syntaksen, samt finde en variant af KLAIM , som er så simpel som muligt, men stadig indeholder nok udtrykskraft til at modellere den første model.
- Designe og implementere oversættelsen af KLAIM , så det kan anvendes til, at oprette et oversættelses map der indeholder de nødvendige informationer fra KLAIM -programmet.
- Design og udvikling af model 1. På baggrund af det oversatte KLAIM dannes en Storm topologi til behandling af tupler.
- Test af robustheden ved at fejle tupler og se om tupler altid behandles i Storm for model 1.
- Udvikling af en variant af sKLAIM og designe den tilhørende syntaks og semantik.
- Design og udvikling af model 2. På baggrund af det oversatte KLAIM dannes en transaktions topologi til behandling af tupler.
- Test af systemets robusthed ved at fejle tupler og se om tupler kun behandles én gang i Storm for model 2.
- Teste Storms ydeevne, skalerbarhed og at analysere resultaterne for at finde flaskehalsen i systemet.
- Forbedring af systemet ved at komme med ideer til løsning på flaskehalsen i systemet, hvor SQL-spaces ikke anvendes.

1.2 Afhandlingens disposition

Denne afhandling består af 7 kapitler, hvor introduktionen er det første. Resten af kapitlerne er som følger:

Kapitel 2: indeholder baggrundsmaterialet, hvor både KLAIM og Storm introduceres. For at forstå de to modellers arkitektur, så er koncepterne fra de to forskellige systemer præsenteret. I KLAIM afsnittet præsenteres en variant af KLAIM , kaldet μKLAIM, hvor syntaks og semantikken gennemgås. I Storm afsnittet præsenteres forskellige Storm termer, almindelige Storm topologier og transaktions topologier.

Kapitel 3: indeholder mit bidrag. Hvor det implementerede subset sKLAIM præsenteres, hvor syntaksen og semantikken gennemgås. En analyse der indeholder forskellige løsningsforslag, hvor problemer og ideer bliver diskuteret. En gennemgang af hvordan sKLAIM er blevet oversat. Yderligere haves en beskrivelse af hvordan model 1 og model 2 fungerer ud fra de præsenterede ideer. Dernæst fortæller jeg hvordan systemet er testet og afprøvet under udviklingen af model 1 og 2. Til sidst rundes af med at forklare en idé til en forbedring af systemet.

Kapitel 4: indeholder en gennemgang af en udvalgt use case. Hvor man får en indsigt i hvordan et KLAIM program oversættes til en Storm topologi med henblik på den udvalgte use case.

Kapitel 5: er et kapitel der indeholder resultater fra forskellige tests af Storms ydeevne og skalerbarhed. Yderligere haves der en diskussion omkring Storms robusthed.

Kapitel 6: indeholder relateret arbejde, som også er blevet anvendt i denne afhandling. Her i blandt kan nævnes SQL-space implementeringen.

Kapitel 7: indeholder konkluderende bemærkninger, diskussion og ideer til videreudvikling af systemet.

Yderligere haves to bilag:

Bilag 1: indeholder en forklaring, hvor der gives indsigt i, hvad der sker når de forskellige operationer i et KLAIM program mødes. Der tages udgangspunkt i en KLAIM in-operation. Implementeringsdetaljerne haves som et bilad pga. det indeholder meget kildekode, da det er nemmere at forklare de teknisk betonede detaljer ud fra.

Bilag 2: beskriver hvordan man eksekverer systemet med den eksporterede .jar fil fra eclipse projektet. Man kan se hvilke argumenter der kan gives til programmet for at inputte nye tupler, programmer eller lignende.

Chapter 2

Baggrundsmateriale

I denne afhandling vil der præsenteres to forskellige implementeringer til orkestrering af distribuerede systemer over store datamængder. KLAIM er den første implementering der introduceres og det anvendes som det fortolkende domæne specifikke sprog. Det næste system der præsenteres er Storm [Mar13]. Storm er et tidstro distribueret beregningssystem, hvor det er muligt at behandle ubegrænsede strømme af data.

Det kræves at systemet der skal udvikles skal have integritet og være robust. Derfor skal transaktionsbegreberne implementeres på ryggen af det system der hedder Storm. Storm er et open source, distribueret, tidstro beregnings system hvor data altid er garanteret behandling.

2.1 KLAIM

KLAIM er den første implementering der betragtes i denne afhandling. Ønsket fra Kapow er at KLAIM programmer skal oversættes og i dette kapitel vil KLAIM blive introduceret. Der findes forskellige varianter af KLAIM og i næste kapitel uddybes netop den variant af KLAIM der kan oversættes.

2.1.1 Overblik

KLAIM [1] er et sprog som blev introduceret i 1998 i [RDNP98] og det anvendes til at modellere mobile agenter i et distribueret miljø. Valget af KLAIM's primitiver er stærkt påvirket af proces algebra og Linda[BBN+03, DBP07]. De mobile komponenter interagerer gennem multiple distribuerede tupel spaces. Kommunikationsmodellen med disse tupel spaces er asynkron og KLAIM har udvidet og bygget oven på modellen fra Linda. Det betyder at basis operationerne i KLAIM er de samme, som de originale fra Linda. Man har udvidet kommunikationsmodellen i KLAIM - bla. ved at tilføje oplysninger om hvor knuder, processer og tupler er allokeret.

KLAIM's primitiver(mht. eksplicitte lokationer) gør det muligt for programmøren at distribuere data og processer frit over knuderne i et netværk. Lokationer er sprogelementer - dvs. de kan blive oprettet dynamisk og kommunikeret over netværket og håndteret via. avancerede scoping[2] regler. Inter-proces kommunikation er asynkron, da forbruger og producent af en tupel ikke behøver at synkronisere. Oprindeligt anvendte man Linda modellen til parallel programmering på en enkelt maskine. Senere blev multiple distribuerede tupel spaces foreslået for at forbedre modulariteten, skalerbarheden og ydeevnen.

2.1.2 Forskellige KLAIM varianter

Der findes mange forskellige varianter af KLAIM, så for at give et overblik over KLAIMs mange muligheder vil de kort blive gennemgået. En mere avanceret gennemgang vil ske af netop det subset af KLAIM, som jeg har tænkt mig at anvende. Min udvalte variant af KLAIM vil blive gennemgået i et andet kapitel, hvor semantikken også introduceres. De forskellige varianter, som bliver beskrevet er:

- cKLAIM - Core KLAIM
- uKLAIM - Micro KLAIM
- KLAIM
- OPENKLAIM
- HOTKLAIM - Higher-Order Typed KLAIM

[1] a Kernel Language for Agents Interaction and Mobility

[2] Et scope er den kontekst inden for et computer program, hvor et variable navn eller en anden identifikation er gældende og kan blive anvendt.

- O'KLAIM - object oriented KLAIM
- X-KLAIM - Extended KLAIM

De første to KLAIM varianter er subsets og der vil være en dybere gennemgang af disse. De næste KLAIM specifikationer er udvidelser og disse behandles i kapitlet om relateret arbejde - se kapitel 6. Det skyldes at disse udvidelser af KLAIM er avanceret og ligger langt fra den syntaks af KLAIM der anvendes i denne afhandling. Der vil ikke blive vist nogen syntaks-tabel, men derimod vil der blive forklaret hvad de forskellige syntakser indeholder og hvilket formål de har. Dog kan en dybere gennemgang findes i [BBN+03].

2.1.3 Forskellige KLAIM implementeringer

Der findes forskellige implementeringer af KLAIM. Som tidligere nævnt har jeg fået udleveret en kildekode af Henrik Pilegaard fra Kapow Software, som indeholder en implementering af KLAIM. Den udleverede KLAIM implementering bliver anvendt i denne afhandling. En bemærkelsesværdig implementering af KLAIM er KLAVA [BDNP02]. KLAVA er et bibliotek der repræsenterer KLAIM konstruktioner som Java klasser.

X-KLAIM [BNP01] er et programmeringssprog baseret på KLAIM der anvendes til at programmere distribuerede applikationer med objekt orienteret mobil kode. Der tillades udveksling af data og processer, samt programmering af mobile agenter til at udveksle informationer i et netværk. X-KLAIM compileren producerer Java code ved at man inputter et X-KLAIM program til compileren, og outputtet er et Java program, som anvender Klava biblioteket. X-KLAIM frameworket består af en Java implementation med X-KLAIM primitiverne.

2.1.4 Syntaks

I denne sektion vil den generelle syntaks blive gennemgået - dvs. betydningen af de forskellige symboler og termer bliver forklaret. Efterhånden som mere avanceret KLAIM syntaks mødes, så vil denne blive forklaret i det pågældende afsnit. Der startes med introduktion af den simpleste KLAIM variant og løbende tilføjes ny syntaks som udvider sproget.

Et KLAIM netværk består af lokaliserede processer og lokaliserede tupler. N bliver brugt til at benævne et netværk og sammensætningen af knuder i et netværk er givet ved operatoren $||$. P og Q benævner processer og a benævner

en operation. x anvendes til at repræsentere navnet for en generel variabel. $!x$ repræsenterer bindingen af en værdi til variablen x og e repræsenterer et aritmetisk udtryk. l er en lokations konstant hvor ℓ kan referere til enten en lokations konstant eller en variabel. En tupel består af informationselementer og repræsenteres med t - et tupel element består af konstanter, variabler og variabel bindinger. Lokaliserede tupler opskrives som: $l ::< t >$ og lokaliserede processer skrives som $l :: P$. Parallel sammensætning af processor beskrives med |, som det også kendes fra CCS[3]. $a.P$ er action præfiks som beskriver en proces der først eksekverer aktionen a og derefter opfører sig som beskrevet af P. **nil** beskriver nil-processen. I KLAIM artiklerne udelades **nil** i slutningen af en process og det tillader følgende implementering: $\mathbf{out}(t)@l.\mathbf{in}(t)@l$ er det samme som $\mathbf{out}(t)@l.\mathbf{in}(t)@l.\mathbf{nil}$. Et eksempel på en KLAIM syntaks kan ses i figur 2.1.

N	::=		NETVÆRK	a	::=		AKTIONER
		$l :: P$	*Enkel knude*			$\mathbf{out}(\ell')@\ell$	*Tilføj*
	|	$l ::< l' >$	*Lokaliseret data*		|	$\mathbf{in}(T)@\ell$	*Fjern*
	|	$N_1 \mid\mid N_2$	*Net sammensæt*		|	$\mathbf{eval}(P)@\ell$	*Migrering*
					|	$\mathbf{newloc}(u)$	*Oprettelse*
P	::=		PROCESSER				
		nil	*Inaktiv process*	T	::=		TEMPLATES
	|	$a.P$	*Aktion præfiks*			ℓ	*Navn*
	|	$P_1 \mid P_2$	*Sammensætning*		|	$!u$	*Formelt*
	|	A	*Process aktivering*				

Figure 2.1: cKLAIM syntax

2.1.4.1 cKLAIM syntaks

cKLAIM kan ses som en variant af pi-calculus med process distribuering, process mobilitet og asynkron kommunikation af navne gennem delte lokaliserede datalagre. Processerne er cKLAIM's aktive beregnings enhed. De kan enten blive eksekveret samtidigt på den samme lokation eller på forskellige lokationer. cKLAIM er den simpleste version af KLAIM og den indeholder fire forskellige basis operationer, kaldet *actions*: **output**, **input**, **migration** og **creation**. To af operationer håndterer data overførsel mellem datalagerne via. **output**/**input**. **Eval** aktiverer en ny tråd som evaluerer den givne process. Den sidste operation, $\mathbf{newloc}(u)$ gør det muligt at lave en ny netværks knude. Det kan i tabellen ses at den sidste action ikke er indekseret med en adresse fordi den altid handler

[3]Calculus of Communicating Systems

lokalt. Alle andre actions indikerer eksplicit lokationen hvor de skal udføres. Syntaksen for cKLAIM kan ses i figur 2.1.

2.1.4.2 μKLAIM syntaks

μKLAIM er en udbygning af cKLAIM , hvor man tilføjer tuples, pattern-matching og en mulighed for at læse en tupel uden at fjerne den fra tupel spacet. En væsentlig forskel fra den anden syntaks er at vi nu generaliserer data begrebet fra navne til tupler. Det kommer til udtryk ved at de forskellige aktioner nu opererer på tupler - f.eks. er $\mathbf{out}(l')@\ell$ blevet til $\mathbf{out}(t)@\ell$. En tupel er en sekvens af felter, hvor et felt kan indeholde en variabel, et udtryk, en lokationen eller en lokations variabel. Der hvor en tupel er lagret kaldes et tupel space. Der er ikke nogen præcis syntaks for kategorien af udtryk(expression) e, men vi antager at den mindst indeholder basis værdier, V, og variable x - begge dele afhænger af implementationen. Templates er sekvenser af aktuelle og formelle felter og de anvendes som et mønster til at vælge tupler i et tupel space. Man skriver et formelt felt som $!x$ for værdier og $!u$ for lokationer og de anvendes til at binde variabler til værdier/lokationer.

N	::=		NETVÆRK	T	::=	$F \mid F,T$	TEMPLATES
		$l :: P$	*Enkel knude*	F	::=	$f \mid !x \mid !u$	TEMPLATE FELTER
	$\mid$	$l ::< et >$	*Lokaliseret tupel*				
	$\mid$	$N_1 \mid\mid N_2$	*Net sammensæt.*	t	::=	$f \mid f,t$	TUPEL
				f	::=	$e \mid \ell \mid u$	TUPEL FELTER
P	::=		PROCESSER				
		nil	*Inaktiv process*	et	::=	$ef \mid ef,et$	EVAL. TUPEL
	$\mid$	$a.P$	*Aktion præfiks*	ef	::=	$V \mid l$	EVAL. TUPEL FELT
	$\mid$	$P_1 \mid P_2$	*Sammensætning*				
	$\mid$	A	*Aktivering*	e	::=	$V \mid x \mid ...$	UDTRYK
				ℓ	::=	$l \mid u$	NAVN
a	::=		OPERATIONER				
		$\mathbf{out}(t)@\ell$	*Tilføj*				
	$\mid$	$\mathbf{in}(T)@\ell$	*Fjern*				
	$\mid$	$\mathbf{read}(T)@\ell$	*Læs*				
	$\mid$	$\mathbf{eval}(P)@\ell$	*Migrering*				
	$\mid$	$\mathbf{newloc}(u)$	*Oprettelse*				

Figure 2.2: μKLAIM syntax

Operationen $\mathbf{in}(T)@\ell$ evaluerer tuplen T og søger efter en matchende tupel T' i tupel spacet ℓ. Når T' er fundet vil den blive fjernet fra tupel spacet. De

tilhørende værdier af T' bliver tildelt til variablerne i T og operationen slutter. Operationen $\mathbf{out}(t)@\ell$ tilføjer den evaluerede tupel t til tupel spacet ℓ. **eval**-operationen tager som argument en proces P i stedet for en tupel og det tillader programmering af mobile agenter. Operationen $\mathbf{eval}(\mathbf{out}(t)@\ell.nil)@\ell$ kan anvendes til at simulere den forventede opførsel af $\mathbf{eval(t)}@\ell$. I μKLAIM kan processorne også læse tupels uden at fjerne dem fra tupel spacet vha. operationen $\mathbf{read}(T)@\ell$. I tupel spacet eksisterer der kun evaluerede tupels, og templates skal evalueres før de kan anvendes til at udtrække tupler fra tupel spacet. En template evalueres ved at man udregner værdien i den givne template og der sker ingen ændring for lokationen eller de formelle felter. Templates med ubundne variable i aktuelle felter kan ikke blive evalueret. Syntaksen for μKLAIM kan ses i figur 2.2.

Nu er både cKLAIM og μKLAIM syntaksen gennemgået. Da μKLAIM mest minder om den version af KLAIM vi skal implementere. Vil semantikken for μKLAIM her blive gennemået. cKLAIM er meget forskellig fra den version vi skal bruge og anvender ikke tupler.

2.1.5 Strukturel kongruens for muKLAIM

Strukturel kongruens, $\equiv$, identificerer netværk som intuitivt repræsenterer det samme netværk. I figur 2.3 ses strukturel kongruens reglerne for μKLAIM. Reglerne i figur 2.3 opfylder den mindste kongruens relation i netværket. Det kan ses at operatoren $||$ er både kommutativ og associativ. nil processen kan behandles som en afsluttet process og fjernes fra netværket(Identificer reglen) og en process invokation kan udbyttes for sin definition. Der er ikke nogen regler for de kommutative og associative egenskaber for $|$ operatoren, fordi de kan udledes af reglerne SC_1, SC_2 og SC_5.

(SC_1)	$N_1 \mid\mid N_2$	$\equiv$	$N_2 \mid\mid N_1$
(SC_2)	$(N_1 \mid\mid N_2) \mid\mid N_3$	$\equiv$	$N_1 \mid\mid (N_2 \mid\mid N_3)$
(SC_3)	$l :: P$	$\equiv$	$l :: (P \mid \mathbf{nil})$
(SC_4)	$l :: A$	$\equiv$	$l :: P \quad if A \triangleq P$
(SC_5)	$l :: (P_1 \mid P_2)$	$\equiv$	$l :: P_1 \mid\mid l :: P_2$

Figure 2.3: Strukturel kongruens for μKLAIM

2.1.6 Matching mekanismen

Funktionen til at matche tupler kan ses i figur 2.4. Matching mekanismen er det første skridt til at definere den operationelle semantik for sKLAIM. Pattern matching bruges til at vælge evaluerede tupler fra et tuplespace svarende til templaten. Da en template kan indeholde et udtryk skal disse evalueres inden de kan bruges til at matche en evalueret tupel. En evalueret tupel benævnes med $[[T]]$ fremover. Et matchende felt matcher enten værdier jvf. M_1 eller lokationer M_3. En formel værdi variabel matcher en værdi, som det ses i M_2, eller en formel lokations variabel mathcer en lokation, som det ses i M_4. Den sidste matching regel M_5 siger at en template og en tupel skal have samme antal felter og de tilhørende felter skal matche.

Resultatet af en succesfuld match funktion er en substitutions funktion. Funktionen mapper variabler fra de formelle felter, af en template med værdier eller lokationer, til de tilhørende tupel felter.

$$(M_1)\ match(V, V) = \epsilon \qquad (M_2)\ match(!x, V) = [V/x]$$

$$(M_3)\ match(l, l) = \epsilon \qquad (M_4)\ match(!u, l) = [l/u]$$

$$(M_5)\ \frac{match(eF, ef) = \sigma_1 \quad match(eT, et) = \sigma_2}{match((eF, eT), (ef, et)) = \sigma_1 \circ \sigma_2}$$

Figure 2.4: Matching regler for μKLAIM

2.1.6.1 Pattern-Matching

Patter-matching er en algoritme der anvendes til at udtrække tupler fra et tupel space. Read og in er de operationer der anvender matching mekanismen. Følgende betingelser skal overholdes når en tupel vælges fra et tupel space:

1. Template tuplen t indeholder samme antal af elementer ligesom kandidat tuplen t' fra tupel spacet.
2. I hver indeks position i i t som ikke er en variabel binding vil elementer i t_i være lig med elementet i samme position i t'. Med andre ord, $t_i = t'_i$ for alle i hvor t_i ikke er en variabel binding.

Overholder en tupel ikke disse kriterier bliver det ikke valgt og udtrukket fra tupel spacet. Hvis den søgte tupel ikke eksisterer i tupel spacet vil processen

blokke indtil tupel spacet indeholder en tupel der overholder de søgte kriterier. Resultatet af match funktionen er en binding af variabel navne til værdier i henhold til variablernes position i tuplen. Et eksempel på en matching kan være hvis vi har lokaliseret en tupel:

```
1   TelefonBog::<Jesper, 28733479> || Camilla::read(Jesper, !tlfNr)
        @TelefonBog
```

Listing 2.1: Et eksempel på en matching

Processen der er lokaliseret hos Camilla vil matche tuplen med read operationen, (Jesper, !tlfNr) med tuplen <Jesper, 28733479> lokaliseret i TelefonBog lokationen. Det første element matcher tuplen, og det andet element er en variabel binding. Tuplen bliver matchet og værdien 28733479 bliver bundet til variablen tlfNr og den kan anvendes fremover fra Camilla lokationen.

Senere vises en use case hvor KLAIM syntaksen bruges.

2.1.7 Reduktions relationen

Den sidste del af den operationelle semantik af μKLAIM består af reduktions relationen og denne kan ses i figur 2.5. Hver regel vil blive forklaret og vi starter med de to sidste regler. Når en præfiks aktion er blevet eksekveret vil aktionen blive fjernet og processen reduceres til postfix processen.

Reglen (Parallel) siger, hvis der sker en reduktion på en del af netværket, så er hele netværket ligeledes reduceret. Reglen (Strukturel) anvendes til at relatere strukturel kongruens med reduktion relationen og den siger at, hvis to netværk er strukturelt kongruente så kan de udføre de samme reduktioner.

De sidste regler opererer på aktions. Alle regler for at eksekvere aktions kræver at den søgte lokation findes. Efter eksekvering fjernes præfikset og der er kun postfix processen tilbage.

Reglen (Out) udtrykker at den evaluerede tupel, et, er resultatet af den evaluerede tupel t - et tilføjes til tupel spacet l'.

Reglen (In) og (Read) udtrykker, at hvis den matchende funktion lykkes mellem T og et, så vil den resulterende substitution σ være dannet i den tilbageværende process P. Substitutionen laves ved at substituere enhver forekomst af variabler, som både eksisterer i P og σ - med værdierne fra σ - givet at $P\sigma$. Ydermere så fjernes tupel, et, fra tupel space l' - dog gælder dette ikke for (Read).

Reglen (eval) udtrykker at process Q bliver tilføjet til node l'. Som tidligere nævnt er det antaget at alle process definitioner bliver defineret globalt og derfor er det ikke et problem hvis Q indeholder en process invokation.

Reglen (Newloc) danner en ny lokation, u, og tilføjer den til netværket.

(Out)	$\dfrac{[\![t]\!] = et}{l :: \mathbf{out}(t)@l'.P \mid\mid l' :: P' \longrightarrow l :: P \mid\mid l' :: P' \mid\mid l' :: \langle et \rangle}$
(In)	$\dfrac{[\![T]\!] = eT \;\; match(eT, et) = \sigma}{l :: \mathbf{in}(t)@l'.P \mid\mid l' :: \langle et \rangle \longrightarrow l :: P\sigma \mid\mid l' :: \mathbf{nil}}$
(Read)	$\dfrac{[\![T]\!] = eT \;\; match(eT, et) = \sigma}{l :: \mathbf{read}(t)@l'.P \mid\mid l' :: \langle et \rangle \longrightarrow l :: P\sigma \mid\mid l' :: \langle et \rangle}$
(Eval)	$l :: \mathbf{eval}(Q)@l'.P \mid\mid l' :: P'' \longrightarrow l :: P \mid\mid l' :: P \mid\mid Q$
(Newloc)	$l :: \mathbf{newloc}(u).P \longrightarrow l :: P[l'/u] \mid\mid l' :: \mathbf{nil}$
(Parallel)	$\dfrac{N_1 \longrightarrow N_1'}{N_1 \mid\mid N_2 \longrightarrow N_1' \mid\mid N_2}$
(Strukturel)	$\dfrac{N \equiv N_1 \;\; N_1 \longrightarrow N_2 \;\; N_2 \equiv N'}{N \longrightarrow N'}$

Figure 2.5: Operationel semantik for μKLAIM

2.2 Storm

Der har de seneste årtier været en revolution inden for databehandling. Datamængderne bliver hele tiden større og større og derfor skal der sendes, behandles og gemmes mere data. Man kan anvende Hadoop[Fou13a], MapReduce eller andre implementationer og de har gjort det muligt at behandle utænkelige mængder af data. Dog går vi en tid i møde, hvor teknologien betyder rigtig meget og jo hurtigere respons vi kan få jo bedre. Desværre er disse databehandlings implementeringer ikke tidstro - og det er heller ikke meningen. Denne type af databehandling kaldes batch-behandling og der findes ingen måde hvorpå man kan ændre Hadoop (eller lignende systemer) til at blive et tidstro system da et tidstro system har nogle helt andre krav. Tidstro systemer skal kunne garantere et svar inden for strenge tidsfrister. Det kræves ofte af et tidstro system at der

leveres et svar inden for millisekunder eller nogen gange helt ned til mikrosekunder.

