Øving 4 - Distribuerte systemer

I denne øvingen har vi implementert vranglåsresolusjon i et transaksjonssystem. Denne rapporten vil gi en forklaring på de endringer som er gjort i rammeverket for å implementere dette.

Timeout

Den første løsningen er å implementere en timeout. Konseptet er da at dersom en transaksjon venter mer enn den tildelte mengden tid, så må den avbryte (abort) transaksjonen.

Systemet vet om det skal benyttes timeout ved å se på variabelen `PROBING_ENABLED` i `Globals.java`. Dersom denne er false vil vi benytte timeout.

Timeout er implementert i klassen `Resource.java`. I metoden `lock()` gjøres det en sjekk på `PROBING_ENABLED`. Er denne true, vil vi vente en gitt mengde tid, og sette haveWaited til true. Når programmet kommer til toppen av løkken sjekkes det om resursen nå er blitt tilgjengelig. Dersom den ikke har blitt ledig vil vi returnere -1, noe som vil føre til at transaksjonen blir aborted.

Dersom vi får lås på resursen får vi 1 returnert.

Edge Chasing

Konseptet er her at en server som ikke kan gå videre i sin transaksjon må sende en probe til de serverene som sitter på resurser den trenger. I denne proben legges det ved serverID. Når en server mottar en probe, vil den sjekke om den tidligere har hatt denne proben. I så tilfelle vil det være en sykel i behovsgrafen for resursen, og vi har dermed en deadlock. Når en server finner ut at det er en deadlock vil den kalle abort() på sin egen transaksjon. Dette fører til en NotifyAll() som vekker evt. transaksjoner som venter på resursen.

Når en serveren skal sende ut en probe oppretter den en ny tråd, og kaller metoden probe() på den serveren som holder resursen. Dette er altså et RMI-kall som går asynkront med resten av serverprosessen. På denne måten kan vi legge serveren i wait(), og vente på at resursen enten blir tilgjengelig, eller at vi får proben tilbake, og dermed må aborte.

Vi har også gjort en del endringer i lockResource() i klassen ServerImpl.java. Vi gjør her en sjekk på om serveren forsøker å låse en transaksjon som den selv har satt en lås på. Dersom dette skjer trenger vi å kalle abort(). Denne sjekken gjøres i utgangpunktet av rammeverket, men vi trenger å gjøre den på et tidligere stadium.

Dersom en annen server sitter på resursen får vi result = 0, og vi går inn i loopen. Her finner vi ut hvem vi skal probe, sender proben til den aktuelle serveren, og går til wait(). Når vi så mottar en notify vil vi på nytt forsøke å få resursen.

Kjøringer

Timeout
Her finner vi det mest hensiktsmessig å eksperimentere med timeout-tiden. I de første kjøringene har vi hatt en tid på 3000 ms, noe som har ført til enkelte lange stop i kjøringen. Med få servere er det ingen problemer med å senke timeout-tiden. Dersom vi har mange servere vil en for lav timeout føre til at en server burde vente lenger enn den tilmålte tiden. Når den nå våkner for tidlig, vil den aborte for tidlig, men hadde den ventet lenger ville transaksjonen kunne ha gått igjennom.

Valget av 3000 ms virker fornuftig da det er relativt effektiv, men samtidig ikke gir for mange aborts.

Edge Chasing
En utfordring med Edge Chasing er en situasjon hvor en transaksjon venter på en annen, som igjen er avhengig av den første. I dette tilfellet vil begge transaksjonene sende ut prober, og vi kan komme i tilfellet hvor begge transaksjonene velger å avbryte. Dette er unødvendig, da en av transaksjonene kunne ha fullført når den andre avbrøt.

Edge Chasing er mer effektivt enn timeout, da vi kun venter når det er behov for det, og vi får tilbakemelding når resursen er tilgjengelig igjen. Metoden skalerer også bedre, da metoden fungerer godt på en større mengde servere, og vi må ikke, som i timeout, gjøre tilpasninger ettersom mengden servere øker.