Inden for bussiness intelligence branchen kræves det i stigende grad at man kan udføre interaktiv databehandling over store datamængder. Som tidligere nævnt mangler der et system som kan udfylde dette manglende hul med tidstro databehandling over store datamængder. For at håndtere tidstro databehandling så skal man manuelt bygge et netværk bestående af køer og arbejdere. Arbejderne står for at tage beskeder af en kø, opdatere databaser og sende nye beskeder til andre køer for yderligere databehandling. Der opstår en hel del begrænsninger ved at anvende denne metode, nemlig:

- Udviklingsdelen kan blive meget tung fordi man bruger lang tid på at konfigurere systemet og finde ud af hvor beskeder skal sendes hen, implementere arbejdere, samt implementere mellemliggende køer.
- Der er en lille fejl-tolerance og det betyder at man hele tiden skal sørge for at hver kø og arbejder er oppe at køre for ikke at få et alt for skrøbeligt system.
- Det er rigtig krævende at skalere, for så snart mængden af beskeder bliver for stor for en enkel arbejder eller kø, så skal dataen uddelegeres. Derfor skal man rekonfigurere de andre arbejdere således at de kender den nye lokation de skal sende beskederne til. Dette skaber flere nye scenarier som kan mislykkes.

Det grundlæggende paradigme for tidstro databehandling består i at kunne håndtere mange beskeder. Det er ikke muligt at håndtere et stort antal af beskeder med kø og arbejder paradigmet, da det vil bryde sammen.

2.2.1 Introduktion af Storm

Til at starte med er det vigtigt at nævne at Storm er gratis og open-souce og det kan hentes på http://storm-project.net/. Storm er et distribueret tidstro beregnings system, som gør det muligt at behandle ubegrænsede datastrømme. Det er muligt at anvende Storm med forskellige programmeringssprog. Der findes en række forskellige use cases, hvor der her i blandt kan nævnes: tidstro analyser, kontinuerlige beregninger, distribuerede RPC og mange flere. En vigtig egenskab for Storm er at det er hurtigt og en benchmark-test har klokket det til at lave mere end én million tupel behandlinger i sekundet pr. knude. Storm er

skalerbart, fejl-tollerant, garanterer at data altid bliver behandlet. En dybere gennemgang af Storms funktioner følger i dette afsnit.

Storm integrerer med køer og databaser som anvendes nutildags, her kan nævnes: Kestrel, RabbitMQ/AMQP, KAFKA og JMS. Det er muligt at få Storm til at integrere med nye kø eller datanase systemer. Man konfigurerer topologier, hvor en strøm af data bliver behandlet på vilkårlige komplekse måder med mulighed for at ompartitionere strømmene mellem de forskellige beregningstrin.

2.2.2 Storm klynger og topologier

For at lave tidstro beregninger skal man i Storm konfigurere topologier - hvor en topologi er en graf af beregninger. Hver knude i en topologi indeholder behandlingslogik og forbindelserne mellem knuderne[4] indikerer hvordan data bliver sendt rundt i mellem dem. I en Storm klynge har man to forskellige slags knuder: én masterknude og arbejdsknuder. Masterknuden kører en daemon[5] kaldet *Nimbus*. Nimbus sørger for at at distribuere kode rundt i klyngen, tildele opgaver til forskellige arbejdsknuder og kontrollere om der opstår fejl. Hver arbejds knude kører en daemon kaldet *Supervisor*. Supervisoren lytter på om der bliver tildelt noget arbejde til dens maskine for så at starte og stoppe en arbejder process hvis det er nødvendingt - arbejdet er baseret på hvad Nimbus har tildelt til maskinen. En kørende topologi består af mange forskellige arbejdsprocessor spredt over forskellige maskiner. Koordineringen mellem Nimbus og Supervisors bliver styret af en *Zookeeper*[Fou13b] klynge. Nimbus daemonens og Supervisor daemons fejlreporterer hurtigt og indeholder ingen states. De forskellige states er bevaret i Zookeeper eller på maskinens lokale harddisk. Det giver anledning til at man kan dræbe Nimbus eller Supervisors og de vil automatisk blive startet igen, som om intet var sket. Det medfører at en Storm klynge er meget stabilt.

2.2.3 Strømme

Nu introduceres kernen i Storm nemlig "streams" som på dansk kaldes strømme. Storm anvender tuples og en strøm er en ubegrænset sekvens af tuples som bliver sendt afsted. Med Storm tilbydes primitiver til at transformere en strøm til en ny strøm på en distribueret og pålidelig måde. Et eksempel her på kan være det som Twitter anvender Storm til, da de transformerer en strøm af tweets

[4] Startknuden i en topologi er altid en spout, hvorefter der er tilkoblet en eller flere bolts, hvor hver bolt indeholder behandlingslogikken. En uddybelse ses i 2.2.3

[5] En daemon er et program der kører som en baggrunds process, og den bliver styret direkte af en interaktiv bruger.

til en strøm af trending topics. Et trending topic anvendes på Twitter til at markere hvilke ord eller sætninger der oftest bliver anvendt af twitter-brugerne. På denne måde kan Twitter måle hvilke ord der hyppigst bliver anvendt. F.eks. op til præsident valg figurerer kandidaternes navne og det giver mulighed for Twitter at se hvad der rører sig i verden[6]

Til at håndtere transformationer af strømme anvendes der som de basale primtiver *spouts* og *bolts*. Spouts og bolts har et interface som man implementerer til at køre sin applikations specifikke logik på. En spout er en strøm kilde og den kan eksempelvis udtage tupler fra en Kestrel kø og udsende en strøm af tupler til de bolts den er forbundet til. Twitter anvender en spout til at tilkoble til Twitters API og derved udsendes en strøm af tweets - iform af tupler. En bolt tager et hvilket som helst antal af input strømme, behandler det input og udsender nye strømme. Når man vil have komplekse strøm-transformationer så kræver det multiple skridt og derfor laves en kombination af multiple bolts. Et godt eksempel er når man beregner en strøm af trending topics fra en strøm af tweets, så vil det blive gjort i flere skridt. En bolt kan gøre alt fra:

- Samle strømme
- Forbinde strømme
- Filtrering af tupler
- Snakke med databaser

Som tidligere nævnt består en topologi af et netværk af spouts og bolts, som er top-level abstraktionen som sendes til Storm clusteret for at blive eksekveret. Kanterne i grafen beskriver hvilke bolts som tilhører hvilken strøm. Når en spout eller en bolt udesender en tupel til en strøm, så udsender den tuplen til enhver bolt som tilhører dens strøm. Det betyder at kanterne i topologien beskriver hvordan tuples skal sendes rundt i grafen.

I figur 2.6 ses et eksempel på hvordan forbindelsen ser ud mellem noderne i en topologi. Forbindelsen mellem noderne indikerer hvordan tupler bliver sendt rundt. I eksemplet ses det at der haves en forbindelses mellem spout A og bolt B, en forbindelse fra spout A til bolt C og en forbindelse fra bolt B til bolt C. Hver gang spout A udsender en tupel bliver det både sendt til bolt B og bolt C. Yderligere sendes bolt b's output til bolt C på.

[6] Der kan også være en masse opmærksomhed i forhold til kendisser som Lady Gaga, Justin Bieber osv. pga. deres kæmpe skare af fans. Dog er algoritmen blevet modificeret så den tager højde for dette.

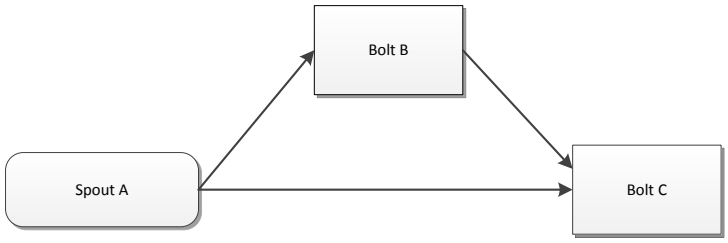

Figure 2.6: Forbindelsen mellem noder i en topologi

Hver knude i en Storm topologi ekseveres parallelt. Det er muligt at specificere hvor meget parallelisme man vil have for hver knude og Storm sørger for at antallet af tråde til eksekvering bliver fordelt i clusteret. En topologi vil køre forevigt eller indtil man dræber den. Fejler en opgave vil Storm automatisk tildele opgaven til en anden maskine.

2.2.4 Garantering af data behandling

Storm garanterer at en besked udsendt fra en spout altid bliver fuldt behandlet. I denne sektion vil jeg beskrive hvordan Storm opnår denne garanti og hvordan man som bruger anvender Storm til at opnå denne pålidelighed.

Først vil jeg starte med at forklare hvad det vil sige at en besked bliver fuldt behandlet. En tupel der kommer fra en spout kan medføre at tusindvis af nye tupler skal dannes, baseret på den. Vi betragter topologi-eksemplet hvor man tæller antallet af ord i en strøm af sætninger.

I topologien bliver forskellige sætninger aflæst fra en kestrel kø. Hver sætningen bliver splittet op i ord og der tælles hvor mange gange et ord har optrådt. En tupel som bliver udtaget fra spout'en udløser mange nye tupler baseret på den udtagne tupel. Der haves en tupel for hvert ord i sætningen, samt en tupel som fortæller hvor mange gange det pågældende ord har optrådt. I figur 2.7 ses et mddelelses træet for sætningen "Det er godt det er forår".

Storm betragter en tupel fra en spout som fuldt behandlet hvis tupel træet er tømt og alle beskeder i meddelelses træet er behandlet. Der anvendes en timer for at se om en tupel er timet ud - dvs. at hvis der sker en timeout bliver tuplen fejlet og Storm udsender tuplen igen. Det er muligt for brugeren at konfigurere hvor lang tid der skal gå før en tupel fejles. Man konfigurerer

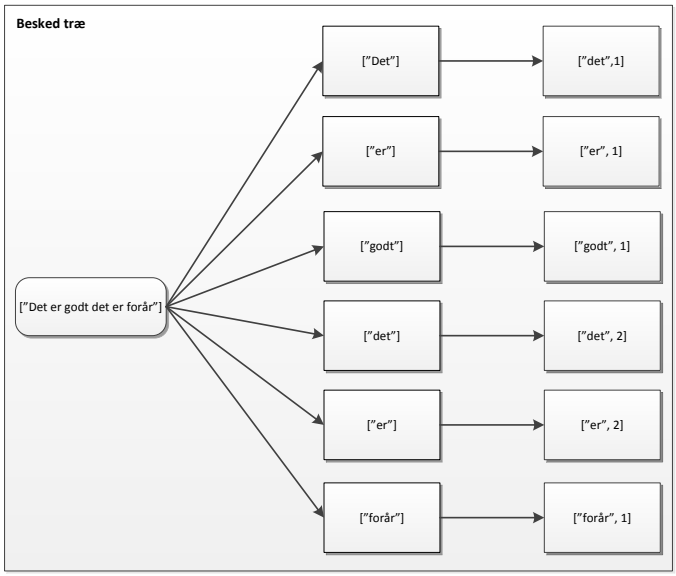

Figure 2.7: Meddelelses træet for sætningen "'Det er godt det er forår"'

topologien med kommandoen: TOPOLOGY_MESSAGE_TIMEOUT_SECS hvor standard værdien er sat til 30 sekunder.

2.2.5 En tuples livscyklus

For at se på en tupels livscyklus skal vi se hvad der sker hvis en tupel bliver fuldt behandlet eller hvis der sker en fejl under behandlingen når den forlader en spout. Interfacet for en spout har følgende metoder:

```
1 public interface ISpout extends Serializable {
2     void open(Map conf, TopologyContext context,
3         SpoutOutputCollector collector);
4     void close();
5     void nextTuple();
6     void ack(Object msgId);
7     void fail(Object msgId);
8 }
```

Til at starte med anmoder Storm om en tupel fra en spout ved at kalde metoden **nextTuple**(). Spout'en anvender **SpoutOutputCollector**, som er angivet i **open**() metoden til at udsende en tupel til en af dens output strømme. Når

spout'en udsender en tupel tildeles tuplen en *besked_id* der bruges til at identificere tuplen senere hen i forløbet. Når en *KestrelSpout* læser en besked fra Kestrel-køen, så bliver den aflæste besked tildelt en *besked_id* fra Kestrel som Storm anvender. En besked der udsendes til `SpoutOutputCollector`'en vil se ud på følgende måde:

```
1 _collector.emit(new Values("field1", "field2", 3) , besked_id);
```

Tuplen der sendes består af tre forskellige værdier, nemlig "$field1$","$field2$"$og3$. Tuplen sendes til den forbrugende bolt og Storm tager sig af at spore det dannede meddelses træ. Når en tupel er færdig behandlet vil Storm detektere det og kalder *ack*-metoden på den oprindelige spout-task med den besked_id som spout'en gav Storm. På samme måde vil Storm detektere en fejl hvis der opstår en time-out og **fail**-metoden vil blive kaldt på spout'en. Det vil altid være den spout der har udsendt tuplen der også kalder **ack**/**fail** metoden. Selvom en spout udfører en masse forskellige opgaver henover en klynge, så vil en tupel ikke blive bekræftet eller fejlet af en anden spout end den som udsendte den.

Når en besked bliver udtaget fra Kestrel køen, så bliver beskeden åbnet. En besked bliver ikke fjernet fra køen med det samme, men derimod bliver den placeret i en vente-tilstand, hvor den venter på en bekræftelse om at beskeden er færdig læst. Når en besked er i vente tilstand, så er det ikke muligt at sende beskeden til andre forbrugere af køen. På samme måde, så vil alle beskeder i vente-tilstand blive læst tilbage på køen for en klient hvis han logger af. Det gør at beskeder ikke bliver duplikeret. Når en besked er åbnet så sender Kestrel data, samt en unik besked-id til klienten. Senere, vil kestrelspouten så modtage enten ack eller fail og det videresender den til Kestrel sammen med besked-id'en og Kestrel fjerner beskeden fra køen(hvis der modtages ack) ellers sendes beskeden igen(hvis der modtages et fail).

2.2.6 Hvordan anvendes Storm's pålidelige API

I denne sektion forklares hvordan man benytter sig af Storm's pålidelige API. Det er vigtigt at man fortæller Storm hver gang man tilføjer en ny forbindelse i tupel-træet. Det er også nødvendigt at fortælle Storm hvornår du er færdig med at behandle en individuel tupel. Ved at gøre disse to ting, så er det muligt for Storm at detektere når tupel-træet er fuldt behandlet og derved godkende eller fejle den korrekte spout-tupel. Storms API er opbygget, så disse to opgaver kan udføres på en præcis måde.

Først skal man specificere forbindelserne i tupel træet og det kaldes forankring. Forankring betyder at man specificerer en ny forbindelse i tupel træet og det

skal gøres når en ny tupel skal udsendes. I en topologi er det første element altid en spout, og den er tilkoblet til én eller flere bolts, som den sender en strøm af tupler til. Bolts kan også være forbundet og for at sende en tupel videre til næste bolt anvendes emit-metoden. For at specificere en forbindelse i tupel træet, vha. forankring, så skal man i emit-metoden angive input tuplen som det første argument. Hvis den nyudsendte tupel fejles senere i topologien, vil den forankrede tupel blive genudsendt i roden af træet - altså fra spouten. Hvis man ikke forankrer en tupel, og den fejler i en efterfølgende behandling, så vil den ikke blive genudsendt fra spouten. I figur 2.8 ses to forskellige eksempler, hvor det ses at en forankret tupel der fejles vil blive genudsendt og en ikke forankret tupel bliver ikke genudsendt. Et system der ikke forankrer tupler vil ikke have nogen fejl-garanti. For at have fejl-tolerance i sit system er det derfor meget vigtigt at forankre de udsendte tupler. Det er dog muligt at undgå denne fejl-tolerance, da man sagtens kan forestille sig at have et system hvor det godt kan være hensigtsmæssigt at udsende ikke forankrede tupler.

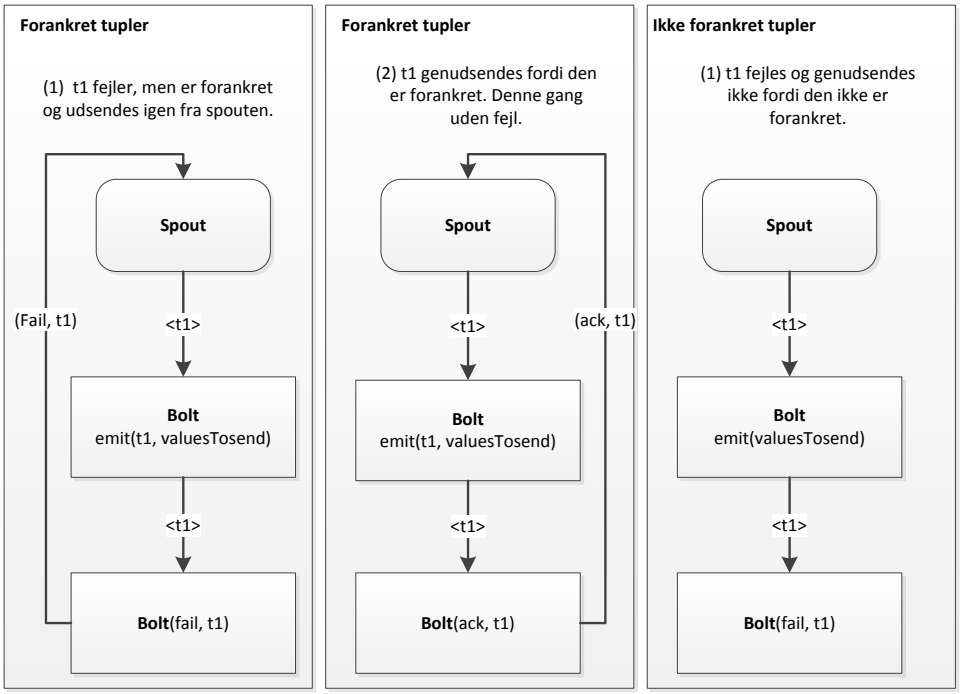

Figure 2.8: De to bokse til venstre viser et eksempel på en forankret tupel der fejles og bliver genudsendt. Boksen til højre viser en ikke forankret tupel der ikke genudsendes.

Nu til det andet trin i anvendelse af Storm's pålidelige API, hvor vi ser på

hvordan man specificerer hvornår man har færdig behandlet en individuel tupel i tupel træet. Det gøres ved at man bruger metoderne ack og fail. Når en tupel er færdig behandlet kaldes derfor ack metoden med input tuplen som argumentet. Det er muligt at bruge Storms indbyggede fail-metode til at fejle en tupel i roden af tupel træet. Det kan bl.a. anvendes hvis man har en applikation der skal fange en undtagelse fra en database klient og man fejler derfor tuplen med det samme. Ved at fejle tuplen med det samme, så vil spouten hurtigere genudsende en ny tupel i forhold til at vente på at en tupel timer ud. Det er vigtigt at alle tuples bliver bekræftet eller fejlet. Storm anvender hukommelse til at følge hver tupel, så hvis man ikke bekræfter eller fejler hver tupel, vil programmet på et tidspunkt løbe tør for hukommelse. Oftest følger en bolt et bestemt mønster:

1. Læs input tuplen
2. Udsend de tupler som afhænger af den
3. Bekræft tuplen i slutningen af execute-metoden

Man kalder denne slags bolts for filtre eller simple funktioner. I Storm er dette interface indbygget og man kan anvende dette mønster ved at udvide ens klasse med BasicBolt. Hvis man derimod anvender BaseRichBolt-klassen, så behøver man ikke at kalde ack-metoden med input tuplen i argumentet eller at forankre input tupler, da dette automatisk bliver gjort. Hvis man ved at man skal implementere et system hvor data altid skal behandles, så er det nemmere at anvende BaseRichBolts-klassen fordi man ikke skal tænke på godkendelse og forankring af tupler. Senere gennemgås det hvordan man sørger for at en tuplen kun bliver behandlet én gang, selvom den bliver genudsendt flere gange - i afsnittet transaktions topologi.

2.2.7 Multi-forankrede tupler

Det er muligt at forankre en output tupel til mere end én input tupel - det kan bl.a. bruges til sammenlægninger af strømme. En multi forankret tupel som bliver fejlbehandlet vil være skyld i at multiple tupler bliver genudsendt fra spoutsne. For at håndtere multiple forankret tupler, skal man lave en liste af tupler i stedet for at nøjes med en enkel og det gøres ved at multi-forankret tupler tilføjer output tuplen til multiple tupel træer. Ydermere er det muligt at lave DAGs vha. multi-forankret tuples og man bryder dermed træ-strukturen. Storm's implementering virker både for DAGs, såvel som træer.

2.2.8 Storm's pålidelighed på en effektiv måde

I denne sektion ses på hvordan Storm opnår sin pålidelighed på en effektiv måde. En Storm topologi har et sæt specielle bekræftelses jobs og deres opgave er at spore DAG'en af tupler for hver spout tupel. Når et bekræftelses job ser at en DAG er færdiggjort, så sender den en bekræftelses besked til spout jobbet(for den spout som oprettede tuplen). For at forstå Storms pålideligheds implementation studeres levetiden for tupler og tupel DAGs. Når en tupel dannes i en spout eller en bolt bliver den tildelt et tilfædigt 64 bits ID. Id'et anvendes til at spore tupel DAG'en for hver spout tupel.

En tupel kender id'et på alle spout tupels den er knyttet til. Hvis der udsendes en ny tupel i en bolt, så bliver spout tupel id'et fra den forankrede tupel kopieret til den nye tupel. En tupel bekræftes ved at sende en besked til det tilhørende bekræftelses job med information om hvordan tupel træet er ændret. Bekræftelses beskeden siger at den pågældende tupel er færdiggjort i træet for den tilhørende spout tupel, og videresender de nye tupler i træet som var forankret til tuplen.

I figur 2.9 ses et eksempel på hvordan tupel træet ændrer sig når en tupel bliver bekræftet som fuldt behandlet. Hvis vi ser på eksemplet, så bliver tupel D og E skabt på grundlag af tupel C. Figur 2.9 viser hvordan tupeltræet ændrer sig når C bliver bekræftet som fuldt behandlet. Tupel C bliver fjernet fra træet på samme tid som D og E bliver tilføjet. Det er vigtigt at nævne at selvom antallet af interne tupel behandlinger bliver mindre efter hver tupel behandling, så betyder det ikke at Storm har mulighed for at køre videre fra det fejlede stadie - hvis en tupel fejler eller timer-out. Hvis en tupel fejler informeres Storm og hele processen starter forfra, hvor tuplen bliver genudsendt fra dens tilhørende spout.

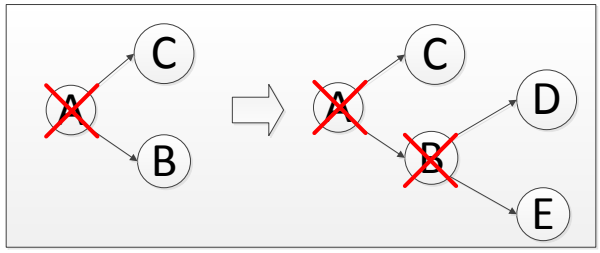

Figure 2.9: Eksempel af tupel bekræftelse i tupel træet

Fordi en spout kan indeholde forskellige bekræftelses jobs, så skal en udsendt tupel også vide præcis hvilket bekræftelses job den tilhører. Med brugen af

hashing, mapper Storm en spout tupels id til et bekræftelses job. Alle udsendte tupler indeholder spout tupel id'et og på denne måde vides hvilket bekræftelses job de skal kommunikere med. Når en spout udsender en ny tupel vil den sende en besked til bekræftelses jobbet og fortælle at dets job id har ansvar for den udsendte tupel. Når et bekræftelses job ser at et tupel træ er færdig behandlet, så ved den hvilket job id den skal videresende og fortælle at den er færdig behandlet.

2.3 En transaktions topologi

Nu introduceres kernen i projektet, nemlig transaktions topologier. En transaktions topologi gør det muligt at få udført en beregning præcis én og kun én gang. Det giver mulighed for at anvende STORM til at lave en præcis optælling. Transaktions topologier er et højere abstraktions niveau, bygget oven på Storm's primitiver af strømme, spouts, bolts og topologier.

2.3.0.1 Det simple design

Ideen bygger på at man vil opnå en stærk ordning[7] af behandlede data. Her vil den første idé være at udsende én tupel af gangen, og når denne er blevet fuldt behandlet i topologien, udsendes den næste. Denne metode er dog ikke særlig effektiv, da det kræver, vi skal vente på at en tupel er blevet færdigbehandlet, før vi kan udsende den næste. Hver tupel får tildelt et transaktions id. Hvis tuplen fejler og den skal udsendes igen, så bliver den udsendt med præcis det samme transaktions id. Et transaktions id er et tal som bliver inkrementeret for hver tupel. Det betyder, den første udsendte tupel starter med transaktions id 1, den næste 2, osv. Ved at bevare den stærke orden af tupler, opnår man præcis én gangs semantik - selv når den samme tupel bliver udsendt flere gange. Dette design er dog ikke særlig optimalt, fordi tupler behandles en af gangen og det giver en langsom behandlingstid.

2.3.1 Eksempel på én-gangs semantik

Vi ser på et eksempel, hvor det samlede antal af tupels i en strøm skal tælles. Idéen er at vi har en database, hvor en værdi gemmes. Værdien indeholder

[7] Med stærk ordning menes at tupler behandles i den rækkefølge de er udsendt. Dvs. at den første udsendte tupel behandles altid før den anden udsendte tupel osv.

antallet af optalte tupler samt transaktions id'et for den sidste behandling. Antallet af talte tupler skal kun opdateres, hvis transaktions id'et i databasen er forskelligt fra transaktions id'et for den tupel der behandles nu. Fordi vi har en stærk ordning af de udsendte tupler, og transaktions id'et i databasen er forskelligt i forhold til den nuværende tupels transaktion, så ved vi med sikkerhed at den nuværende tupel ikke er repræsenteret i optællingen. Derfor inkrementeres optællingen og transaktions id'et opdateres. Hvis transaktions id'et er det samme som det nuværende i databasen, så ved vi at tuplen allerede tilhører optællingen og derfor opdaterer vi ikke databasen. Det scenarie forekommer hvis tuplen er fejlet efter at have opdateret databasen, men før den har reporteret succes tilbage til Storm. Denne logik og den stærke ordning af transaktioner sikrer, at optælleren i databasen vil være nøjagtige, selv om tupler genudsendes. Dette trick med at gemme transaktions id'et i en database sammen med en værdi kommer fra Kafka udviklerne, helt præcist dette design dokument[n/a13].

Selv om man har en topologi med mange forskellige states er det stadig muligt at opnå præcis én gang behandlings semantik. Hvis en tupel fejler, vil de opdateringer der allerede er lykkedes, blive sprunget over. De opdateringer som har fejlet eller ikke er blevet udført, vil blive udført når tuplen bliver genudsendt.

2.3.2 Et bedre design

I stedet for kun at behandle én tupel af gangen er idéen nu at behandle et parti af tupler for hver transaktion. Hvis vi tager udgangspunkt i eksemplet fra før hvor man har en global optælling så inkrementeres optællingen med det antal af tupler, som et parti består af. Hvis et parti fejler, så vil man genudsende præcis det samme parti af tupler. I stedet for at hver tupel har sit eget transaktions id, tildeles et parti et transaktions id. På samme måde som før har vi nu en meget stærk ordning, men denne gang af partier. Hvis man behandler 100 tupler pr. parti vil ens program have 100 gange færre database operationer i forhold til det simple design. Ydermere udnytter dette design Storms paralleliserings kapacitet, da der er mulighed for at parallelisere behandlingerne for hver batch. I figur 2.10 ses et diagram af det bedre design - hvor man ser tupler blive sendt ind i topologien i forskellige partier.

Dette design er dog ikke helt optimalt, da det ikke er så ressourceeffektivt som muligt. Arbejderne i en topologi bruger meget tid på at idle, mens de venter på at andre dele bliver færdige. I figur 2.11 ses en topologi hvor det bedre design idler meget. Når bolt A har færdig eksekveret første parti, vil den stå og vente på at de andre bolts har eksekveret det, før det næste parti bliver udsendt fra spouten.

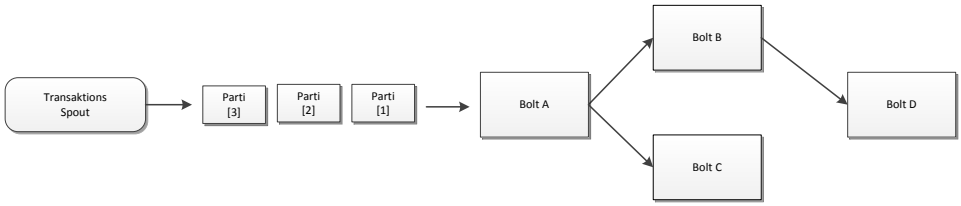

Figure 2.10: Diagram der illustrerer "'det bedre design"'

Figure 2.11: Figuren viser en topologi hvor "'det bedre design"' idler meget

2.3.3 Storm's design

En vigtig erkendelse er at det ikke er nødvendigt at have en stærk ordning af alt arbejdet, når et parti af tupler behandles. Man deler derfor en beregning op i to dele. Hvis vi tager udgangspunkt i eksemplet fra tidligere hvor den globale optælling beregnes, så haves følgende to dele:

1. Beregn antallet af tupler for et parti.
2. Opdater den globale optællings værdi i databasen med den beregnede værdi fra del 1.

Den første del behøver ikke at være stærkt ordnet og vi kan derfor pipeline beregningerne af partierne. Det er kun del 2, hvor der kræves en stærk ordning af partierne. Derfor kan det første parti arbejde på at opdatere databasen, mens parti 2 til 10 kan beregne antallet af tupler for deres parti. Storm deler derfor en beregning af et parti op i to:

- Behandlings fasen - denne del kan udføres i parallel for mange partier.
- Forpligtelses fasen - Forpligtelses fasen er stærkt ordnet. Med stærkt ordnes menes der at parti 2 ikke er færdigbehandlet før parti 1. Det vil altid være parti 1 der bliver færdigbehandlet først.

Sammenkoblingen af de to faser kaldes for en transaktion. Mange partier kan være i behandlings fasen, hvor der kun kan være et parti i forpligtelses fasen. Hvis der opstår en fejl i behandlings fasen eller forpligtelses fasen af et parti, så bliver hele transaktionen behandlet igen i begge faser.

Der er en række ting Storm selv udfører når man anvender transaktions topologier og heraf kan følgende nævnes:

- Storm holder styr på de forskellige tilstande i topologien - dvs. at Storm gemmer alle de nødvendige tilstande i Zookeeper. Dataen Storm gemmer, inkluderer det nuværende transaktions id samt den metadata der beskriver parametrene for hver udsendte parti.
- Storm sørger for at håndtere alt som er nødvendigt for at bestemme hvilken transaktion der skal behandles eller forpligtes.
- Når man anvender transaktions topologier, så vil Storm også benytte bekræftelses frameworket til effektivt at bestemme når et parti er blevet behandlet succesfuldt, succesfuldt forpligtet eller fejlet. Partier bliver derfor automatisk genudsendt fra spouten. Man behøver ikke at forankre eller bekræfte tupler i denne form for topologi.
- Storm lægger et API ovenpå de regulære bolts for at tillade behandling af tupler i partier. Yderligere håndteres koordineringen af hvilke jobs der har modtaget alle tupler for en bestemt transaktion. Til sidst ryddes op efter en hver akkumuleret tilstand for hver transaktion.

Til sidst er det meget vigtigt at nævne at det for transaktions topologier kræver, at den kø der anvendes skal kunne genudsende præcis det samme parti af tupler. Hvis kilden, spouten er koblet sammen med, ikke kan genudsende det samme parti af tupler vil præcis én gangs semantikken falde til jorden. Apache Kafka er bl.a. en teknologi man kan anvende som kilde for spouten.

2.3.4 Transaktions forsøgs objektet

Under anvendelse af transaktions topologier, anvendes et bestemt id hvilket i Storm kaldes for et transaktions forsøgs objekt. Id'et anvendes til at holde styr på de forskellige transaktioner. Det er vigtigt alle udsendte tupler i en transaktions topologi har transaktions forsøgs objektet som det første felt i en tupel. Det gør det muligt for Storm at identificere, hvilke tupler der hører til hvilket parti. Et transaktions forsøgs objekt indeholder to værdier: et transaktions id

og et forsøgs id. Transaktions id'et er unikt valgt for et specifikt parti og er det samme, uanset hvor mange gange et parti bliver genudsendt. Forsøgs id'et er unikt for et bestemt parti af tupler og anvendes så Storm kan skelne mellem tupler fra forskellige emissioner af det samme parti. Uden forsøgs id'et vil det være umuligt at skelne et genudsendt parti, med et tidligere udsendt parti. Transaktions id'et inkrementeres med 1 for hvert udsendte parti - startende med 1 og derfor får det næste parti transaktions id 2, osv.

2.3.5 Transaktions topologi API'et

I en transaktions topologi haves tre forskellige typer bolts. Den ene type kender vi fra den normale Storm topologi - kendt som en BasicBolt. BasicBolt er en bolt der ikke håndterer partier at tupler, og i modsætning blot udsender tupler baseret på en enkel input tupel. BatchBolt er en bolt der behandler et parti af tupler. Hver tupel i et parti kalder execute-metode og til sidst når alle tupler har eksekveret execute-metoden kaldes finishBatch-metoden. Det er derfor en god ide at lave en intern tilstandsrepræsentation, så man sørger for at alle tupler også håndteres i finishbatch, da den kun kaldes en gang for et parti. Det afhænger dog af, hvad man vil have at bolten skal gøre. Den sidste type bolts er en BatchBolt markeret med forpligtelser: den eneste forskel mellem denne type bolt og en regulær bolt er når finishBatch-metoden kaldes. En bolt markeret med forpligtelser kalder finishBatch-metoden under forpligtelses-fasen. Forpligtelses fasen garanteres kun at forekomme efter alle tidligere partier har forpligtet sig succesfuldt. Det vil blive forsøgt indtil alle bolts i topologien har færdiggjort forpligtelses fasen for hele partiet.

Der er stor forskel på behandlings fasen og forpligtelses fasen og for at forstå denne, ser vi på et eksempel af en topologi.

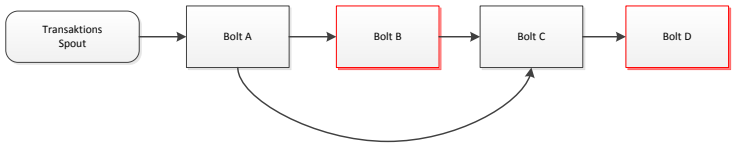

Figure 2.12: Topologi - behandlings fasen vs. forpligtelses fasen i bolts

I figur 2.12 ses en topologi, hvor bolts markeret med rød er forpligtelses bolts og de resterende er normale batchbolts. Under behandlingsfasen vil bolt A behandle hele partiet fra spouten, kalde finishBatch og sende tupler til bolt B og C. Bolt B derimod er en forpligtet bolt og den vil derfor behandle alle modtagede tupler, men finishBatch vil ikke blive kaldt - endnu. Ligeledes vil bolt C heller

ikke kalde finishBatch, da den ikke ved om den har modtaget alle tupler fra bolt B endnu. Bolt B venter på at transaktionen skifter til forpligtelses fasen. Bolt D vil modtage de tupler bolt C udsender under eksekvering af execute-metoden. Når partiet skifter til forpligtelses fasen kaldes finishBatch på bolt B. Når bolt B har færdig eksekveret, så er det muligt for bolt C at detektere at den har modtaget alle tupler, og derefter kaldes finishBatch. Til sidst modtager bolt D hele partiet og slutter af med at kalde finishBatch. Forpligtigelses bolts opfører sig præcis ligesom batchbolts under forpligtigelses fasen. Den eneste forskel, er at forpligtigelses bolts ikke kalder finishBatch under behandlings fasen i en transaktion.

Chapter 3

Mit bidrag

I dette kapitel gennemgås det subset af KLAIM der er tilpasset til at køre på ryggen af Storm. Først introduceres subsettet af KLAIM syntaksen med dens tilhørende semantik. Efter at have stiftet bekendtskab med KLAIM introduceres et højere abstraktions niveau af Storm - nemlig transaktions topologier. Transaktions topologier er et niveau bygget oven på Storm's primitiver af spouts, strømme, bolts og topologier og det anvendes til at sørge for at data kun behandles én og netop kun én gang.

Analyse, overvejelser samt diskussion af det implementerede system vil blive gennemgået og til sidst forklares de implementerede dele af systemet. I Storm er der udviklet to forskellige modeller, hvor model 1 garanterer at data altid bliver behandlet - dog med mulighed for sideeffekter. Model 2 fokuserer på at fjerne sideeffekterne og sørger for at alting behandles én og kun én gang.

3.1 Implementerede KLAIM subset

KLAIM subsettet der kan oversættes kaldes for stormKlaim hvor forkortelsen er sKLAIM. Her gennemgås den implementerede sKLAIM syntaks.

3.2 Syntaks og semantik

Syntaksen for sKLAIM kan ses i figur 3.1. Lokationer ses som netværks adressen for en knude, hvor en knude kan indeholde processer og tupler. For at få en forklaring af hvad syntaksen betyder se afsnit 2.1.4. Da min implementering er et subset af μKLAIM vil der ikke være nogle tilføjelser og derved er det blot en simplere variant.

N	::=		NETVÆRK	T	::=	$F \mid F, T$	TEMPLATE
		$l :: P$	*Enkel node*	F	::=	$f \mid !x \mid !u$	TEMPLATE FELTER
	$\mid$	$l ::< et >$	*Lokaliseret tupel*				
	$\mid$	$N_1 \mid\mid N_2$	*Sammensætning*	t	::=	$f \mid f,\ t$	TUPEL
				f	::=	$e \mid \ell \mid u$	TUPEL FELTER
P	::=		PROCESSER				
		$\emptyset$	*Inaktiv process*	et	::=	$ef \mid ef,\ et$	EVAL. TUPEL
	$\mid$	$a.P$	*Aktion præfiks*	ef	::=	$V \mid l$	EVAL. TUPEL FELT
a	::=		AKTIONER	e	::=	$V \mid x \mid ..$	UDTRYK
		$\mathbf{out}(t)@\ell$	*Tilføj*	ℓ	::=	$l \mid u$	NAVN
	$\mid$	$\mathbf{in}(T)@\ell$	*Fjern*				
	$\mid$	$\mathbf{read}(T)@\ell$	*Læs*				
	$\mid$	$\mathbf{newloc}(u)$	*Oprettelse*				

Figure 3.1: sKLAIM syntaks som kan oversættes til Storm

3.2.1 Strukturel kongruens

Strukturel kongruens, $\equiv$ identificerer et netværk som intuitivt repræsenterer det samme netværk. I figur 3.2 ses strukturel kongruens reglerne for sKLAIM. Reglerne i figur 3.2 opfylder den mindste kongruens relation i netværket. En forklaring af operatorne kan ses i afsnit 2.1.5.

$N_1 \mid\mid N_2$	$\equiv$	$N_2 \mid\mid N_1$	KOMMUTATIV
$(N_1 \mid\mid N_2) \mid\mid N_3$	$\equiv$	$N_1 \mid\mid (N_2 \mid\mid N_3)$	ASSOCIATIV
$l :: P$	$\equiv$	$l :: (P \mid \mathbf{nil})$	IDENTIFICER

Figure 3.2: Strukturel kongruens for sKLAIM

3.2.2 Matching mekanismen

Funktionen til at matche tupler for sKLAIM kan ses i figur 3.3. Matching mekanismen er det første skridt til at definere den operationelle semantik for sKLAIM. Pattern matching bruges til at vælge evaluerede tupler fra et tupelspace svarende til templaten. Da en template kan indeholde et udtryk skal disse evalueres, inden de kan bruges til at matche en evalueret tupel. En dybere gennemgang kan findes i afsnit 2.1.6.

$$(M_1)\ match(V, V) = \epsilon \qquad (M_2)\ match(!x, V) = [V/x]$$

$$(M_3)\ match(l, l) = \epsilon \qquad (M_4)\ match(!u, l) = [l/u]$$

$$(M_5)\ \frac{match(eF, ef) = \sigma_1 \quad match(eT, et) = \sigma_2}{match((eF, eT), (ef, et)) = \sigma_1 \circ \sigma_2}$$

Figure 3.3: Matching regler for sKLAIM

3.2.3 Reduktions relationen

Den sidste del af den operationelle semantik af sKLAIM består af reduktions relationen og disse kan ses i figur 3.4. Forklaring til de forskellige regler kan ses i afsnit 2.1.7.

3.3 Analyse

Ønsket fra Kapow Software er at der skal udvikles et system som sørger for en sekvens af behandlinger bliver udført én og kun én gang og hvor ingen transaktioner går tabt. Kapow anvender KLAIM til at beskrive selve opførslen af transaktionerne. Ved at implementere transaktionsbegreberne på ryggen af Storm får systemet robusthed og integritet, fordi Storm garanterer at data altid bliver behandlet. Det implementerede system bliver delt op i to modeller. I den første model (model 1) garanteres at data altid vil blive behandlet, dog er der mulighed for, at samme transaktion kan blive udført flere gange. Derfor udvikles model 2 hvor en transaktion altid kun behandles én og netop kun én gang. Data flyttes i mellem forskellige komponenter via multiple distribuerede tupel spaces, hvor tupel space implementationen der anvendes er SQLspaces.

$$\text{(Out)} \quad \frac{[\![t]\!] = et}{l :: \mathbf{out}(t)@l'.P \;||\; l' :: P' \longrightarrow l :: P \;||\; l' :: P' \;||\; l' :: \langle et \rangle}$$

$$\text{(In)} \quad \frac{[\![T]\!] = eT \;\; match(eT, et) = \sigma}{l :: \mathbf{in}(t)@l'.P \;||\; l' :: \langle et \rangle \longrightarrow l :: P\sigma \;||\; l' :: \mathbf{nil}}$$

$$\text{(Read)} \quad \frac{[\![T]\!] = eT \;\; match(eT, et) = \sigma}{l :: \mathbf{read}(t)@l'.P \;||\; l' :: \langle et \rangle \longrightarrow l :: P\sigma \;||\; l' :: \langle et \rangle}$$

$$\text{(Newloc)} \quad l :: \mathbf{newloc}(u).P \longrightarrow l :: P[l'/u] \;||\; l :: \mathbf{nil}$$

$$\text{(Parallel)} \quad \frac{N_1 \longrightarrow N_1'}{N_1 \;||\; N_2 \longrightarrow N_1' \;||\; N_2}$$

$$\text{(Strukturel)} \quad \frac{N \equiv N_1 \;\; N_1 \longrightarrow N_2 \;\; N_2 \equiv N'}{N \longrightarrow N'}$$

Figure 3.4: Operationel semantik for sKLAIM

For at få KLAIM beskrivelsen til at eksekvere i Storm kræver det en form for oversættelse af KLAIM koden. Til at starte med gives et indblik i de forskellige løsningsforslag, der har været oppe til overvejelse.

3.3.1 Løsningsforslag - problemer, idéer og diskussion

For at finde ud af hvordan man smartest oversætter KLAIM og udtrykker det i Storm, vil det være vigtigt at have et godt kendtskab til begge implementeringer og finde korrespondancen mellem de to systemer. KLAIM er eksekveret af operationer i en linæer form, hvor data eksempelvis udtrækkes fra en komponent og indsættes i en anden. Storm er en anelse anderledes, fordi en topologi består af en række sammenhængende komponenter, hvor data bliver udsendt fra den ene ende og strømmer til den anden. Hver komponent i Storm giver anledning til at udføre en eller flere operationer på de data der strømmer igennem den. De to systemer opfører sig derfor vidt forskelligt, og data bliver flyttet på helt forskellige måder. Dog er begge systemer forholdsvis lineære i deres behandling af data, hvilket giver anledning til en mulig 1 til 1 oversættelse. Med en 1 til 1 oversættelse tænkes at en bestemt KLAIM operation oversættes til en bestemt bolt i Storm domænet. Der implementeres derfor en bolt for hver KLAIM operation. Figur 3.5 giver et meget godt billede af oversættelsen. Det ses at parametrene fra KLAIM koden udtrækkes og anvendes når den mødte operation

gemmes i listen.

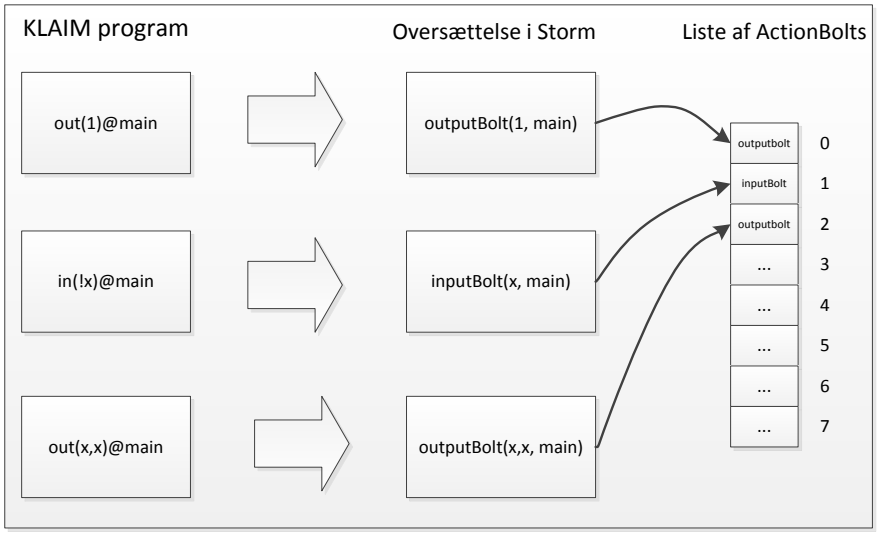

Figure 3.5: 1 til 1 oversættelse fra KLAIM program til Storm

KLAIM giver anledning til at tildele en variabel en værdi. I figur 3.1 ses sKLAIM syntaksen og det ses at to af operationerne giver mulighed for at tildele variabler værdi, nemlig in-operationen og read-. Hvis man vil tildele x en værdi skrives det på følgende måde: $in(!x)@main$. Helt konkret betyder det, at man søger efter en 1-tupel i main, og hvis der findes sådan en så tildeles x-variablen med værdien fra den fundne tupel. Når x har fået tildelt sin værdi kan man senere hen i programmet anvende den og udskrive den til en anden komponent med out-operationen - $out(x)@s1$. Dette er en af de fundamentale problemstillinger i oversættelsen fra KLAIM til Storm. Uheldigvis er det ikke muligt bare at overføre den udtænkte løsning fra model 1 til model 2, da eksekveringerne er vidt forskellige i de to design. Den grundlæggende idé er dog den samme og løsningen er, at man har et hashmap. Variablen gemmes som nøglen og variablens tilhørende værdi gemmes som værdien i hashmappet. Hvis man sørger for at opdatere hashmappet efter en endt in-operation, kan man nemt lave et opslag i hashmappet, når man senere skal bruge variablen og dens tilhørende værdi. Den præcise løsning for de to modeller bliver forklaret senere, da en forståelse kræver man har kendskab til de to forskellige design.

En anden problemstilling der skal op til overvejelse er i Storm regi. Skal oversættelsen af et KLAIM program ende ud i, at man danner én eller flere Storm topologier. Hvis man skal se på en løsning, hvor man danner flere topologier for en oversættelse af KLAIM programmet, så vil det være i en løsning, hvor

man eksempelvis sammensætter input-operationer og output- i en gruppe og eksekverer det i en topologi - ideen kan ses i figur 3.6. Ideen er at en input-operation oversættes til en Storm spout, hvor den læser data fra et tupel space og videresender de udtrukne tupels til den bolt den er koblet sammen med. Bolten udfører out-operationen og dens opgave er at modificere eller udtrække de variabler der skal outputtes til det angivne tupel space. Det er mulighed for at have en enkeltstående output-operation i KLAIM , hvor output-operationen ikke afhænger af andre operationer. En sådan topologi vil blot bestå af en spout der ikke udsender nogen tupler(altså en tom spout), hvor den hænger sammen med en bolt som udsender det ønskede output til et tupel space. I figur 3.6 svarer sådan en oversættelse til eksemplet for topoligi 1.

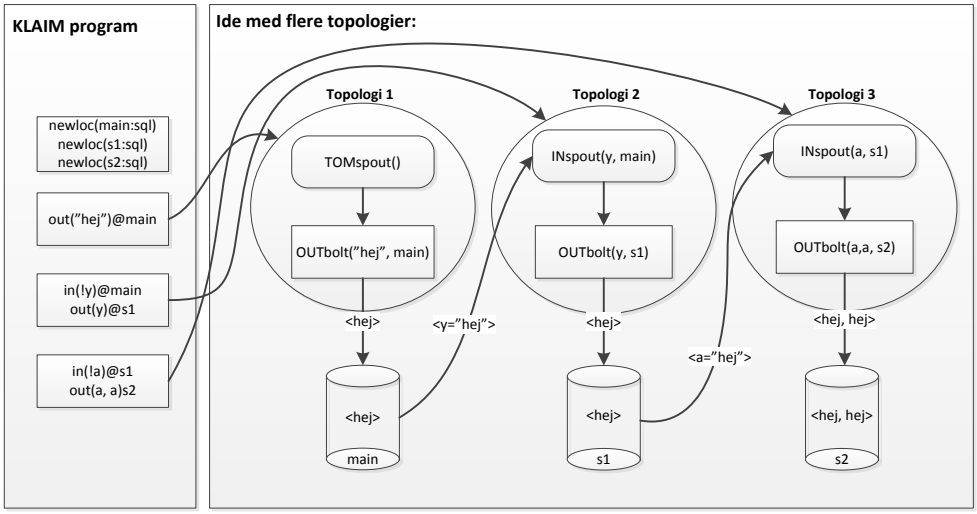

Figure 3.6: Ide med gruppering af operationer, hvor man så danner en topologi for hver gruppering

Der er nogle negative sider ved denne løsning, da det kræver at KLAIM brugeren har stor indsigt i præcis hvordan KLAIM koden skal opskrives for at kunne oversættes - han skal sammensætte input og output operationerne i grupper. Det fjerner en del af fleksibiliteten, når man laver et KLAIM program. Det bliver også et problem mht. brugen af tildelte variabler. Ser man på KLAIM programmet i listing 3.1, vil der opstå et problem med at håndtere de variabler, der tildeles en værdi i linje 4, da de skal outputtes i både linje 5 og 6. Linje 3 og 4 vil være at finde i samme topologi, men det vil linje 5 ikke være i denne løsningsmetode. Det er derfor ikke muligt at opdele et KLAIM program til flere topologier, hvis det indeholder afhængighed. For at kunne danne flere topologier skal programmet først analyseres, for at se om det er muligt at lave en opdeling. Det vil være svært at autogenerere topologierne på en simpel måde, og der vil

være mange specielle scenarier der skal tages i betragtning. På baggrund af disse problemstillinger er det bedst, at en oversættelse kun sker til én topologi.

```
1    newloc(main:sql).
2    newloc(s1:sql).
3    out("foo", 1337)@main.
4    in(!x, !y:int)@main.
5    out(x,y)@s1.
6    out(x)@s1. Nil
```

Listing 3.1: Eksempel på et KLAIM program

Der skal også være mulighed for at systemet kan indeholde en service. En service dannes for et tupel space, hvor servicens opgave er at modificere de tupler der outputtes til tupel spacet. Det betyder, at en service kun berør output operationerne ud af de fire sKLAIM operationer. Muligheden for service håndtering er tilføjet til Storm implementeringen for at give mulighed for at berige en tupel, så systemet bliver mere praktisk orienteret. I min afhandling er de dannede services meget simple. En service kan f.eks. være, hvor man outputter en tupel med et enkel felt, hvor servicen tager tuplen og danner nye felter og gemmer det i tupel spacet. I praksis kan man tænke sig at servicen anvender tupel feltet til at søge mere information på nettet og derved berige tuplen med flere oplysninger. Senere vil der blive forklaret eksempler på hvordan services kan anvendes. Service-delen er et ønske fra Henrik Pilegaard fra Kapow og det berører ikke varianten af KLAIM.

Efter at have forstået KLAIM semantikken og arbejdet med den har, jeg analyseret mig frem til at en fortolkning af KLAIM, bedst og nemmest oversættes til en enkel Storm topologi - da det giver en større frihed. Et KLAIM program er lineært og efterhånden som det gennemløbes mødes forskellige variabler, hvis værdier skal gemmes så de kan anvendes senere. Hvis en allerede eksisterende variabel mødes, skal dens værdi blot opdateres. Et eksempel på hvordan ideen med at oversætte et KLAIM program til en enkel topologi kan ses i figur 3.7.

En enkel Storm topologi vil eksekvere et oversat KLAIM program hurtigere end flere forskellige topologier. Grunden er at et KLAIM program med eksempelvis 20 forskellige input og output operationer, skal danne 10 topologier med en spout og en bolt. Og med 1 til 1 oversættelsen vil man danne en topologi med 20 bolts. Når en Storm topologi startes vil den bruge noget tid på at starte op - selvfølgelig kan man antage at 10 topologier starter op på samme tid som en enkel, men det vil ikke være tilfældet i praksis. Derfor vil hastigheden for behandlingerne ved en 1 til 1 oversættelse også klart være at foretrække - da Storm bruger en del tid på at starte en topologi op. Ydermere vil man ved en topologi med mere end en bolt kunne få Storm til at parallellisere behandlingerne, og dermed behandle mange tupler af gangen.

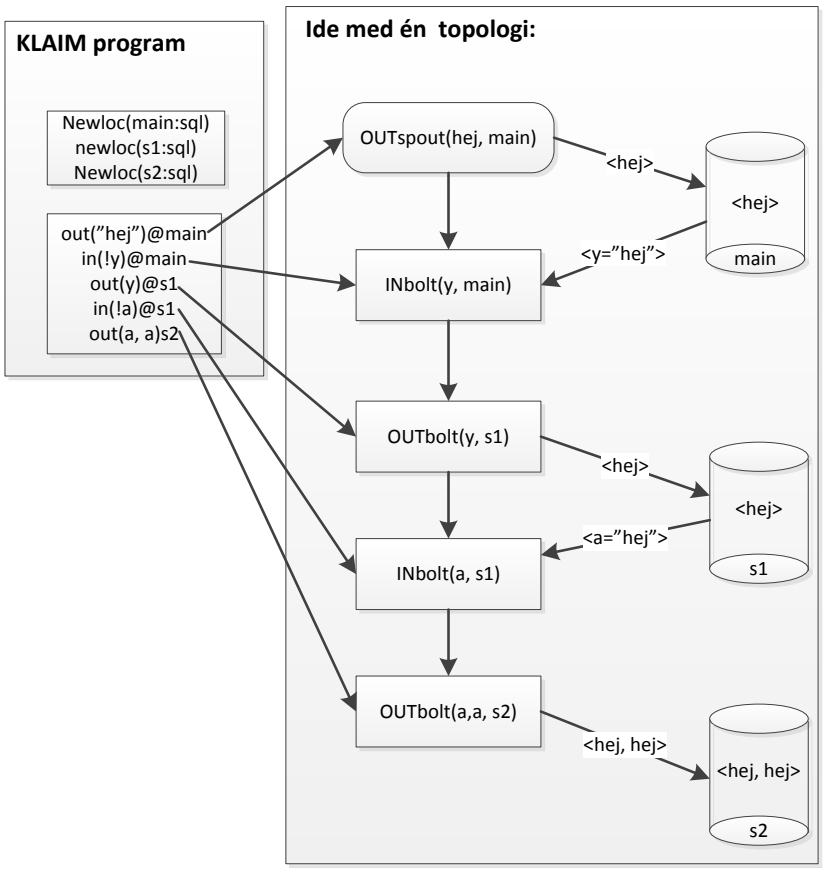

Figure 3.7: Ide hvor et KLAIM program oversættes til en enkel topologi

3.4 Overblik

Nu er der blevet diskuteret en hel del om forskellige løsningsforslag, men hvad er det vi i realiteten ønsker. KLAIM implementeringen giver en form beskrivelse af hvordan data skal flyttes igennem et system bestående af tupel spaces. Konkret sagt betyder det, hvordan flytter vi tupler fra et tupel space til et andet, hvor KLAIM anvendes til at programmere det distribuerede systemer. I sig selv fungerer det perfekt, men set i et større perspektiv ønskes det i denne afhandling at operationerne udføres tidstro, og man kan garantere at operationerne altid behandles. Det næste ønske er, at man også kan garantere at en operation kun udføres én gang for at undgå sideeffekter. Til at implementere opførslen af tupel space operationerne anvendes SQL-space implementeringen. Det betyder, at vi ønsker at oversætte KLAIM operationerne og på en eller anden måde skal tupel space operationerne isoleres i Storm. Storms egenskaber skal anvendes til at sørge for, at tupel space operationerne altid bliver behandlet, og at det kun sker én gang. Ved at anvende Storm topologier kan vi garantere, at data altid bliver behandlet, for hvis en tupel fejler registrerer Storm det, og genudsender den, og det er netop det den første model bygger på.

Vi ønsker at gå et skridt videre og garantere at SQL-space operationerne kun udføres én og netop én gang. Med Storms transaktions topologier loves der at data altid kun behandles én gang og det er hvad vi ønsker med model 2. Igen skal vi have oversat KLAIM operationen og sørge for at få isoleret SQL-space operationen, så den kun bliver behandlet én gang, da vi ikke ønsker, at den samme tupel bliver tilføjet til et tupel space flere gange.

3.5 Oversættelse af KLAIM

KLAIM implementationen er udviklet af Henrik Pilegaard fra Kapow Software. Han har udleveret koden til mig, og den har jeg brugt til at arbejde videre med, så jeg kan anvende den i min implementering. Koden er blevet modificeret, så den passer til mit behov. Afhandlingen omhandler ikke parsing, hvorfor parsingdelen ikke er ændret fra den udleverede implementering. Derfor bliver gennemgangen ikke særlig dybtgående fordi det ligger udenfor fokus området. I parsingdelen kaldes forskellige klasser når operationer mødes, og der oprettes en ny klasse passende til den operation der mødes, hvor værdier og variabler oprettes sammen med klassen.

Ideen er at den nødvendige information om en operation skal gemmes, i en oversættelses liste, i den rækkefølge operationerne mødes. Den nødvendige informa-

tion udtrækkes under evaluering af hver operation, hvor der er implementeret en funktion, der tilføjer den nødvendige data til oversættelses listen. Et eksempel på hvordan hver operationerne fra et KLAIM program gemmes i en liste efterhånden som de mødes ses i figur 3.8. Efterhånden som KLAIM programmet parses af KLAIM implementeringen evalueres hver operation og det er her at alt den nødvendige information gemmes. På denne måde opbygges en liste med information om hver operation fra KLAIM programmet i præcis den række følge de er mødt.

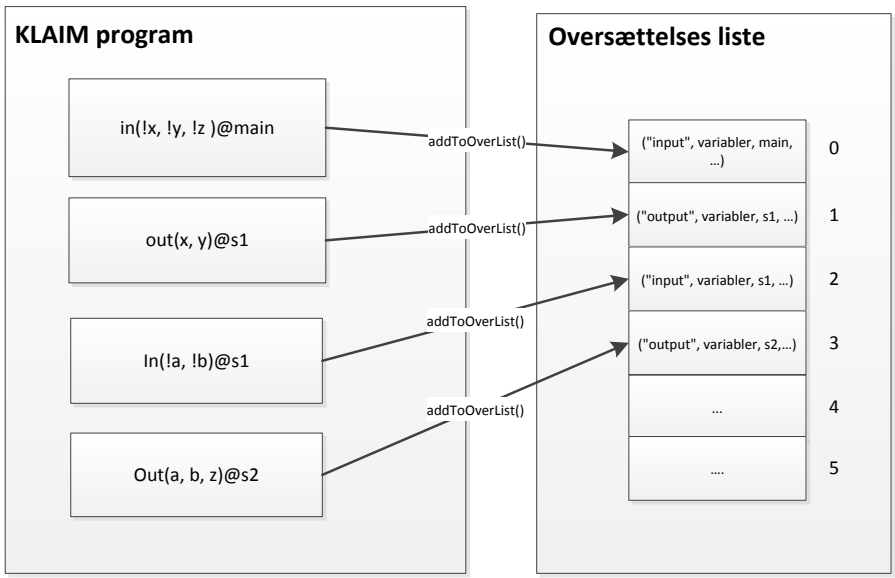

Figure 3.8: Eksempel der viser hvordan operationerne gemmes i en liste efterhånden som de mødes i KLAIM programmet.

Der er implementeret en klasse, ActionBolt.Java, til at gemme informationerne om de forskellige operationer. Klasse beskrivelsen ses i figur 3.9. Nu haves en liste med en lineær beskrivelse af KLAIM programmet, og den liste skal anvendes til at opbygge Storm topologi.

En operation i KLAIM skal direkte oversættes til en bestem type bolt i Storm, hvor boltens opgave er at udføre netop denne KLAIM operation i form af at udføre den passende SQL-space operation. ActionBolt-klassen er implementeres så den gemmer alt det data vi skal bruge for at implementere Storm topologien. Som det ses i figur 3.9 haves fem forskellige private felter:

1. **_actionname -** beskriver typen af KLAIM operation.

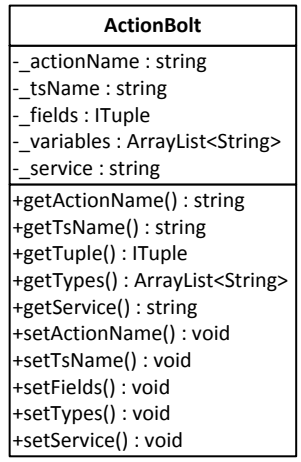

Figure 3.9: Klasse beskrivelse af klassen ActionBolt.java

2. **_tsName** - holder navnet på det tupel space der manipuleres.
3. **_fields** - gemmer de variabler der bliver anvendt i operationen.
4. **_variables** - er en liste der anvendes til at gemme de variabler som bliver anvendt i den specifikke operation.
5. **_service** - anvendes til at gemme tupel space knudens type (det kan være sql eller en service).

_variables gemmer for en input-operation hvilken data type der skal bruges til at søge efter i tupel spacet.

3.6 Model 1

I model 1 er det muligt at garantere, at data altid bliver behandlet. I KLAIM sammenhæng betyder det, at alle operationer bliver eksekveret minimum en gang. Hvis en tupel udsendes fra spouten og fejler på dens vej igennem systemet, vil den blot blive genudsendt, og operationerne har derfor mulighed for at blive eksekveret flere gange. I figur 3.10 ses et eksempel på model 1's bekræftelses og fejlfindings mekanisme. I figure udsender spouten $t1$, og videresender den til INbolt, hvor den bliver bekræfter og derfor videresendes til den næste bolt i topologien, nemlig OUTbolt. Her bliver tupel, $t1$, fejlet og spouten udsender

derfor $t1$ igen. Denne gang bliver den fejlet i tredje bolt og spouten udsender igen tupel, $t1$. Denne gang lykkedes det at få sendt tuplen hele vejen igennem topologien. Den første bolt har eksekveret $t1$ tre gange og den anden bolt har eksekveret tupel, $t1$, to gange. Vi tillader denne situation for nu, da ønsket i første omgang er at sikre, at data altid bliver behandlet.

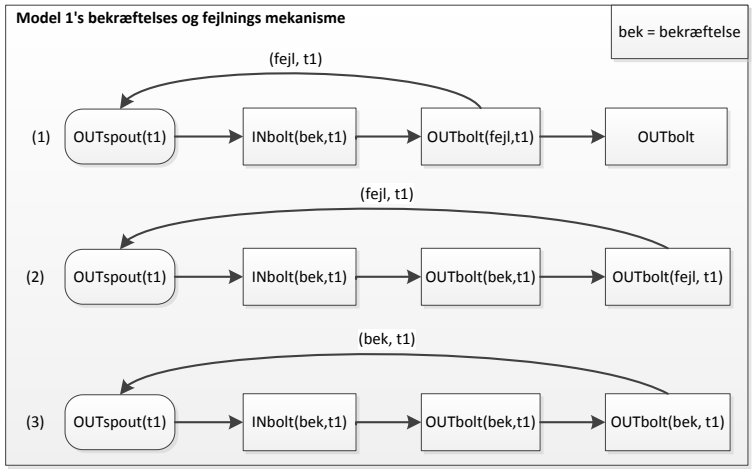

Figure 3.10: Figuren beskriver model 1's bekræftelses/fejlfindings system, når en tupel bliver genudsendt og det garanteres at den altid bliver behandlet.

Vores sKLAIM syntaks fortæller os at der haves fire forskellige operationer. Dog er det kun tre af dem, der skal anvendes i oversættelsen, da det er disse der flytter data. Output operationen giver anledning til to forskellige slags bolts, da der er forskel på om man outputter til et tupel space der indeholder en service, eller man outputter til et helt normalt SQL-space uden en service. Der findes ikke nogen direkte oversættelse af newloc, men det er netop her det bliver afgjort, om et tupel space indeholder en service eller ej. Newloc anvendes til at oprette nye lokationer i KLAIM. KLAIM implementationen tjekker, at lokationer der anvendes i input og output eksisterer og er oprettet. Anvendes en lokation som ikke er blevet oprettet, vil KLAIM implementationen melde fejl og KLAIM programmet kan derfor ikke oversættes. I figur 3.11 ses hvilke Storm bolts og spouts de forskellige KLAIM operationer oversættes til. Tabellen er for model 1, da det giver nogle lidt andre bolts/spouts for model 2.

Når et program startes er alle tupel spaces tomme fra start af og indeholder derfor ingen tupler. Derfor er det antaget, at den første operation altid er en output-operation, så der kan blive sat gang i systemet. Det ville være umuligt at starte med en input-operation, da den forgæves ville stå og søge efter en

(Out)	$out_i(T)@ts$	$\longrightarrow$	OUTspout(T, ts)	if $i = 0$
(Out)	$out_i(T)@ts$	$\longrightarrow$	OUTbolt(T, ts)	if $i > 0$ && if (ts==sql)
(Service)	$out_i(T)@ts$	$\longrightarrow$	SERVICEbolt(T, ts)	if $i > 0$ && if (ts==service)
(In)	$in_i(T)@ts$	$\longrightarrow$	INbolt(T, ts)	if $i > 0$
(Read)	$read_i(T)@ts$	$\longrightarrow$	READbolt(T, ts)	if $i > 0$

Figure 3.11: Tabellen viser hvad KLAIM operationerne bliver oversat til.

matchende tupel. Det er derfor nødvendigt, at et KLAIM program altid starter med en output-operation og i en Storm topologi er det første element altid en spout, hvor tupler udsendes fra. Derfor skal den første operation være en output og den oversættes til en spout i Storm domænet. Listen med beskrivelsen af KLAIM programmet udrulles og linæert opbygges topologien ved at de forskellige KLAIM operationer oversættes direkte til passende bolt typer, i overensstemmelse med figur 3.11. Bolts bliver forbundet i den lineære rækkefølge KLAIM operationerne er mødt under opbygning af oversættelses listen.

Udover de tre forskellige bolt typer - der direkte er oversat - er der også implementeret en ekstra funktion. Med denne ekstra funktion, er det muligt at danne en service for et givent tupel space. Den service jeg har implementeret modtager 1-tupler og danner 4-tupler. Når et tupel space har en service, er det kun muligt at outputte 1-tupler til tupel spacet. Servicen vil danne tupler med fire felter ud fra feltet fra 1-tuplen. Ved at udtrække feltet fra 1-tuplen og derefter lave en tupel med 4 nye felter, hvor de nye felter består af det oprindelige indhold med en tilføjelse af hhv. 0, 1, 2, 3. En illustration kan ses i figur 3.12, hvor 1-tuplens felt indeholder en streng med værdien Storm.

Figure 3.12: Et eksempel på hvordan servicen danner en 4-tupel

I tupel spacet haves derfor kun tupler med fire felter. Dette er blot en illustration af, at man kan outputte en tupel med et felt, og pga. servicen giver det mulighed for at berige en tupel med noget ny information - og derfor får man nu fire felter i den ny dannede tupel der gemmes i tupel spacet. Skulle man gå videre med service funktionen, så kunne man tænke sig at servicen modtager tuplen, og anvender informationen i feltet til at lave et opslag på internettet, og på denne

måde berige tuplen med ny information der er fundet ud fra den outputtede tupel.

Et eksempel på en oversættelse af et KLAIM program til en Storm topologi kan ses i figur 3.13. Det skal lige pointeres at ikke alle argumenterne der anvendes i kildekoden vises i eksemplet - kun de vigtigste er medtaget for at simplificere figuren. Tupel space, s1, indeholder en service hvor der dannes en ny tupel med fire felter i stedet for et enkel.

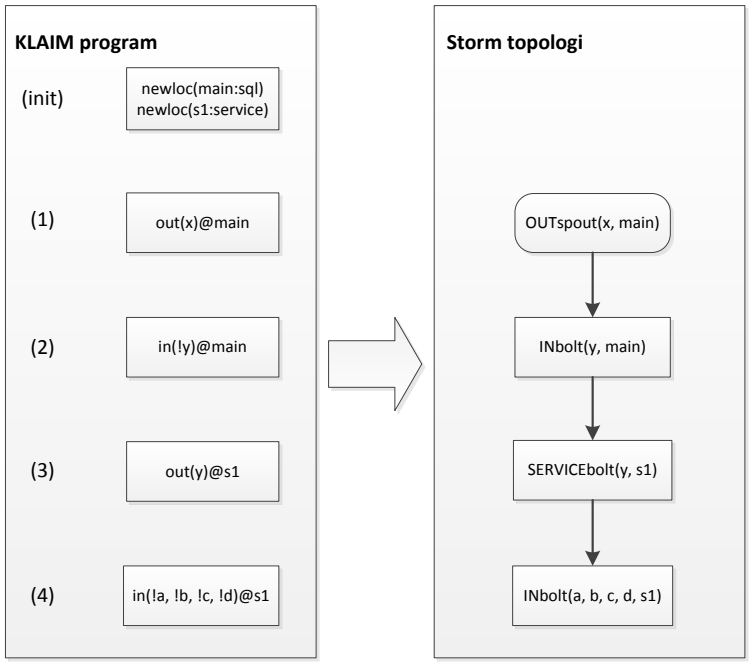

Figure 3.13: Et eksempel på hvordan Storm topologien ser ud for et oversat KLAIM program

Det ses i figuren at den tredje operation, out(y)@s1, er en output operation, men den bliver oversat til en SERVICEbolt. Der haves to forskellige typer SQL-space: SQL-space med service og et SQL-space uden service. Når en ny lokation oprettes, er der mulighed for at vælge hvilken type SQL-space der skal dannes, jævnfør init-delen i figur 3.13 hvor man kan se oprettelse af de to forskellige spaces.

3.6.1 Tildelte variabler

Til sidst vil jeg for model 1 forklare, hvordan jeg håndterer værdierne, en variabel får tildelt i KLAIM programmet. I KLAIM er det muligt at tildele en variabel en værdi, via read eller in operationerne, og når en variabel får tildelt en værdi, er det muligt at outputte den senere i programmet. Som tidligere nævnt er ideen at man har et hashmap, hvor variablen gemmes som nøglen, og den tildelte værdi gemmes som den tilhørende værdi i mappet. Variabler kan altid kun have tildelt en værdi, og derfor passer et hashmap perfekt til at holde styr på, hvad værdien er for en variabel.

I Storm udsendes en tupel fra spouten og da topologien er lineært sat sammen vil den udsendte tupel besøge alle bolts på dens vej igennem topologien. Med denne egenskab er ideen derfor, at oprette et hashmap i spouten og at det udsendes sammen med tuplen - man pakker hashmappet ind i tuplen. Efterhånden som nye variabler får tildelt nogle værdier, bliver mappet opdateret. Storm gør det rigtig nemt at medsende mappet, da Storm tupler, der sendes igennem systemet, faktisk er en liste af objekter, og mappet skal blot tilføjes til denne liste. Ved at man sender mappet sammen med Storm tuplen, så er det muligt at udtrække mappet i en bolt, opdatere det, og videresende det opdaterede map til næste bolt.

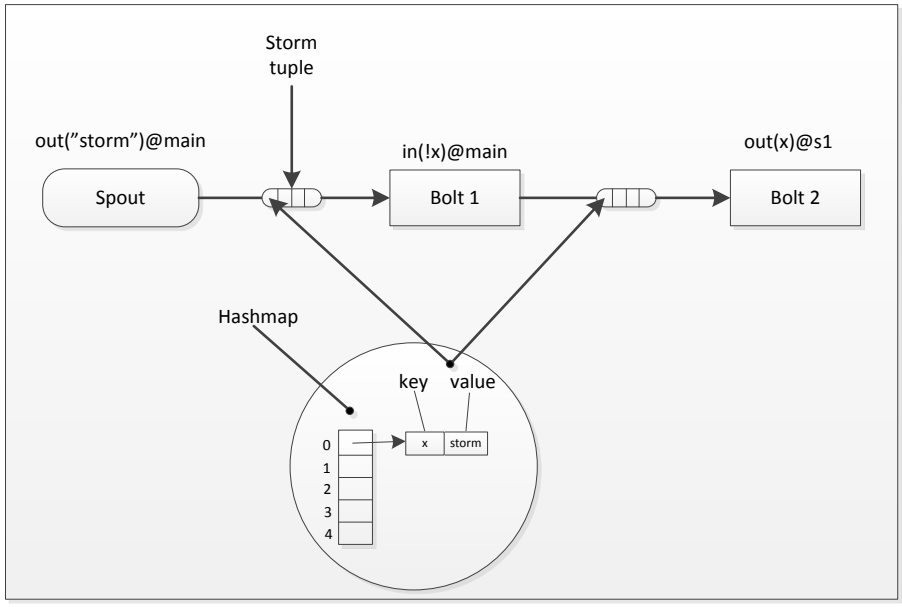

Figure 3.14: Et eksempel på hvordan hashmappet sendes rundt i en topologi

I figur 3.14 ses det hvordan et hashmap sendes imellem komponenterne i en Storm topologi via en Storm tupel.

3.7 Model 2

Med model 2 er det muligt at garantere at data altid eksekveres én og kun én gang. Ligeledes skal det nævnes, at det også garanteres at data altid bliver eksekveret. Det vil sige at KLAIM operationerne kun udføres én og netop én gang. Ligesom model 1 vil en tupel udsendes fra en spout og fejler den på vej igennem systemet, vil den blive genudsendt. I denne model ønsker vi at holde styr på om en operation er blevet udført før, så en genudsendt tupel ikke bliver behandlet flere gange. Da Storm er bygget op, ved at man altid genudsender tupler fra starten, hvis de er fejlet, så er ideen, at en bolt hvor den genudsendte tupel allerede er blevet behandlet skal springes over. Til det formål anvendes Storms transaktions topologier, da de lover at en behandling kun udføres en gang.

En vigtig ting at nævne med transaktions topologier er at tupler udsendes i et parti, i stedet for at en spouten udsender en enkel tupel af gangen. Derfor er det også et helt parti af tupler der vil blive udsendt igen, i tilfælde af at en tupel fejler. Hvert parti får tildelt et transaktions id der bruges til at skelne mellem forskellige udsendte partier. Under introduktionen af transaktions topologier blev det nævnt at en bolt nu er delt op i to forskellige faser. Den første fase er behandlingsfasen og disse operationer kan behandles parallelt hen over topologien. Den anden fase er en forpligtelses fase, og pga. den stærke ordning af partierne behandles data kun én gang, selv hvis et parti udsendes igen. Forpligtelses fasen behandler partierne i en ordnet rækkefølge, dvs. at et parti med et højere transaktions id ikke kan behandles før et parti med et lavere.

I denne model bliver vi derfor nødt til at tænke over, hvilke dele der ikke behøver at være stærkt ordnede og de dele der kun må behandles en gang. Behandlingsfasen er forskellig for en output operation og en input operation. For en output operation skal vi blot finde ud af hvor mange tupels det følgende parti består af og det gemmes lokalt og anvendes i forpligtelses fasen, når det er tid. I behandlingsfasen gemmes også lokalt hvilket tupel space den oversatte KLAIM operation opererer på. De implementerede bolts der er implementeret for input og read er stort set ens, bortset fra at man med en read-bolt læser fra tupel spacet, og for output bliver de udtrukne tupler fjernet. Derfor er de to faser fuldstændig ens og under behandlingsfasen undersøges det, hvilken type tupler der skal matches efter i tupel spacet. Hvert matchende tupel i et parti gemmes lokalt for at kunne bruge det senere i forpligtelses fasen. Tilstanden styres netop

ved at tupler gemmes lokalt for hver bolt.

Oversættelsen af KLAIM koden gøres på fuldstændig samme måde som model 1. Når der anvendes en transaktions topologi, giver det anledning til fire forskellige typer basis bolts og en speciel bolt. De bolts der dannes ud fra KLAIM programmet kan ses i figur 3.15. Igen er det de operationer fra sKLAIM der flytter data, der skal oversættes. For output operationen giver det anledning til to forskellige typer bolts. Hvis tupel spacet der outputtes til indeholder en service, skal service bolten anvendes ellers anvendes en normal out bolt. Newloc operationen har ingen direkte oversættelse i model 2, men fungerer på præcis samme måde som før, hvor det med newloc afgøres om et SQL-space indeholder en service eller ej.

(Out)	$out(T)@ts$	$\longrightarrow$	OUT_tbolt(T, ts)	if (ts==sql)
(Service)	$out(T)@ts$	$\longrightarrow$	SERVICE_tbolt(T, ts)	if (ts==service)
(In)	$in(T)@ts$	$\longrightarrow$	IN_tbolt(T, ts)	
(Read)	$read(T)@ts$	$\longrightarrow$	READ_tbolt(T, ts)	

Figure 3.15: Tabellen viser hvad sKLAIM operationerne bliver oversat til i model 2

I en transaktions topologi er det et krav at en transaktions spout har den egenskab at den har mulighed for at udsende præcis det samme parti af tupler. Derfor indgår spouten ikke i oversættelses tabellen for model 2 og der laves en speciel håndtering.

Model 2 er en del anderledes end model 1 og det skyldes at transaktions spouten i en transaktions topologi er meget anderledes fra en normal spout. Transaktions spouten lytter på et map for at se om det indeholder data, hvis det er tilfældet bliver det udsendt i form af en tupel. Storm sørger for at sende tupler af sted i partier indtil der ikke findes mere data i mappet. Man kan løbende tilføje ny data til mappet og det vil derefter blive sendt ud i topologien via. spouten. Model 2 er opbygget, så spouten altid først sender tupler til en start bolt, hvor de udtrukne tupler bliver lagret i et tupel space - med en output operation. Det betyder at selve KLAIM programmet starter her fra, hvor det vides at tupler er lagret i et tupel space kaldet main. Grunden til at systemet er opbygget på denne måde er, at man så starter med at udtrække tupler og derved starter sit KLAIM program med at lytte. Det passer godt overens med den use case

der vil blive anvendt, hvor man lytter på en mail-boks om der er kommet nogle nye emails. Emails vil blive udtrukket og behandlet efterfølgende ved hjælp af de operationer brugeren har implementeret i sKLAIM. I figur 3.16 ses at der er lavet en opdeling, hvor man kan se at initialiseringsdelen består af nogle steps, for at få transaktions topologien igang. Initialiseringen er nødvendig for at vi får ændret systemet til netop den måde, vi ønsker det skal opføre sig på. Det er muligt at integrere spouten så den lytter på andre slags databaser, men databasen der anvendes skal garantere at et parti af tupler kan blive genudsendt. Ellers vil man ikke kunne garantere at data altid behandles én og kun én gang.

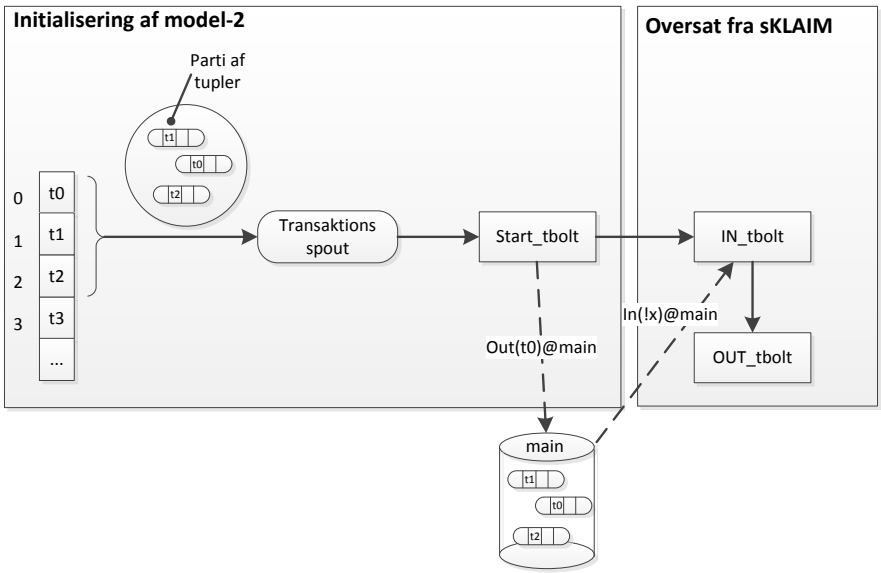

Figure 3.16: I figuren ses initialiseringsdelen for model 2

3.7.1 Behandling kun én gang

Med model 2 er det muligt at garantere at data kun bliver eksekveret én gang på den samme bolt. Trikket er, at man har et hashmap, hvor nøglen er boltens hash-code, da den er unik for hver bolt i topologien. Værdien der gemmes sammen med nøglen er det nuværende transaktions id. Hvert parti tildeles et transaktions id, og transaktions id'et er et tal. Dette tal inkrementeres hver gang et nyt parti udsendes fra spouten. Hvis et parti fejler, udsendes det igen. Det genudsendte parti vil indeholde samme transaktions id som før, hvor partiet består af præcis de samme tupler. Fordi Storm har en stærk orden af partierne, vil man kunne sørge for at tupler kun behandles én gang.

Ønsket er, at man kun vil udføre tupel space operationen én gang, og derfor er den lokaliseret i forpligtelses fasen. For at tupel space operationen kun skal udføres én gang, anvendes hashmappet til at tjekke om det nuværende partis transaktions id er forskelligt fra transaktions id'et i hashmappet. Der opstår to scenarier, der skal tages hånd om:

1. Transaktions id'et i hashmappet er forskelligt fra det nuværende transaktions id: Pga. den stærke ordning af transaktioner ved vi med sikkerhed, at det nuværende parti ikke har udført tupel space operationen før. Derfor udføres tupel space operationen for det nuværende parti af tupler og derefter opdateres hashmappet med transaktions id'et.
2. Transaktions id'et i hashmappet er det samme som det nuværende transaktions id: Så ved vi, at dette parti af tupler allerede har udført tupel space operationen og derfor skal vi springe den over. Det skyldes at partiet må have fejlet efter opdatering af hashmappet, men før det har rapporteret succes til Storm. Selvom partiet af tupler springer tupel space operationen over, er det vigtigt at videresende tupel partiet til næste bolt i topologien.

Hver bolt holder styr på om et parti af tupler skal springe tupel space operationen over eller ej. Hvis halvdelen af en topologi er blevet kørt igennem, og en af tuplerne bliver fejlet, så vil tupler i det pågældende parti blive genudsendt fra spouten. De bolts, hvor partiet allerede er blevet behandlet springer tupel operationerne over. På denne måde sørger vi for, at en tupel udsendt fra spouten altid kun udfører en tupel space operation én og kun én gang.

Det er dog umuligt at garantere, at det i praksis virker 100 procent, da man for et system med 100 procent garanti bliver nødt til at tjekke, om hver udsendte tupel tidligere er blevet behandlet i en bolt. Dette vil gøre systemet meget langsommere og det er netop derfor, vi opererer på et parti af tupler. Man kunne selvfølgelig sørge for at hver tupel fik et transaktions id, men dermed ville tupel behandlingen blive udført meget langsommere, da man ikke vil udnytte Storms parallelisering. Det svarer til at man udsender partier, hvor hvert parti indeholder en tupel.

I praksis vil der godt kunne opstå en fejl i model 2. I en bolt behandles et parti af tupler ved at tupel space operationen udføres lige efter hinanden. Når alle tupler i partiet har udført tupel space operationen, så opdateres hashmappet og den pågældende bolt bliver ikke eksekveret mere for det færdigbehandlet parti. Dog kan man forestille sig et scenarie, hvor en tupel bliver fejlet lige efter et parti har udført tupel space operationen, men før den har opdateret databasen. I et sådan scenarie vil vi få eksekveret det samme parti på en bolt, mere end én gang. Det kan ikke undgås fordi der eksisterer et scenarie, hvor denne model brister. Det

minder lidt om spørgsmålet: Hvem kom først? hønen eller ægget? Vi kan ændre i koden og opdatere hashmappet, før vi udfører nogle operationer. Det vil gøre, at vi igen kan opstille en situation hvor en tupel fejles, lige efter hashmappet er blevet opdateret, men før operationerne er blevet udført. I dette tilfælde vil operationerne for den pågældende bolt aldrig nogensinde blive eksekveret. Den sidste metode garanterer ikke at data altid bliver eksekveret, så det vil heller ikke være tilfredsstillende. Man kan ikke konkludere hvilken af de to metoder der er bedst, da det afhænger af situationen. I nogle tilfælde er det bedre at en operation ikke bliver behandlet, i forhold til at operationen bliver behandlet én gang for meget. Set i et overordnet perspektiv, så er model 2 en klar forbedring i forhold til model 1. Dette kapitel rundes af med afsnit 3.8, hvor jeg har illustreret med en figur hvordan det ser ud når systemet springer en bolt over fordi den allerede er blevet behandlet der.

Model 2 bygger på Storms transaktions topologier og det opfører sig lidt anderledes ift. en normal topologi. Det er meget vigtigt at en bolt videresender samme antal tupler, som den har modtaget, fordi antallet af modtagne tupler svarer til antallet i et parti, og alle tupler i et parti skal behandles af hver bolt i topologien. Så selvom en operation springes over i en bolt fordi den er blevet behandlet før, så skal vi huske at videresende alle tupler alligevel. Hvis man ikke gjorde dette, så ville genudsendte tupler i et parti ikke blive sendt videre fra en bolt, hvor det genudsendte parti allerede har været behandlet. Det vil resultere, i at ikke alle bolts bliver besøgt af alle tupler i et parti.

3.7.2 model 2 version 2

For at gøre én gangs semantikken bedre har jeg valgt at implementere en version 2. Den første version byggede på at opdatere en database for at fortælle at et parti af tupler nu var færdig behandlet. I stedet for at opdatere databasen for hver tupel blev den kun opdateret for hvert parti. Det gør at vi får en del færre database opdateringer, men det gør også vores system mere ustabilt. I praksis er der større risiko for, at en knude går ned på et vitalt tidspunkt, og det medfører at præcis én gangs semantikken ikke bliver overholdt. Ønsket til version 2 er derfor at forbedre vores design og sørge for at fjerne dette vitale tidspunkt, så vi stadig opnår præcis én gangs semantik. I systemet haves en stærk ordning af partierne og det betyder at forpligtelses fasen for en bolt altid behandles af parti 1 før parti 2 i Storm topologien. I den første version gemtes i en database transaktions id'et for det pågældende parti, hvor nøglen var en bolts unikke hashcode. Den nye idé er at man gemmer en værdi i databasen og værdien indeholder antallet af allerede behandlede tupler sammen med transaktions id'et for det pågældende parti.

I den nye version er det muligt at en tupel bliver fejlet midt i behandlingen af et parti. Fordi databasen opdateres hver gang en tupel har udført en tupel space operation, så giver det mulighed for at forsætte en fejlet behandling, hvor den gik i stå. For at gøre det nemmere at forstå har jeg lavet et eksempel. I eksemplet udtrækkes 3 tupler og hver tupel indeholder 2 felter. Databasen hvor vi gemmer vores tilstand består af et hashmap. Til hashmappets nøgle anvendes boltens unikke id og værdien der gemmes er en værdi der indeholder partiets transaktions id og antallet af behandlede tupler. Hver gang en tupel har udført SQL-space operationen opdateres vores database. KLAIM programmet for eksemplet kan ses i listing 3.2 og det er med vilje holdt rigtig simpelt.

```
1    newloc(main:sql).
2    newloc(s1:sql).
3    in(!x, !y)@main.
4    out(x)@s1.Nil
```

Listing 3.2: KLAIM program til illustration af model 2 V2.

I figur 3.17 ses den første iteration af det oversatte KLAIM program. Efterhånden som de forskellige SQL-space operationer bliver udført opdateres databasen ligeledes. Der er lavet en farve indikation og den grønne illustrerer at alle SQL-space operationen er bekræftet og den røde betyder at der er opstået en fejl. For output bolten ses det at det for én enkel tupel er lykkes at udføre SQL-space operationen og derfor er den også tilføjet til tupel spacet s1. Ser vi på databasen ses det at der for bolt id 01182 er blevet behandlet 1 tupel før der skete en fejl i systemet. Pga. at der er fejlet en tupel bliver partiet genudsendt og vi skal forsætte med at behandle de ubehandlede tupler for den bolt, hvor der opstod en fejl.

I figur 3.18 ses anden iteration for Storm topologien, hvor parti 1 er blevet genudsendt. Vi skal ikke udføre nogle SQL-space operationer for de bolts, hvor parti 1 allerede er blevet behandlet. De bolts hvor SQL-space operationen springes over for det pågældende parti markeres ved at bolten er farvet gul. Når vi ankommer til output bolten skal vi fortsætte SQL-space operationerne for de tupler der ikke er blevet fejlet. Derfor skal vi springe den første tupel i partiet over, da den blev behandlet i sidste operation - tuplen der bliver sprunget over markeres med gult og de tupler der behandles i denne iteration markeres med grøn. Der haves ikke nogle tupler i main tupel spacet fordi de blev udtrukket af in-operationen, men for s1 tupel spacet er der tilføjet 3 tupler bestående af et enkelt felt hver. Ser vi på databasen har jeg med grønt markeret antallet af behandlede tupler, for output bolten, for at gøre læseren opmærksom på at værdien nu er opdateret fra 1 til 3. Figuren giver også et rigtig godt billede af hvordan transaktions id'et gemmes.

Dog er der et helt specielt tidspunkt hvor en tupel ikke må fejles. Hver tupel i

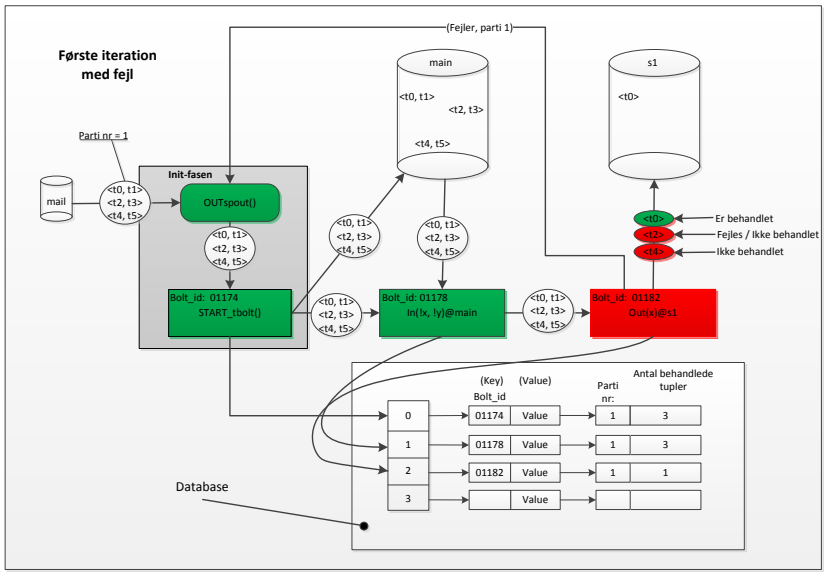

Figure 3.17: Første iteration i et eksempel på behandling af tupler for systemets model 2 version 2

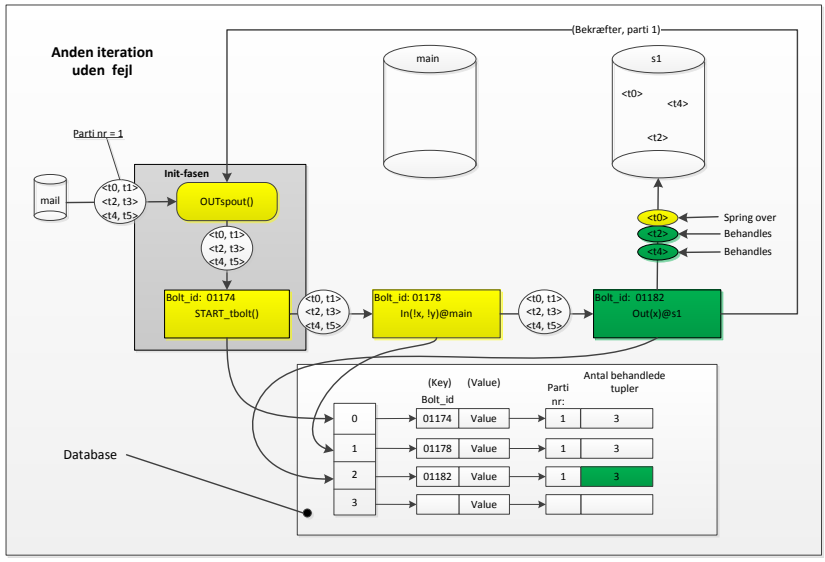

Figure 3.18: Anden iteration i et eksempel på behandling af tupler for systemets model 2 version 2

et parti skal udføre en SQL-space operation og det er netop denne operation der kun skal udføres præcis én gang fordi vi ikke ønsker nogle sideeffekter, hvor den samme tupel blive outputtet flere gange. Når SQL-space operationen er udført for den pågældende tupel, så skal en database opdateres og det er her imellem der opstår en kritisk zone. Hvis en SQL-space operation udføres og tuplen fejles før databasen er blevet opdateret, så vil systemet ikke overholde præcis én gangs semantikken. Det er dog svært at stille noget op i sådan et tilfælde fordi det er svært at bestemme hvilken operation der skal foretages først: opdatering af databasen eller SQL-space operationen. Hvis man først opdaterer databasen, så vil der være situationer, hvor SQL-space operationen aldrig bliver udført. Man kan vurdere om det er bedre at en operation aldrig udføres i forhold til at den bliver udført flere gange. Det afhænger lidt af hvad systemet skal bruges til, så det er op til brugeren hvilken metode han vil anvende. Men i forhold til den første version af model 2, så er fejl-margin faldet markant.

3.8 Afprøvning af systemet

Som det fremgår allerede, er systemet trinvist blevet implementeret. I starten brugte jeg meget tid på at gennemgå og ændre i den udleverede KLAIM implementering, for at finde ud af hvordan jeg udtrak den nødvendige information. Der er igennem hele denne del anvendt sysout for at se netop hvad og hvilke ting der skulle gemmes, så oversættelses listen kun indeholdte de nødvendige data.

I Storm er der flere forskellige måder at afprøve funktionaliteten på. Storm har en indbygget debug funktion. Med den slået til er det muligt at se hvordan tupler sendes fra den ene bolt til den anden. Det har været rigtig nyttigt under udvikling af de forskellige typer bolts. Da tupel space implementeringen er baseret på SQL-spaces, har det været muligt at følge tuplerne i de dannede lokationer i et webinterface ved at logge ind via. localhost. Det har været meget nyttigt til debugging og til at tjekke programmets funktionalitet for at se om det stemte overens med hvad man selv forventede. I kapitel 6 ses et screenshot af webinterfacet - jvf. figur 6.1, hvor der også gives et indblik i hvordan SQL-space implementeringen virker.

For at teste model 2, skal vi have mulighed for at fejle en udsendt tupel i topologien. Med Storm kan det gøres ved at kaste en FailedException(). For at gøre det mere realistisk, har jeg lavet en random funktion, hvor det er muligt at angive en procentsats for hvor ofte den skal kaste en exception. På denne måde kan man teste, om en behandling bliver gjort flere gange i Storm eller om vi springer over, hvis en operation allerede er blevet behandlet. Se figur 3.19.

Systemet er også implementeret så brugeren har mulighed for at tilføje en tupel når Storm programmet kører og denne vil blive udtrukket og udsendt af spouten under runtime. Denne funktionalitet er implementeret for at give den rigtige opførsel af vores use case, hvor man kan sætte spouten til at lytte på en ekstern database. Så snart der tilføjes noget data vil Storm spouten udtrække den.

Ser vi på figur 3.19, så har jeg illustreret hvordan fejl mekanismen er håndteret i model 2. Figuren er delt op i to iterationer, hvor vi i den første iteration fejler en tupel i den sidste operation. Når en tupel fejles i et parti skal hele partiet udsendes fra spouten igen, hvilket vi kan se at systemet gør i anden iteration. Fordi at vi allerede har udført de første 3 operationer springes selve tupel space operationerne over. Det er markeret med gult i figuren. Der er lavet tre forskellige farve indikationer i figuren:

- Grøn - tupel space operationen udføres for den pågældende bolt.
- Gul - tupel space operationen er udført for den pågældende bolt og springes derfor over.
- Rød - her fejles en tupel, og partiet genudsendes derfor fra spouten igen.

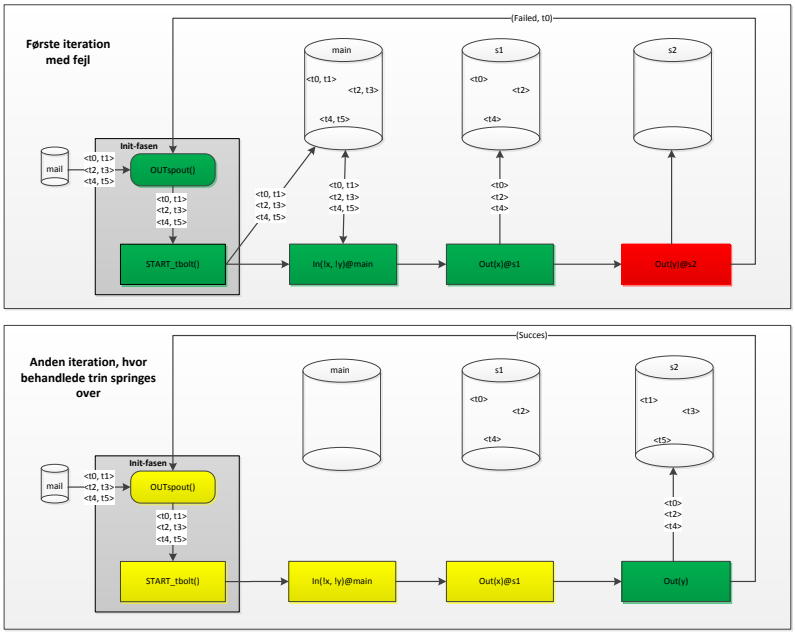

Figure 3.19: Storms fejlmekanisme for model 2

3.9 Udvidelse af det nuværende system

Flaskehalsen i model 2 og for den sags skyld også model 1 er helt klart SQL-spaces. SQL-spaces er anvendt fordi de er en del af KLAIM implementeringen og de er nemme at anvende. Det vil derfor være en fordel at fjerne tupel spaces, da indlæsning og udtrækning af tupler/data er flaskehalsen i systemet. I stedet for at anvende SQL-spaces kan man bruge Storms kommunikationsvej. Hvis vi udelukkende gør brug af Storm, så vil systemet yde bedre og derfor behandle flere tupler pr. minut. I Storm sendes data fra en komponent til en anden via forbindelserne i en topologi. I model 1 anvendte vi denne forbindelse til at sende et hashmap i mellem hver bolt i topologien. Den nye idé til udvidelsen af systemet ville være, hvor man sløjfer SQL-spaces og ikke anvender ekstern hukommelse til at lagre data/tupler. I stedet anvender man udelukkende Storms forbindelser til at sende tupler fra en bolt til en anden. I figur 3.20 ses det nuværende system og en udvidelse af det, hvor der er taget udgangspunkt i et udklip af en topologi. På venstre side ses det hvordan en tupel sendes fra en bolt til et SQL-space, og hvordan den næste bolt udtrækker den fra et SQL-space. På højre side ses en illustration af det nye system, hvor alle forbindelser til SQL-space er fjernet og i stedet anvendes Storms forbindelse mellem to bolts til at sende data. Udvidelsen af det nuværende system er ikke implementeret pga. manglende tid, men da jeg løbende har haft idéer til hvordan en udvidelse og forbedring kunne se ud har jeg valgt at fremlægge ideen.

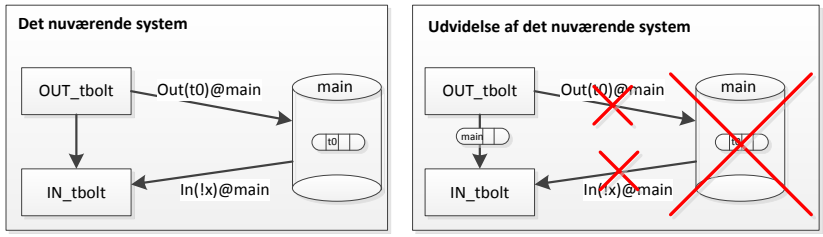

Figure 3.20: Illustration af forskellen på det nuværende system og det udvidede system, hvor der ikke anvendes SQL-spaces.

I den første model af det nye system bliver oversætningen af et KLAIM program ikke ændret, hvor vi beholder oversættelses forholdet på 1 til 1. Data skal stadig flyde rundt i systemet i form af tupler, men vi vil gerne have styr på den, på samme måde som da vi anvendte SQL-spaces. Det betyder, at det skal være muligt stadig at have en form for lager, hvor man kan gemme bestemte tupler og hvor KLAIM syntaksen holdes intakt. Grunden til at vi gerne vil have denne lagring af data, skyldes at det er rart at have styr på data. For at gør implementeringen nemmere, skal en lagring, holdes så tæt på KLAIM syntaksen som muligt. Der tænkes at man skal lagre dataen i en Arrayliste

af tupler. Og det betyder, at hvis vi outputter en tupel til @main, så er den nye idé at man danner $ArrayList < Tuples >$ og kalder den main. Denne liste indeholder alle tupler der outputtes til main. Ligeledes fjernes en tupel fra listen - på samme måde som den normalt blev fjernet/udtrukket fra et SQL-space. Pattern-matching algoritmen vil i denne løsning være noget man selv skal implementere, da vi skal have en måde at udtrække tupler på. Det er dog ikke særlig kompliceret at lave en matching algoritme, da den der anvendes for SQL-space blot udtrækker en random tupel der matcher størrelsen og søge kriterierne - se afsnit 2.1.6. Det vil dog være den mest avancerede del at implementere i denne model.

Man kan ikke have en model 1 uden også at have en model 2. Den sidste idé bygger på at man danner færre bolts, og går væk fra ideen med at oversætte i et forhold på 1 til 1. Grunden til at jeg har valgt at oversætte i et forhold fra 1 til 1, skyldes at vilkårlige KLAIM programmer kan oversættes. Brugeren der modulerer programmerne behøver ikke, at tænke på hvordan systemet laver oversættelsen, og det er i sig selv også en fordel. Dog kunne man anvende ideen med at gruppere output og input operationer, hvis de har en form for relation. Med den første model kan vi ende med at der skal sendes meget data i mellem de forskellige bolts i en topologi. Det er ikke et problem at sende meget data, men det er et problem at sende udnødvendig meget data der ikke skal bruges. I figur 3.21 ses et udsnit af en topologi, hvor den første topologi er et eksempel på at man sender udnødvendig data i mellem bolts. I topologi 1, tilføjes der en tupel til s1 ArrayListen i den første bolt, hvor efter den videresendes til næste bolt. I den sidste bolt skal vi igen bruge s1 til at udtrække en tupel. Det havde været meget smartere hvis den første og den sidste bolt var direkte forbundet efter hinanden i topologien, fordi de anvender samme lager. Da de ikke er direkte forbundet bliver ArrayListen s1 udnødvendigt videresendt til anden bolt før den ankommer til den sidste bolt. Topologi 2 i figur 3.21 viser en bedre sammensætning, hvor data bliver behandlet på samme måde, men rækkefølgen er ændret. Vi får ikke sendt, ArrayListen med data, udnødvendigt fra den ene bolt til den anden. Topologi 3 viser et eksempel, hvor vi ikke behøver at sende s1 ArrayListen mellem to bolts, fordi operationerne er grupperet og behandles i den samme bolt.

Topologi 2 fra 3.21 vil være en optimering af den første model. Den vil være lige til at implementere, da man blot skal implementere en sorterings algoritme som sorterer operationerne. Alle topologier der anvender samme tupel space forbindes lige efter hinanden. Topologi 3 vil være en ret stor udfordring, da bolt typerne skal ændres meget for at gøre det muligt at udtrække og inputte tupler i samme bolt. Det vil klart være en fordel at vi undgår at flytte data og minimerer antallet af bolts i en topologi.

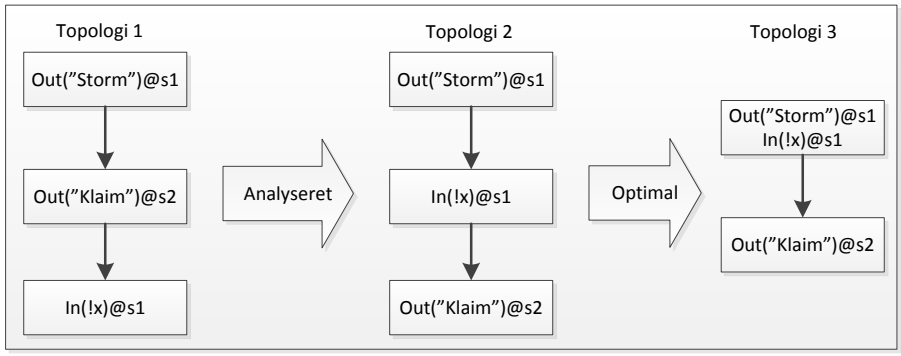

Figure 3.21: Tre forskellige topologier, men af den samme KLAIM beskrivelse. Hvor topologierne bliver bedre og bedre fordi der flyttes mindre og mindre data.

Chapter 4

Gennemgang af en use case

I dette kapitel gennemgås hele det udviklede system, så det giver indblik i hvordan systemet oversætter et KLAIM program for til sidst at ende med at have en Storm topologi til behandling af data. Først forklares en use case der giver et godt billede af, hvad man vil bruge systemet til. Systemet opdeles i 3 dele:

1. KLAIM programmet
2. Oversættelsen
3. Storm topologien

Det skal lige siges, at der tages udgangspunkt i model 2 for dannelse af Storm topologien.

4.1 Use casen

Use casen der vil blive brugt i denne afhandling, tager udgangspunkt i jobansøgninger fremsendt til et firma via email. En ansøger fremsender sin jobansøgning via email til adressen: jobansøgning@firma.dk. Email adressen bruges

udelukkende til modtagelse af jobansøgninger. Job opslaget findes på firmaets hjemmeside, hvor ansøgeren bliver gjort opmærksom på, hvordan ansøgning skal fremsendes. Hvert jobopslag har et unikt nummer der anvendes til at referere til et bestemt jobopslag. En ansøger skal i email feltet angive 'job' og nummeret på det job, der søges. Yderligere er det vigtigt at der vedhæftes tre forskellige filer:

- Ansøgning - ansøgerens personlige ansøgning.
- CV - en levnedsskildring der oplyser om uddannelses- og erhvervsmæssige kvalifikationer.
- Formular - en formular firmaet har opstillet.

I jobopslaget findes en formular der skal udfyldes, for at søge jobbet. I formularen skal ansøgeren blandt andet angive hvad id'et er til hans Facebook og Linkedin profiler. Firmaet er meget interesseret i at kunne bruge Facebook og Linkedin til at søge flere informationer om ansøgerne. Systemet der ønskes udviklet skal anvende en ansøgers Facebook id for at undersøge, hvor mange venner og billeder han har på sin profil. På samme måde anvendes også linkedin id'et til at se hvor mange forbindelser ansøgeren har. De nye informationer fra Facebook og Linkedin bliver samlet i to nye pdf-filer med navnene hhv. Facebook og Linkedin. De fem pdf filer arkiveres til sidst i en database sammen med ansøgerens navn og nummeret på jobopslaget. Dette system er rimelig simpelt, men det tænkes at man kan udvide disse services til mere avancerede teknikker. F.eks. kan man undersøge, om ansøgeren er tilhænger af nogle bestemte grupper på Facebook, som firmaet ikke kan acceptere.

Systemet udvikles i Storm, hvor der tænkes at man har en spout, der er koblet op til at lytte på en mail box. Hvis der indkommer en email bliver den udtaget af spouten, og de vigtige informationer videresendes i form af en tupel med felterne <navn, job-nummer, ansøgning, formular, CV>. En topologi bestående af seks forskellige bolts dannes, hvor der haves to service bolts. Den først service bolt anvender facebook id'et til at finde ansøgerens facebook profil for at se hvor mange venner og billeder ansøgeren har - i praksis vil et screenshot være tilstrækkeligt at gemme. De fundne oplysninger fra facebook anvendes til at berige tuplen med informationer om ansøgeren. Den anden service bolt anvender linkedin id'et til at finde linkedin profilen, hvor der undersøges hvor mange forbindelser ansøgeren har. Den resulterende tupel kan nu dannes og den har følgende felter: <navn, job-nummer, ansøgning, formular, CV, facebook, linkedin>. Den videresendes til den sidste bolt, som gemmer den i en database/tupelspace.

4.2 KLAIM programmet

Transaktionsbegreberne for use casen moduleres i KLAIM . Der haves to forskellige services, og begge services modtager en tupel med et enkelt felt og danner i servicen et ekstra felt: linkedin.pdf eller facebook.pdf. Derfor haves nu i tupel spacet tupler med to felter. Hele KLAIM programmet kan ses i listing 4.1.

```
1  newloc(face:facebook).
2  newloc(linked:linkedin).
3  newloc(database:sql).
4      in(!navn, !jobopslag, !ansogning, !formular, !CV)@main.
5      out(navn)@face.
6      in(!navn, !facebook)@face.
7      out(navn)@linkedin.
8      in(!navn, !linkedin)@linked.
9      out(navn, jobopslag, ansogning, formular, CV, facebook,
           linkedin)@database.nil
```

Listing 4.1: KLAIM program for use casen

Spouten er sat op til at lytte på et map og dette map svarer til aflytningen af en mail boks. Det betyder, hvis der indkommer en ny "mail" i vores map, så vil spouten udtrække oplysningerne fra mailen og tilføje dem til tupel spacet main. Dette sker i Storm regi, da det giver en god opførsel i forhold til ønsket fra Kapow Software. Det er nu muligt at udtrække de indkomne jobansøgninger fra main tupel spacet og derved udføre de transaktions, som vi ønsker ud fra vores use case.

Først udtrækkes alle oplysningerne, der er blev tilsendt mail boksen. Så anvendes navnet på jobansøgeren til at tjekke hans facebook profil og der dannes et nyt felt i vores tupel med de fundne informationer. Derfor outputtes en tupel med to felter til face tupel spacet. Præcis det samme sker for linkedin servicen. Til sidst skal alle informationerne gemmes i en database. Da et tupel space også kan bruges, som en database, er det nærliggende at anvende det til at gemme de berigede jobansøgninger. Det er også rigtig nemt at udtrække en jobansøgning, hvis man ved at "Henrik" har lavet en ansøgning på jobbet med ansøgnings nummer "1337", kan man blot gøre følgende i KLAIM :

```
1  newloc(database:sql).
2      in("Henrik", "1337", !ansQgning, !formular, !CV, !facebook,
           !linkedin)@database.nil
```

Listing 4.2: KLAIM program for use casen

I listing 4.2 ses et eksempel på hvordan man nemt udtrækker en jobansøgning med bestemte søgekriterier.

4.3 Oversættelsen af KLAIM

Selve parsningen af KLAIM koden er allerede håndteret for mig i den udleverede del af KLAIM implementeringen. Der haves forskellige operationer i KLAIM og når en af operationerne matcher en regel i grammatikken, dannes en ny klasse hvor de vigtige elementer fra operationen gemmes. Det betyder at vi internt får repræsenteret elementerne så som tupel space, og hvilken tupel den pågældende operation indeholder.

Hver operation har sin egen evalueringsfunktion og på denne måde ved vi præcis, hvilken operation der nu evalueres. Vi danner oversættelses mappet og det indeholder alle de vigtige oplysninger fra KLAIM koden, der bruges for at danne en Storm topologi. Mappet er bindeleddet mellem de to forskellige implementeringer. I figur 4.1 ses hvordan hver operation oversættes til Storm. En dybere gennemgang af oversættelsen og opbygningen af en Storm topologi kan ses i bilag A. Der er taget udgangspunkt i hvordan en input operation oversættes fra den mødes i KLAIM, til den bliver behandlet i Storm, hvor den udfører en tupel space operation.

I figur 4.1 ses hvordan KLAIM operationerne oversættes og gemmes i oversættelses mappet - dog er det ikke med de rigtige argumenter, der er vist i tabellen. Det ville ikke give lige så god mening, da der f.eks. for tupel space variablen, blot ville stå node.getName(), i stedet for selve navnet på tupel spacet. Da det blot er en illustration, er det nemmere at se, hvad variablerne i mappet er sat til. Når mappet er opbygget, kan vi skifte til Storm domænet, og udrulle oversættelses mappet, hvor vi opbygger vores topologi ved at oversætte operationerne til bolts og forbinde dem i den rækkefølge de mødes. Det kan også ses, at Storm topologien har en initialiseringsfase og det er netop denne der implementerer udtræknings systemet. Spouten udtrækker automatisk indkomne mails og sender dem ud i topologien. Mailen er blot et map, hvor det er muligt at genudsende partier af tupler, hvilket er en meget vigtig faktor for, en transaktions topologi overholder én gangs semantikken.

Dette system implementeret således, at man vælger om man vil anvende model 1 eller model 2. Det betyder at man enten danner en topologi bestående af batch bolts eller regulære bolts. Det er ikke muligt at sammensætte dem. Ser vi bort fra de to modeller og analyserer systemet mht. hvilke typer af bolts der skal anvendes i Storm topologien ses, at de to service bolts er af samme type. Operationerne de to bolts foretager er ikke afhængige af at måtte blive udført én og netop én gang. Det skyldes, at de ikke har nogen sideeffekter for deres handlinger. At der haves en process, der flere gange [1] har mulighed for at

[1] Det kræver at en tupel i topologien bliver fejlet, for så skal den udsendes fra spouten igen

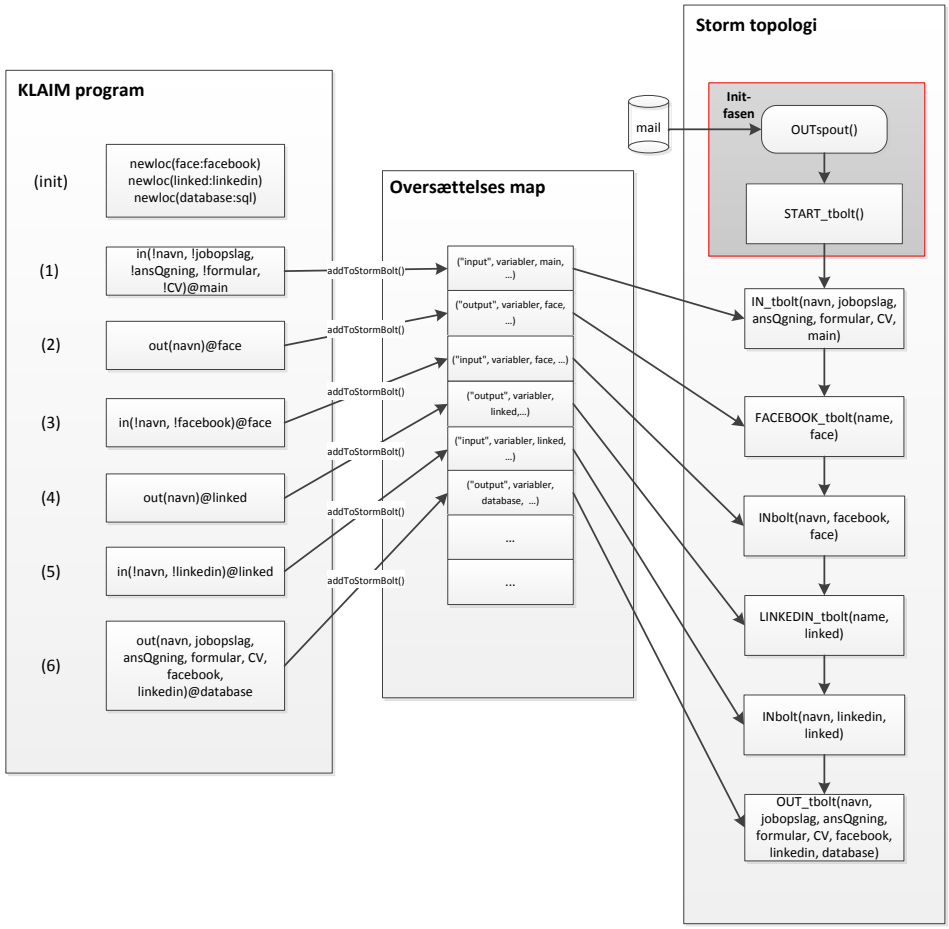

Figure 4.1: Use case, hvor man kan se oversættelsen fra KLAIM til Storm

undersøge brugerens facebook/linkedin profil giver ikke katastrofale følger. Men det er ikke optimalt at der er mulighed for at udtrække samme data 50 gange, men ønsket for denne bolt er, at den uanset hvad, henter nødvendige data.

Det er derimod vigtigt at den sidste bolt kun gemmer informationerne netop én gang. Operationen, hvor den resulterende tupel gemmes i databasen, må derfor kun udføres en gang, da man ikke ønsker duplikeret data. Der skal derfor anvendes en transaktions bolt for at opfylde dette ønske. At have duplikeret tilfælde af samme ansøger er ikke særlig katastrofalt, men hvis det kan undgås er det optimalt, og stadig ret vigtigt, for hvis man skalerer og har mange brugere, så optager det enorme mængder af plads med duplikeret data. Man kan tænke sig eksempler hvor sideeffekter vil have mere katastrofale følger. F.eks. en service der har til formål at gå ind på en hjemmeside og bestille en jumbojet, hvor der i dette tilfælde vil være stor forskel på om operationen kun behandles én eller flere gange. Det vil klart være at foretrække, hvis jumbojetten kun bestilles det antal gange der ønskes. I et system med betydelige sideeffekter, er det derfor klart at foretrække at anvende transaktions bolts.

Ideen med at analysere typen af bolts skyldes, at man med regulære bolts får en større parallelisering i Storm, da arbejdet nemmere kan fordeles fordi Storm ikke skal tage højde for, at tupler skal behandles i en bestemt rækkefølge i forpligtelses fasen.

Chapter 5

Resultater

I dette kapitel skal Storm testes for at se hvordan det yder i forskellige scenarier. Testene har ikke noget at gøre med KLAIM koden eller måden det oversættes på. Dog dannes topologierne stadig ved, at der laves en række KLAIM programmer, og derefter anvendes de til, at teste systemets ydeevne. Topologierne der testet består af forskellige antalt bolts, hvor der testes ved, at udsende partier af forskellig størrelse. Robustheden af systemet vil også blive gennemgået. Da implementeringen af robustheden også har været afgørende for ydeevnen af systemet.

5.1 Robusthed

Robusthed er også en af de ting der skal testes, men det er en del sværere, da jeg ikke har mulighed for at opsætte en klynge af computere. Dog kan man med VMware eller lignende godt opsætte en klynge, men tiden var ikke til det og alt er kørt lokalt i Storm. Ved at have en klynge af computere, vil det være muligt at teste hvordan Storm opfører sig hvis en af computerne går ned. Skal der alligevel sige et par ord om robusthed for det udviklede system, så er det at Storm giver en del optimeringer i forhold til den oprindelige KLAIM implementering. Vi kan love at data altid bliver behandlet. Hvis en knude får uddelegeret noget arbejde og den går ned før den overhovedet har modtaget og

behandlet nogle tupler. Så vil Zoopkeeperen i Storm prøve at se om knuden er fuldstændig død, og hvis det er tilfældet vil den rykke det uddelte arbejde over på en anden knude. I selve topologien vil den i første omgang prøve at sende tupler til den døde knude, men da den er gået ned og aldrig reporterer noget tilbage til Storm, så vil Storm genudsende det fejlede parti. Da arbejdet nu er rykket over på en anden fungerende knude, så vil de genudsendte tupler blive behandlet.

Ser vi på model 2's robusthed, så bliver alle tupler i en topologi kun behandlet én gang. Det betyder at en udsendte tupel der fejles og bliver genudsendt vil springe de operationer over, hvor den allerede er blevet behandlet. Selv hvis en knude går ned midt i en behandling af et parti. Hvis et parti afbrydes midt i behandlingen, så vil det afbrudte parti blive genudsendt. Når partiet ankommer til den knude hvor behandlingen blev afbrudt, så vil behandlingen fortsætte med de ubehandlede tupler. Dog er der et helt specielt tidspunkt hvor knuden ikke må gå ned. Hver tupel behandles ved at der for den pågældende tupel udføres en tupel space operation og derefter gemmes det i databasen at den pågældende tupel er behandlet. Hvis tupel space operationen bliver udført, og knuden går ned før databasen er blevet opdateret, så vil systemet ikke overholde præcis én gangs semantikken. Det er dog svært at gøre noget ved denne problemstilling, da vi gerne vil have et hurtigt system. Scenariet kan sammenlignes med, hvad kom først? hønen eller ægget. Det er muligt at gøre systemet mere sikkert ved at man først opdaterer databasen og derefter udfører operationen. Det vil gøre at der kan opstå tilfælde hvor en tupel ikke bliver behandlet. Dog er model 2 også blevet forbedret.

Til at starte med var model 2 udviklet, så databasen blev opdateret når et parti var færdigbehandlet og det gav én database behandling pr. parti. Det er blevet ændret for at give systemet en bedre robusthed og sikre at hver tupel kun behandles én gang. I den første version af model 2 var der en zone, hvor systemet ikke måtte fejle. En bolt er delt op i to faser, og i forpligtelses fasen haves en behandlings zone. Behandlings zonen indebærer SQLspace operationerne for hver tupel i det pågældende parti, hvis knuden går ned her brister systemet. De tupler i partiet der allerede er blevet behandlet, når knuden går ned, vil blive behandlet flere gange - da databasen ikke er blevet opdateret. Storm detekterer at knuden er gået ned og genudsender det fejlede parti. Alle bolts der allerede har behandlet partiet springes over indtil partiet ankommer til den bolt hvor behandlingen fejlede, og hele partiet behandles. Det betyder at nogle tupler bliver behandlet mere end én gang. De to versioner af model 2 har både fordele og ulemper. Fokus for denne afhandling har mest været på robusthed, hvorfor det er prioriteret at tupler kun behandles én gang. Derfor vælges den lidt langsommere version, hvor der foretages et database kald pr. tupel. Hastigheden for systemet forvrænges, men det gør systemet mere robust. Med den første version af model 2 ville der for store partier være færre database kald, men

fejl-marginen ville ligeledes også være større.

5.2 Systemets ydeevne og skalerbarhed

Nu skal systemets ydevne testes for at se hvordan Storm håndterer forskellige størrelser af topologier og se hvordan det reagerer på forskellige størrelser af partier. Der anvendes udelukkende input og output operationer i undersøgelsen for at holde det simpelt. KLAIM programmet der anvendes til at teste systemet kan ses i lising 5.1. KLAIM programmet oversættes til en topologi bestående af 5 bolts. Ligeledes laves der tests hvor en topologi består af 10, 20 og 40 bolts.

```
1 "newloc(s1:sql)."+
2 "newloc(s2:sql)."+
3 "newloc(s3:sql)."+
4 "in(!name, !alder)@main."+
5 "out(name, alder)@s1."+
6 "in(!x, !y)@s1."+
7 "out(x, x)@s2."+
8 "out(y, y)@s3.nil";
```

Listing 5.1: KLAIM program til at teste systemets ydeevne.

Ydeevnen er lidt svær at måle direkte, for Storm danner forskellige tråde og det er lidt ude af mine hænder, hvordan de håndteres. Når topologien har behandlet alle tupler, så kører topologien stadig og man kan ikke anvende en timer, da det er svært at bestemme hvornår topologien er færdig. Dog kan man anvende Storms debug funtkion og slå den til. Mens Storm topologien kører og debug funktionen er slået til, så skrives der til konsollen, hvad de forskellige tråde gør og de er stemplet med den tid der er gået siden topologien startede. Manuelt er det muligt at se i konsollen når alle tupler er færdigbehandlet. Alle tests bliver udført med samme indstillinger, hvor der skrues på nogle parametre. Parameterne der ændres er:

- Parti-størrelsen - hvor de forskellige størrelser er: 1, 2, 3, 5, 10, 20, 30 og 60.
- Størrelsen af topologien - hvor antallet af bolts er: 5, 10, 20 og 40.

Alle forsøgene er udført, hvor spouten udsender 60 tupler. Beskeden der aflæses i konsollen for at se når en topologi er færdig kan ses i figur 5.1. Hvis man ser på figuren, så er der indtegnet en pil på den kommando der markerer at topologien er færdigbehandlet. Dernæst findes findes tiden for den tråd der outputter til

konsollen, lige efter den sidste output operation har opdateret databasen. Tiden topologien har brugt på at behandle alle tupler er markeret med rød og i dette tilfælde er det 120940 ms.

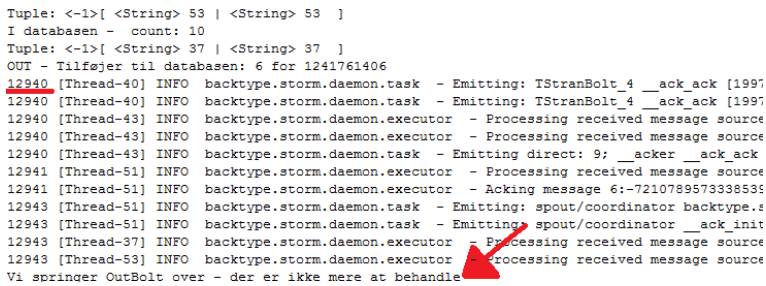

Figure 5.1: Konsol udskrift for Storm der viser hvornår en topologi har færdigbehandlet alle tupler

Der er lavet en tabel, som viser alle målingerne af systemet i de forskellige tilfælde. Hver behandling er lavet tre gange og tabellen viser derfor gennemsnits kørslen for de tre målinger. Resultaterne kan ses i tabel 5.1. Alle tests er lavet ved at SQL-space serveren genstartes og derfor er alle tupel spaces tomme fra start. Det skyldes at jeg under testene fandt ud af at der var stor forskel på om man kørte en test, hvor SQL-space serveren var genstartet eller man blot fortsatte med at bruge den startede server. Jo flere tupler der var i tupel spacene jo længere tid tog det før topologien var færdigbehandlet. Derfor startes en ny server for hver test af et parti.

Bolts:	Parti= 1	Parti=2	Parti=3	Parti=5	Parti=10	Parti=30	Parti=60
5	12237ms	7859ms	6413ms	5752ms	5443ms	5620ms	5483ms
10	13251ms	10396ms	9494ms	8836ms	8374ms	8209ms	8257ms
20	18802ms	15596ms	14676ms	13704ms	12687ms	12656ms	12540ms
40	28527ms	23859ms	22664ms	21455ms	20438ms	20614ms	20734ms

Table 5.1: Tabellen viser hvor lang tid Storm er om at behandle 60 tupler. Hvor der ændres på antallet af tupler i et parti, samt antallet af bolts i topologien.

Skal vi knytte et par ord om måleresultaterne, så ses det tydeligt at vi uanset topologiens størrelse rammer en optimal behandlingstid. Den optimale behandlingstid forekommer for en parti størrelse på 10-60 tupler. Her kan Storm ikke parallelisere operationerne på en mere optimal måde, fordi det er optimalt allerede ved cirka 10 tupler pr. parti. Det kan skyldes at flaskehalsen ligger et

andet sted. Det ses at uanset topologi størrelse, så vil en parti størrelse på 1 ikke yde særlig godt. Fordi tupler er stærkt ordnet, så er der ingen mulighed for at have særlig meget parallelisering og derfor kan vi også se at allerede ved en parti størrelse på 2 falder den samlede behandlingstid en del. Tiden forbedres for hver gang vi øger parti størrelsen indtil man rammer en øvre grænse på 10 tupler pr. parti. Under testene har jeg holdt øje med CPU-forbruget, hvor det kunne ses at CPU'en ydede mere og mere, i form af parti størrelsen steg. Det underbygger igen påstanden om at operationerne her parallelliseres mere end ved lavere parti størrelser. CPU'erne ydede dog maksimalt 70 procent og var derfor aldrig fuldt overbelastet. I figur 5.2 ses en graf der viser hvordan behandlingstiden varierer for forskellige topologi størrelser og variation af antallet af tupler pr. parti. Jeg har valgt ikke at medtage parti behandlingstiderne for parti størrelser på 30 og 60 tupler, da grafen blot ville blive meget flad og behandlingstiden ikke ændre sig synderligt, hvis man har en parti størrelse på mere end 10. Det ses på grafen at uafhængigt at topologi størrelsen falder behandlingstiden kraftigt pga. forøgelsen af antallet af tupler pr. parti. De fire forskellige topologi størrelser giver anledning til den samme kurve for behandlingstiden, hvor alle ender med en optimal behandlingstid når antallet af tupler pr. parti rammer 10.

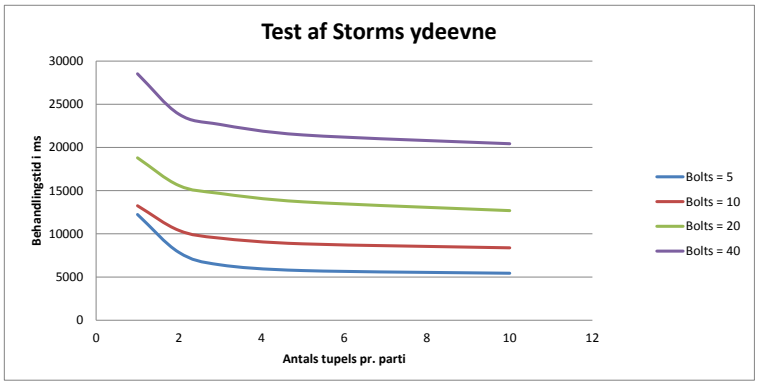

Figure 5.2: Grafen er lavet udfra resultaterne i figur 5.1 - dog er parti størrelsen på 30 og 60 ikke medtaget.

Topologi størrelsen er den sidste faktor der er blevet skruet på. Som forventet tager det selvfølgelig længere tid at behandle en større topologi fordi der skal udføres flere operationer. Antallet af bolts fordobles, men det gør behandlingstiden dog ikke, hvilket er godt at tage med. Det er logisk at behandlingstiden bliver større fordi der skal udføres flere operationer, men dog fordobles den ikke. Det skyldes at Storm er godt til at skalere. I takt med at antallet af bolts forøges bruges der ligeledes også mere hukommelse og det er som forventet fordi vores program optager mere plads.

Overordnet set er det nogle fine resultater Storm giver os. Jeg tror at tiderne ville være endnu bedre hvis vi ikke skulle foretage sql-operationerne. Flaskehalsen i systemet er helt klart SQL-spacene, hvis man havde et system uden SQL-spacene ville det behandle meget hurtigere. Udtrækning og indsættelse af tupler i SQL-spacene er det der kræver tid. Der er flere ting der peget på SQL-space som flaskehalsen. Blandt andet er det tydeligt at se at tupler sendes meget hurtigere mellem boltsne i situationer hvor SQL space operationen springes over. Behandlingstiden forlænges også hvis vi ikke genstarter SQL-space serveren i mellem hver test. Det betyder at præcis den samme test bliver langsommere og langsommere jo flere gange vi kører den uden at genstarte SQL-space serveren. Databasen der bruges til at tjekke om en tupel er behandlet før er blot et map og den er tom hver gang programmet startes og der er ingen tegn på at den har nogen indvirkning på ydeevnen, så længe vi anvender SQL-space implementationen. Storm laver en helt masse forbindelser til SQL-space serveren og jo større topologien er jo flere forbindelser dannes der. Når topologien er færdig med at behandle alle tupler lukkes den ned og CPU-forbruget stiger til 60-70 procent. Fordi SQL-space serveren står og arbejder indtil alle dannede forbindelser er lukket eller til man manuelt lukker SQL-space serveren ned. Igen et godt tegn på at SQL-space nok ikke er den bedste implementering sammen med Storm. CPU-forbrugets stigning til 60-70 procent sker ved topologier med 20 bolts og derover. Jeg var tidligt klar over at flaskehalsen i systemet nok ville være SQL-spacene og det viser resultaterne også fint.

Chapter 6

Relateret arbejde

6.1 tupel space

Et tupel space er en form for distribueret hukommelse der anvendes til parallel eller distribueret beregninger. Den er baseret på en associativ-memory data struktur. Det betyder at tupler kan tilgås ud fra deres type og indhold - og ikke ved deres adresse. Tupel spacet indeholder en samling af tupler, hvor det er muligt at tilgå tupler samtidigt. Som en illustration kan man tænke på at der er en gruppe af processer, som fremstiller forskellige styk data og en anden gruppe af processer anvender disse data. Fremstiller-processerne tilføjer data som tupler til et tupel space, hvor anvender-processerne udtrækker data som matcher et givent mønster. En tupel er anonymt og består af en vilkårlig sekvens af informationselementer. Pattern-mathcing anvendes til at udtrække tuples og i afsnit 2.1.6 kan se en uddybelse af denne mekanisme. Populariteten af tupel space paradigmet har medført en række forskellige run-time systmer. For industrien kan der nævnes:

- SUN JavaSpaces
- IBM T Spaces

og akademisk anvendes følgende:

- PageSpace
- WCL
- Lime
- TuCSon

Der findes et hav af tupel space implementeringer til mange forskellige programmerings sprog. I denne afhandling vil der dog blive anvendt to andre typer tuplespaces end de nævnte, nemlig LighTS og SQLspaces. Da SQLspaces bliver brugt i STORM-delen i denne afhandling, så vil det blive introduceret i afsnit 6.1.1. LighTS følger med i den udleverede kode fra Kapow Software og den bliver ikke anvendt særlig meget af mig.

6.1.1 SQL-spaces

Da der i denne afhandling anvendes tupel spaces til at holde de tupler der opereres på, vil vi kort gennemgå den anvendte implementation, SQL-spaces. Den er baseret på relationel databaser, hvor der her kan nævnes MySQL. API'et er meget simpelt, let at forstå og anvender tuplespace koncepterne. Ideen oomkring tuplespaces startede i midten af 80'erne, hvor det blev introduceret af Gelertner og Carriero fra Yale universitetet. Det blev introduceret sammen med Linda koordineringssproget. Arkitekturen bygger på den velkendte client/server model, men kommunikationen foregår udelukkende via tuples. Brugeren har mulighed for at indlæse og udtrække tuples fra en server uden at have nogen information om de andre brugerere. Mekanismen hvor man laver en forespørgelse i et tuplespace system er hovedsageligt associativ, dvs. en forespørgsel bliver lavet ved at anvende template tuples med et såkaldt formelt felt, som forstås som et wildcard. Tuplespaces er en nem og elegant måde at lave distribuerede software systemer på, som er robuste og modulære fordi de er koblet løst sammen [AG13]. SQLspaces udstiller alle de operationer som anvendes i denne afhandling(out, in, rd) og mange flere, her kan blandt andet nævnes multi-sprog support, versionering og support for Android. Vi benytter os blot af de simple funktioner fra sqlspaces. Til sidst skal der nævnes det smarte webinterface, som kan anvendes til at debugge run-time, da man via webinterfacet har mulighed for at se hvilke tupler de oprettede tuplespaces indeholder.

6.1.2 Anvendelse af SQL-spaces

For at demonstrere hvordan man anvender sqlspaces, så tager jeg fat i en lille stump java-kode, som forklarer de operationer jeg anvender, jvf. listing 6.1.

```
1    TupleSpace ts = new TupleSpace();
2    tupel t1 = new tupel("foo", 1);
3    ts.write(t1);
4
5    tupel tempTuple = new tupel(String.class, Integer.class);
6    tupel retTuple = ts.take(tempTuple);
```

Listing 6.1: Eksempel på anvendelse af et tuplespace

I den første linje danner vi et nyt tuplespace, som vi kalder ts. Dernæst laver vi en ny tupel t1, og denne tupel bliver dannet med to formelle felter "'foo"' og tallet 1. Nu har vi så mulighed for at indlæse tupel t1 i ts ved hjælp af take-metoden, som det gøres i linje 3. Nu har vi tilføjet et tupel til vores tuplespace. For at udtrække dette tupel skal vi lave en template, med to felter af samme type som det tupel vi gerne vil matche i tuplespacet. Da tuplen i tuplespacet består af en String og en Integer, så skal vi oprette en ny template tupel, som det ses i linje 5. Nu skal tuplen så udtrækkes fra tuplespacet og dette gøres med take-metoden hvor argumentet er template tuplen. Via pattern-matching bliver en tupel tilfældig matchet - en tupel bliver blot udtrukket hvis den matches. I figur 6.1 ses et screendump af webinterfacet for et sqlspace, hvor vi har oprettet et tuplespace med navnet s1 og tilføjet et par forskellige tuples som indeholder integeren 13.

6.1.3 Varianter af KLAIM

6.1.4 OpenKlaim

OpenKLAIM er den første udvidelse af KLAIM og den er første gang blevet præsenteret i [BBN+03]. Konkret sagt så er OpenKLAIM designet for at gøre det muligt for brugeren at give mere realistiske repræsentation af åbne systemer. Åbne systemer er dynamiske udviklende strukturer som betyder at nye knuder kan blive forbundet eller eksisterende knuder kan frakobles. Et åbent system er derfor mere ustabilt fordi tilslutninger og frakoblinger i netværket kan ske uventet og være midlertidige og man kan ikke antage at det underliggende kommunikations netværk altid er til rådighed. De forskellige ruter i netværket kan være påvirket af restriktioner - her tænkes på midlertidige fejl eller

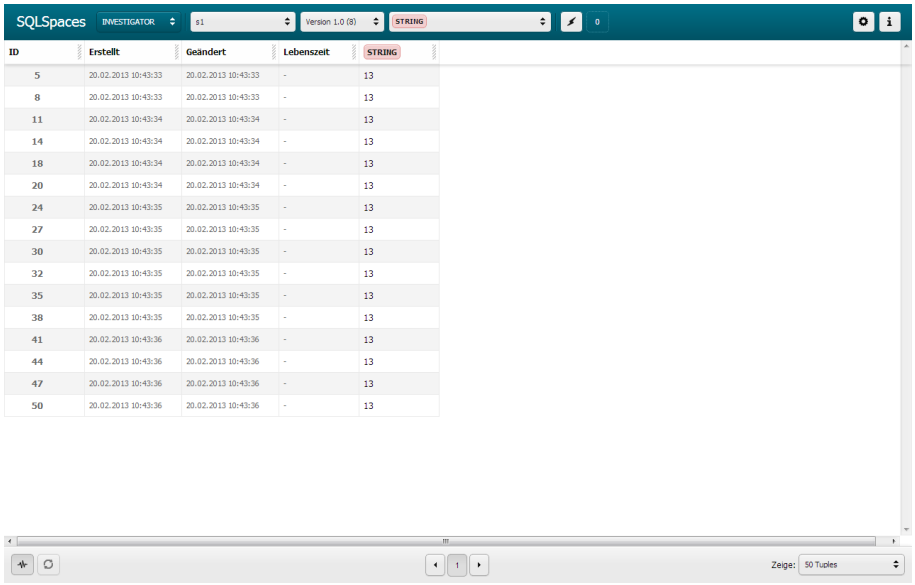

Figure 6.1: Screendump af sqlspace webinterface

firewall regler. Det medfører at navngivning af knuderne ikke er tilstrækkeligt, til at tilslutte til en anden knude eller udføre fjernoperationer. For at få KLAIM til at håndtere åbne systemer, skal sproget udvides til eksplicit at modellere tilslutningsmuligheder mellem netværks knuder og håndtere ændringer i netværks topologien.

OpenKLAIM opnås ved at man udvider KLAIM mekanismer der dynamisk opdaterer allokeringen af omgivelserne og håndterer knude tilslutsningsmuligheder. Ydermere skal der tilføjes en ny kategori af processer, nemlig knude koordinatoren, som udover normale KLAIM operationer kan eksekvere privilegerede operationer som tillader etablering af nye forbindelser, tillader tilslutnings anmodninger og fjerner tilslutninger. Derfor er der bland andet nogle nye operationer i OpenKLAIM, her af kan der nævnes: **login**(ℓ), **logout**(ℓ) og **accept**(u) - som man typisk kender det fra hjemmesider eller applikationer hvor man vil opnå adgang med flere rettigheder. Forskellige typer af brugere har forskellige rettigheder.

6.1.5 HotKLAIM

HotKLAIM også kendt som *High-Order Typed* KLAIM hænger sammen med en udvidelse af *system F* med primitiverne fra KLAIM. System F er en typed lambda calculus, som gør sig anderledes fra simpel typed lambda calculus ved at introducere en mekanisme af universel kvantificering af typer. Da system F ikke er noget der vil blive berørt i dette projekt vil det ikke blive yderligere uddybet. HotKLAIM er også en udvidelse af KLAIM, som skal forbedre KLAIM med generelle funktioner. Det gøres ved at tilføje den kraftfulde abstraktions mekanisme og typerne fra system F, som er ortogonal med netværks bevidst programmering. Disse funktioner gør det muligt at arbejde med stærke parameteriserede mobile komponenter og muliggør dynamisk håndhævelse af værtens sikkerhedspolitik. Sikkerhedspolitikken siger at typerne er metadata udtrukket under run-time og anvendt til at udtrykke garantien af pålidelighed.

6.1.6 O'KLAIM

O'KLAIM også kendt som den objekt orienterede version af KLAIM, som er en sproglig integration med objekt orienterede funktioner. Koordineringsdelen og den objekt orienterede del er ortogonale, hvilket muliggør, i hvert fald i princippet, at en integration vil virke for enhver type af KLAIM - dog nævnes det at cKLAIM er den tidligste version som kan anvendes. Ligeledes vil det også virke for andre typer calculi med mobilitet og distribuering. Man anvender O'KLAIM som koordinerings sproget for udveksling af mobil objekt-orienteret kode blandt processorne i et netværk.

6.1.7 X-KLAIM

Nu introduceres den sidste type af KLAIM nemlig X-KLAIM også kendt som eXtended-KLAIM. Det er et eksperimentielt programmerings sprog som udvider KLAIM med en high-level syntaks for processor. X-KLAIM indeholder: variabel erklæringer, forbedrede operationer, betingelser, sekventiel og iterativ process sammensætninger. Implementeringen af X-KLAIM er baseret på det tidligere nævnte KLAVA. X-KLAIM anvendes til at beskrive på et højere niveau af de distribuerede applikationer, mens KLAVA bruges som både middleware for X-KLAIM programmer og som et Java framework mht. programmeringen i forhold til KLAIM-paradigmet. Med denne anvendelse af KLAVA er det muligt for programmøren at udveksle et hvert Java objekt igennem tuples og derfor implementere en mere finkornet og stærkere mobilitet.

Chapter 7

De sidste overvejelser

I samarbejde med Kapow Software var målet med denne afhandling, at orkestrere distribuerede systemer over store datamængder. Det er blevet undersøgt, om man kan udvikle et system der sørger for, at en sekvens af behandlinger bliver udført én og netop kun én gang - og hvor ingen transaktioner går tabt. Til at beskrive processerne anvendes en variant af KLAIM, som det fortolkende domænespecifike sprog. Interaktionen mellem de forskellige komponenter sker igennem multiple distribuerede tupel spaces. En tupel er anonym og udtrækkes fra et tupel space vha. pattern-matching. Transaktionsbegreberne er implementeret, så de kører på ryggen af Storm, men det kunne godt have været et hvilket som helst andet system. Eftersom Storm har opfyldt de ønsker og krav, der er blevet fremstillet, har det været tilfredsstillende kun at fortsætte arbejdet med det, og ikke stifte bekendtskab med andre implementeringer.

Først skulle der findes en variant af KLAIM, som var så simpel som muligt, men stadig skulle det kunne anvendes sammen med tuple spaces. Først anvendtes en meget simpel variant for derefter at tilføje flere og flere operationer og til sidst ende med at have sKLAIM varianten. Varianten blev valgt som et kompromis mellem funktionalitet og enkelthed.

Da jeg havde opnået en masse erfaring og forståelse af KLAIM begyndte jeg at arbejde med Storm. Her opbyggede jeg en række topologier for at se hvordan Storm behandler tupler. Både KLAIM og Storm behandler lineært data via tu-

pler. Efter at have arbejdet meget med begge implementeringer, og forstået hvordan tupler blev behandlet kom jeg på forskellige ideer til at oversætte KLAIM. Efter at have analyseret forskellige løsningsmetoder fandt jeg det bedst at lave en oversættelse på 1 til 1. Det betyder at en KLAIM operation oversættes til en komponent i Storm.

Når KLAIM varianten var på plads var det muligt at analysere, hvad og hvordan den skal oversættes. Det næste skridt var derfor at finde ud af hvordan KLAIM programmet skulle gemmes. Ideen var at opbygge en oversættelses liste, efterhånden som vi mødte de forskellige operationer, når et KLAIM program blev parset. Det resulterer i at oversættelses listen ville indeholde en lineær repræsentation af KLAIM programmet. For hver operation i KLAIM gemmes i oversættelses listen: hvilken KLAIM operation der bliver mødt, operationens variabler, samt det tupel space der manipuleres.

Topologien opbygges ved at man udruller oversættelses listen og lineært tilføjer den bolt der passer til operationen, med de tilhørende informationer fra listen. I den første model garanteres det at data altid bliver behandlet og for at teste det blev nogle tupler fejlet. SQL-space implementeringen gjorde det nemt at teste om data altid blev behandlet, fordi SQL-space implementeringen har et webinterface, hvor man kan se indholdet. Først startede jeg med kun at oversætte input og output operationer, og senere blev det udvidet så alt fra sKLAIM syntaksen kunne oversættes til Storm.

Henrik Pilegaard havde også et ønske om, at der var mulighed for at have en service tilknyttet til et tupel space. En service har ikke noget med KLAIM programmet at gøre, men det er udviklet og giver mulighed for at manipulere en tupel. Hvis en tupel sendes til et tupel space med en service, vil servicen først manipulere tuplen før den tilføjes til tupel spacet. Denne del er implementeret for at gøre systemet mere praktisk orienteret. De services der er implementeret i denne afhandling er simple, men man kan tænke sig avancerede services, hvor id'et fra en tupel anvendes til at gå på internettet og finde flere oplysninger om det pågældende id. Den nye fundne information tilføjes til tuplen.

Model 2 garanterer også at data altid bliver behandlet, men udover det lover den også at overholde præcis én gangs semantik. I Storm haves to forskellige koncepter til at garantere det, nemlig trident topologier og transaktions topologier. Begge koncepter er et højere abstraktions niveau til udførsel af tidstro behandlinger oven på Storm. Jeg arbejdede først rigtig meget med transaktions topologier, da de mindede en del om en normal topologi. Jeg fandt, så ud af at trident skulle være det nye og kastede mig derfor ud i det. Det var meget mere kompliceret, og efter en del arbejde med trident topologier valgte jeg at skifte tilbage til transaktions topologier. Dog er designet helt anderledes for en transaktions topologi ift. en normal topologi, da man opererer med et parti af

tupler pr. transaktion. Den første version af model 2 blev en simpel version, hvor det garanteres at et parti af tupler altid blev behandlet. Senere blev udviklet en version 2, hvor det garanteres at enhver tupel bliver behandlet én og kun én gang. Med den endelige model er det muligt at oversætte alt fra sKLAIM til Storm.

Nu da det endelige system var udviklet og det levede op til forventningerne kunne systemets robusthed, ydeevne og skalerbarhed testes. Under hele udviklingen og design af de forskellige modeller har det været klart for mig at SQL-space implementeringen var flaskehalsen. Test resultaterne underbyggede endnu engang SQL-space implementeringen som værende flaskehalsen. Derfor kunne man forestille sig en fremtidig model, hvor man ikke anvender SQL-space implementeringen. Og det giver anledning til fremtidigt arbejde.

7.1 Fremtidigt arbejde

Der er flere områder inden for denne afhandling, hvor der er mulighed for fremtidigt arbejde:

Ideen i denne afhandling anvender kun Storm, men der findes også andre systemer, som man kan anvende på samme måde, som jeg har anvendt Storm. Her kan blandt andet nævnes Hadoop [Fou13a]. Hadoop er open source og det supporterer data-intensive distribuerede applikationer. Man kan fremadrettet se på om, der er nogle andre og måske bedre muligheder ved at anvende Hadoop fremfor Storm. En sammenligning af disse to forskellige systemer ville give en endnu bedre indsigt i den distribuerede verden. Hadoop anvender noget de kalder for MapReduce, som er en programmerings model der anvendes til at behandle store mængder data med en parallel, distribueret algoritme i en klynge.

Det er nævnt før, men det mest oplagte at gå videre med er at fjerne SQL-space implementeringen, hvilket denne afhandling også ligger meget op til. Det vil være en klar forbedring af det nuværende system. Ideerne fremgår allerede af afhandlingen. Det vil være mest optimalt at behandle alt i Storm og systemets ydeevne vil blive klart forbedret. Systemet er kun blevet testet lokalt og der kunne også være en del arbejde at gå videre med, ved at se hvordan det behandler, hvis man opsatte en klynge af computere.

I Storm anvender jeg transaktions topologier og det er noget Storm udviklerne lader udgå, fordi det er forældet. Hele model 2 bygger på transaktions topologier, så for at forny koncepterne skal man i stedet for transaktions topologier ændre model 2, så den er tilpasset til koncepterne fra trident topologier. Et

fremtidigt arbejde kunne være at ændre model 2. Trident lover også én gangs semantik, ligesom transaktions topologier, og det er netop denne idé model 2 er opbygget omkring.

Til sidst, så kan man også udvide sKLAIM varianten. Blandt andet er der ikke tilføjet sammensætning af processer, $P_1||P_2$ eller aktivering. Jeg har valgt at holde mig til en simpel KLAIM variant og kun udvalgt det mest væsentlige. Ydermere kunne man forestille sig at tilføje evaluerings operationen, $eval(P)@\ell$, den denne heller ikke indgår i sKLAIM.

7.2 Diskussion

Til at runde af med fremhæves robustheden af systemet. Der kan opstå scenarier, hvor systemet brister. Vi anvender en database til at gemme systemets tilstand, og hvis en knude går ned, efter en tupel er behandlet, men før den har reporteret det tilbage til databasen, går det galt. Det er svært at undgå dette scenarie på den måde en transaktions topologi er opbygget i Storm. Hvis man kunne sørge for at behandlingen af en tupel sker på samme tidspunkt, som tilstanden opdateres i databasen ville det være løst. Dog er der sket en klar forbedring af transaktionerne, fordi vi i alle andre scenarier kan garantere at enhver tupel bliver behandlet. Dog garanteres det at en tupel altid bliver behandlet. Man kunne bytte rundt på det, så databasen først vil blive opdateret og tuplen derefter bliver behandlet, men så vil vi ikke garantere at en tupel altid bliver behandlet. Det er lidt optil hvad brugeren ønsker, hvilken metode der anvendes.

7.3 Konklusion

Jeg betragter projektet som en succes. Jeg har fået udviklet et system, hvor en sekvens af behandlinger vil blive udført én og kun én gang og hvor ingen transaktioner går tabt. Systemet lever derfor op til de ønskede forudsætninger opsat af Henrik Pilegaard fra Kapow Software. Det var ikke et krav fra start at SQL-space implementeringen ikke måtte indgå i det endelige system, men det er blevet analyseret at det er flaskehalsen. Fremtidigt arbejde kunne derfor være at få fjerne det ud fra de præsenterede ideer til en ny implementerings model.

I projektet har jeg undersøgt, om det var muligt at anvende KLAIM til at beskrive processerne, hvor transaktionsbegreberne bliver implementeret på ryggen

af et andet system. Det har vist sig at være muligt at oversætte en variant af KLAIM, så det kan behandles i Storm. Det færdig udviklede system indeholder to modeller. Model 1 garanterer at data altid bliver behandlet. Model 2 er en forbedring, hvor det også garanteres at data kun behandles én og netop én gang.

Use casen der er taget udgangspunkt i demonstrerer rigtig godt kræfterne af det udviklede system. Det er muligt at opskrive et program i KLAIM, oversætte det, for derefter at eksekvere transaktionerne i Storm. En Storm topologien kan køre for evigt og det er muligt at opsætte systemet, så det lytter på en data kilde. Så snart der tilføjes ny data til kilden vil Storm topologien behandle det i overensstemmelse med den beskrivelse der er lavet i KLAIM programmet.

Storm lever op til at løse de problemstillinger Kapow Software søgte efter i et system, og det underbygges i denne afhandling. Derfor har jeg ikke brugt tiden på at undersøge andre implementeringer. Jeg har fået udviklet to modeller, hvor model 1 blot er et skridt på vejen mod model 2. Model 2 er ikke en udvidelse af model 1, da koncepterne er helt forskellige i de to modeller. Jeg har derfor ikke brugt så meget tid på implementeringen af model 1, da fokus hele tiden har været at kunne garantere at en sekvens af behandlinger kun blev behandlet én og netop en gang.

Bilag A

Implementeringsdetaljer for en inputAction

A.1 Oversættelse/parsings delen

For at give en indsigt i hvad der sker når man møder de forskellige operationer i et KLAIM program, så vil der i denne sektion blive forklaret hvad der sker når man møder en inputAction i KLAIM-domænet til vi har oversat den til Storm termologi. Formen for en input-operation i et KLAIM program kan ses i listing A.1:

```
1    in(!x, !y:int)@main
```

Listing A.1: Input KLAIM kode

Oversat til ord betyder udtrykket at vi i main tupel spacet ser om der findes en tupel af formen <String, Int> og hvis det er tilfældet skal den udtrække en tilfældig tupel af denne type og assigne x med string-værdien og y med integer-værdien. Vi har her fire vigtige termer og disse stemmer overens med de fire felter fra ActionBolt-klassen: operations-navnet, tupel space-navnet, tupel-type-felterne og variablerne.

Kigger vi på parsing delen, så er det følgende regel der bliver machet i gramatikken når der mødes en input-operation, jvf. listing A.2:

```
1 inputAction ::=
2     INPUT LPAREN tuple:t RPAREN AT IDENT:id    {: RESULT = new
          InputAction(t, new VariableField(id)); :}
3     ;
```

Listing A.2: Matching regel for input

Der er implementeret en klasse i den udleverede java kode, kaldet InputAction og denne indeholder to private felter `_pattern` og `_locality` og vha. klassens-konstruktør bliver netop disse to felter sat når der i parseren mødes en InputAction - dvs. at tuplen, samt hvilket tupel space der inputtes i bliver gemt.

Nu hopper vi videre til næste skridt hvor vi evaluerer en inputAction. Da vi er inde i evalueringen af en inputAction, så vides det at operationen er en input. _locality feltet fra inputAction fortæller navnet på det tupel space der skal anvendes. Ligeledes bruges pattern fra inputAction til både at finde hvilke typer tuplen indeholder, samt de tilhørende variabler der skal assignes. Vi tilføjer de forskellige typer til en ITuple og variablerne bliver tilføjet til en liste bestående af strenge. En metode er blevet tilføjet `addToStormBolt`, som sørger for at tilføje de forskellige parameter til vores ActionBolt liste. Kilde-koden for evalueringen af en inputAction ses i listing A.3:

```
1  public void evaluate(InputAction inputAction) throws KlaimException
       {
2      final INode node = getNode(inputAction._locality);
3      print("Inputting from " + node.getName()+ " space");
4      final IKlaimField[] fields = inputAction._pattern._fields;
5      ITuple pattern = preparePattern(fields);
6      ArrayList<String> types = new ArrayList<String>();
7
8      for (int i = 0; i < fields.length; i++) {
9          IKlaimField field = fields[i];
10         if (field instanceof FormalField) {
11             String id = ((FormalField) field)._id;
12             types.add(id);
13         }
14         if (field instanceof VariableField) {
15             System.out.println("Skal implementeres TODO:");
16         }
17     }
18     addToStormBolt("input", node.getName(), pattern, types);
19 }
```

Listing A.3: Evaluering af inputAction

Når alle de forskellige operationer er blevet evalueret har vi konstrueret en liste bestående af ActionBolts med alt den nødvendige information.

Da vi tidligere argumenterede for at vi nøjes med at have én STORM topologi for et KLAIM-program, så kunne man blot nøjes med at lave en tom spout til at starte med, da vi i denne model ikke ønsker at tilkoble den til nogen kilde af input-data. Jeg har antaget at et KLAIM-program, skal starte med en output-operation som kickstarter programmet ved at hælde en tupel ind i et tupel space som der lyttes på. Det er implementeret således at den første output-operation altid oversættes til en spout i STORM. Ideen er så at der haves to forskellige typer bolts: én svarende til en input-operation og ligeledes én svarende til en output-operation.

A.1.1 Opbygning af STORM topologien

STORM topologien består altid af én output-spout, som er koblet sammen med én eller flere output eller input bolts. For at få opbygget STORM topologien bliver listen af ActionBolts loopet igennem. Vi matcher så operations navnet, input eller output, for at finde ud af hvilken type bolt der skal benyttes. I den første iteration bliver spout'en dannet og i anden iteration (i=1) er det vigtigt at den dannede bolt er koblet til spout'en. I listing A.4 ses den java-kode som er blevet implementeret for at tilføje de nødvendige bolts typer og forbindelserne mellem dem.

```
1 else if (boltMap.get(i).getActionName().equals("input") && i>1) {
2     Tuple tuple = ITupleToSQLINTuple(boltMap.get(i). getTuple());
3     builder.setBolt("TSBolt_"+i, new TSinBolt(boltMap.get(i).
          getActionName(), boltMap.get(i).getTsName(), boltMap.get(i)
          .getTypes(), tuple.shuffleGrouping("TSBolt_"+(i-1));
4 }
```

Listing A.4: Oprettelse af input bolt

Da vi tidligere fokuserede på inputAction, så er det igen denne som vil blive anvendt i eksemplet. Da STORM arbejder bedre sammen med SQL-spaces i forhold til lighTS, så bliver ITuplen konverteret til en SQL-tupel, som også minder mere om en Storm tupel. Det kan i koden fra listing A.4 ses at vi danner en ny `TSinBolt` med argumenterne _ActionName, _TsName, _tupel og _types. På denne måde bliver de nødvendige felter for en inputBolt sat via. konstruktoren. Da listen af ActionBolts bliver lineært udrullet, så kan vi nemt sætte forbindelserne ved at vide at bolt(i), skal lave en shuffleGrouping til bolt(i-1). Den ny dannede TSinBolt bliver tilføjet til builder, som er en TopologyBuilder og denne skal konfigureres for at få lavet en Storm topologi.

Når TSinBolt bliver kaldt, så sættes de private felter via konstruktøren. Hele essensen i en bolt bliver udført i execute metoden og som argument tager den en Storm tupel. Tuplen i argumentet indeholder de informationer som er blevet tilføjet fra den bolt/spout som den er forbundet med.

I KLAIM domænet, anvendes input til at tildele forskellige variabler og det er senere hen muligt at udskrive disse variabler til andre tupel spaces. Det er derfor vigtigt at man hele tiden har mulighed for at kunne tilgå disse variabler og deres tilhørende værdier. En topologi i STORM forbinder bolts/spouts ved at man sender en tupel fra en enhed til den anden. Tuplen indeholder blandt andet en liste af `Objects` og det er netop denne liste jeg har tænkt mig at anvende. Et hashmap anvendes til at gemme en variabel med dens tilhørende værdi i. Et hashmap gør det nemt at tilføje og udtrække variablerne og hvis man tilføjer en allerede eksisterende variabel med en ny værdi, så vil den gamle værdi blot blive overskrevet med den nye. I spouten danner vi derfor et nyt hashmap, som udsendes til den tilhørende bolt, som tilføjer eller læser fra hashmappet og videresender det til den næste bolt. Problemstillingen med at kende de tildelte variablers værdier er derfor løst. I listing A.5 ses den beskrevne kode for execute-metoden fra TSinBolt.java - dog skal det lige siges at det ikke er den fulde kode som vises. Execute-metoden looper først alle felterne igennem fra den tupel hvis private felt blev sat via. konstruktoren. Der er blevet dannet en ny input tupel, som får tilføjet nye (type.Class) felter og denne tupel anvendes til at søge i et tupel space efter en matchende tupel. Alle felterne i den matchende tupel bliver loopet igennem og variablerne, samt deres tilhørende værdier bliver tilføjet til hashmap. Hashmappet bliver tilføjet til listen af objects og sendes videre til den næste bolt.

```
1  public void execute(backtype.storm.tuple.Tuple tuple) {
2      List<Object> listObjects = new ArrayList<Object>();
3      HashMap<String, String> hashMap = new HashMap<String,String>();
4      Values emit = new Values();
5
6      TupleSpace ts = new TupleSpace(_tsName);
7      Tuple in = new info.collide.sqlspaces.commons.Tuple();
8      Tuple returnTuple = new Tuple();
9
10     for (int i = 0; i < _tuple.getNumberOfFields(); i++) {
11         if (_tuple.getField(i).toString().equals("<String>")) {
12             in.add(String.class);
13         } else if(_tuple.getField(i).toString().equals("<Integer>")
               ) {
14                 in.add(Integer.class);
15         }
16     }
17     returnTuple = ts.take(in);
18
19     for (int i = 0; i < returnTuple.numberOfFields(); i++) {
20         hashMap.put(_types.get(i), returnTuple.getField(i).toString
               ());
```

```
21      }
22      listObjects.add(hashMap);
23      listObjects.add(emit);
24      _collect.emit(tuple, listObjects);
25  }
```

Listing A.5: Execute metoden fra TSinBolt.java

Emit er anvendt som debug, da alt som er tilføjet til emit bliver vist i konsollen og det er på denne måde nemt at holde styr på hvad der bliver sendt imellem de forskellige bolts. Det skal lige nævnes at det er vigtigt at vi også sender input tuplen videre til den næste bolt for at garantere at data bliver fuldt processeret. Dette gøres ved at man har input tuplen som første argument i *_collect.emit*. Man giver derfor STORM mulighed for at detektere tupeltræet og via. ack/fail metoder bestemmes det om data bliver fuldt behandlet. Dette gøres ved at der sendes en ack eller fail til den tilhørende spout. Det er derfor meget vigtigt at huske og medsende input-tuplen for at garantere fejl-tolerance.

Bilag B

Kørsel af systemet

Første del:

For at eksekvere programmet skal man have først og fremmeste starte en SQL-space server. Det kan gøres ved at man går ind og downloader det her:

http://projects.collide.info/projects/sqlspaces/wiki/Download

pt. er følgende version tilgængelig: sqlspaces-server-4.0.0-beta-bundle2.zip. Når filen er downloadet skal den blot udpakkes, hvorefter man åbner filen: *startServer.bat*. Nu køres en SQLspace server, hvor det er muligt at logge ind og se hvilke tupler der haves i de oprettede SQL-spaces via http://localhost:8080/.

Anden del:

Projektet er samlet i en .jar fil med Eclipse. Da systemet er udviklet i Windows er denne guide også lavet til Windows, men det burde være lige til med andre styresystemer. Det kræver at man har installeret java på sin computere. Ved hjælp af commando prompt vinduet navigeres til mappen hvor Thesis.jar ligger. Dernæst kan man eksekvere programmet med følgende kommando:

```
1 java -jar Thesis.jar
```

Listing B.1: Standard kørsel af systemet

Programmet er implementeret, således at man har mulighed for at inputte nogle forskellige ting via. argumentet i kommandolinjen. Hvis brugeren ikke inputter nogle argumenter resulterer det i en eksekvering med følgende standard værdier:

```
String standard = "newloc(s1:sql)."+
                          "in(!name, !alder)@main."+
                          "out(name, alder)@s1.nil";


        int runtime = 10000;
        String topologi = "tran";
        String program = standard;
        String inputTupler = null;
        int batchsize = 3;
```

Listing B.2: Standard værdier i systemet

Det betyder at vi eksekverer programmet fra strengen *standard*, hvor programmet der oversættes til er en transaktions topologi med en parti størrelse på 3 der kører i 10.000ms og der fejles ikke nogle tupler. Der inputtes tre tupler til systemet: <hej, med>, <Lars, Anders og <Storm, Klaim>.

Til programmet haves følgende argumenter:

- -b, angiver antallet af tupler i et parti (standard=3).
- -f, procentsatsen for hvor ofte en tupel skal fejles(standard=0 procent).
- -i, anvendes til at angive filnavnet på den fil der indeholder tupler der skal inputtes.
- -k, anvendes til at angive filnavnet der indeholder det KLAIM program brugeren vil have oversat.
- -t, bruges til at vælge mellem en normal topologi og en transaktions topologi hhv. (norm eller tran).
- -r, angiver eksekverings tiden for topologien (10000ms).

Det er ikke muligt at oprette en service, da det er noget der skal udvikles i Storm. Så derfor skal man have kendskab til opbygningen af Storm og koden der er implementeret. Dog demonstrerer use casen brug af to forskellige services.

Koden til use casen kan ses i listing B.3, og skal blot tilføjes til en .txt fil og placeres på c-drevet. Hvor efter man kan inputte det ved at give argumentet $-kfilnavnet$.

```
1 newloc(facebook:facebook).
2 newloc(linkedin:linkedin).
3 newloc(database:sql).
4 in(!navn, !jobopslag, !ansogning, !formular, !CV)@main.
5 out(navn)@facebook.
6 in(!navn, !facebook)@facebook.
7 out(navn)@linkedin.
8 in(!navn, !linkedin)@linkedin.
9 out(navn, jobopslag, ansogning, formular, CV, facebook, linkedin)
      @database.nil
```

Listing B.3: use casen

Jeg har vedlagt en række eksempler, så det er muligt at lege lidt med systemet. Jeg har lavet to mapper, hvor den ene indeholder eksempler på KLAIM programmer for model 1 og den anden for model 2. De forskellige programmer kan findes i mappen eksempler. Det er nemt at køre en af eksemplerne, for model 1 kan man køre serviceProgrammet på følgende måde:

-java -jar Thesis.jar -k /Eksempler/Model1/serviceProgram -t norm

Model kører i et loop - det betyder at den samme topologi køres igen og igen indtil topologien lukkes ned.

Hvis man vil køre use casen, som er gennemgået i rapporten kan man nemt gøre det på følgende måde:

-java -jar Thesis.jar -k /Eksempler/Model2/usecaseProgram -i /Eksempler/-Model2/usecaseTupler

Det eneste det kræver er at stien til mappen med eksempler er: C:/Eksempler og at man kan eksekvere .jar filer via kommando prompt eller på anden vis. Selve koden kan ses på den vedlagte CD.

Bibliography

[AG13] Sven Manske Adam Giemza. Sqlspaces is an implementation of the tuplespaces concept that features many new ideas while keeping the api clear and simple. @ONLINE, February 2013.

[BBN+03] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gianluigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. In Corrado Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages pp 88–150. Springer Berlin Heidelberg, 2003.

[BDNP02] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.

[BNP01] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. X-klaim and klava: Programming mobile code, 2001.

[DBP07] Paolo Costa Davide Balzarotti and Gian Pietro Picco. The lights tuple space framework and its customization for context-aware applications. *Web Intelligence and Agent Systems*, 29(1):215–231, 2007.

[Fou13a] The Apache Software Foundation. The apache™ hadoop® project develops open-source software for reliable, scalable, distributed computing @ONLINE, February 2013.

[Fou13b] The Apache Software Foundation. Apache zookeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination @ONLINE, February 2013.

[Mar13] Nathan Marz. Storm - distributed and fault-tolerant realtime computation @ONLINE, May 2013.

[n/a13] n/a. Apache kafka - a high-throughput distributed messaging system. @ONLINE, April 2013.

[RDNP98] GianLuigi Ferrari Rocco De Nicola and Rosario Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 33(1):315–330, 1998.