Distribuerede Systemer

Afleveringsopgave - 5 - uge6

Hold: DA4; Gruppe: 2

Emil Madsen - 20105376
Rasmus - 20105109
Sverre - 20083549

8. marts 2012

Introduction

Vi skal i denne uge lave et distribueret regneprogram, der holder sessioner konsistent for en gruppe servere.

Regneservere i gruppen skal forekomme ens for alle brugere, uanset hvilken server i gruppen man forbinder sig til.

Details

Vi har benyttet det udleverede kode sammen med en del af vores kode fra opgave 11, navnligt vores modificerede pointtopointqueue (for dynamisk port) og vores multicastqueue (FIFO og Total). For disse ændringer henvises til tidligere opgaver.

I det udleverede kode har vi tilføjet en klasse 'ServerDistributed', der virker i stil med 'ServerStandalone', men er, som navnet antyder, distributeret. Denne gør bl.a. brug af vores total queue.

Derudover har vi mht. debugging implementeret en 'debugVisitor', der printer samtlige visits den modtager. Til sidst men ikke mindst, har vi implementeret vores egen ServerTUI, kaldet 'ServerTUIDistributed', der ligner den almindelige ServerTUI, men benytter sig af vores underliggende distributerede server.

Implementation

ServerDistributedTUI

Som nævnt har vi lavet en 'ServerDistributedTUI'. Dette er vores server's'main'. Den sørger for at instansere et 'ServerDistributed' objekt og initialisere dette med den information, som brugeren skriver på commandline (som svar på TUI'en spørgsmål).

Eksempelvis beder TUI'en brugeren om addresse og port på den server, der skal joines, hvorefter tui'en parser dette og giver det videre til vores distributerede server klasse.

Derudover er der TUI'en, der tager imod commandline input fra brugeren, og håndtere dette (f.eks. exit og crash kommandoerne).

ServerDistributed

Vores 'ServerDistributed' er den klasse, som vores TUI benytter til alt det tunge arbejde.

Den fungerer således at den, når den initialiseres, åbner en MultiCastQueue, til de andre servere (hvis den joiner, ellers laver den en selv).

Denne MultiCastQueue bruges da til at lade serverne tale sammen i total orden, således at beskeder kommer alle steder hen 'samtidigt'.

Udover dette åbner den også for adgang til klienter, dog på en anden port. Således kan den tage imod klienternes requests.

Når en klient forbinder til serveren og sender et request, f.eks. om at en variabel skal sættes, sætter serveren faktisk ikke variablen med det samme: den sender blot en kopi af denne besked op på MultiCastQueuen.

Når en server modtager en besked på MultiCastQueuen (fra sig selv, eller andre), ved vi efter kontrakt med MulticastQueue'en at de andre servere også modtager denne besked netop nu. Derfor kan vi altså håndtere beskeden når vi modtager den fra MultiCastQueuen, og det er også her vi gør det fx ved at sætte en bestemt variabel på serveren.

Desuden tjekker serveren på dette tidspunkt om det er den, der ejer klienten, og, hvis det er tilfældet, svarer denne server tilbage til klienten med et acknowledgement eller lignende.

På denne måde holdes alle servere synkroniserede (såfremt at alle joiner, før nogen klienter gør noget).

For at kunne joine servere efter klienterne har været i gang, har vi valgt at benytte vores HistoryDecorator fra en tidligere aflevering.

Ideen er nu at når en server forbinder til MultiCastQueuen, starter den med at modtage samtlige klient-beskeder, der har været sendt før jointidspunktet, således at serveren nu kan processere disse og dermed blive synkroniseret med de andre.

Helt implementerings-mæssigt, udfører 'ServerDistributed' sin opgave efter intialisering ved at oprette 2 tråde: 1 tråd, der modtager fra multicastqueuen, og 1 tråd, der modtager beskeder fra klienterne.

Klient-tråden modtager blot beskeder og smider dem op på multicastqueuen, som tidligere nævnt.

MultiCastQueue tråden modtager beskederne og sender dem videre til diverse event listeners, der så tager sig af at processere beskeden. De vigtigste eventlisteners er 'calculator' og 'connectionManager'.

Calculator

'calculator' klassen, er en intern klasse i vores 'ServerDistributed', der har til ansvar at processere events, og opdatere serverens interne variabel tabel. Eksempelvis kan den modtage et 'add' event, hvorefter, den så sammenlægger to variabler fra tabellen, og gemmer dem, ind i tabellen igen. Ligeledes understøtter den også andre matematiske operationer.

ConnectionManager

'connectionManager'en er en speciel klasse, der allerede er blevet løst nævnt, det er nemlig denne, der benyttes til at sende svar tilbage til klienten, denne kaldes kun på den server, der ejer klienten, og den tager sig bl.a. af connect, disconnect, og acknowledge beskeder.

Krav

Herunder er de opstilte krav og hvordan vi har opfyldt dem.

Ability to run at least 4 servers at 3 different machines and tolerate at least 3 clients

Vi har testet dette, ved at starte 4 servere og forbinde dem. Derefter har vi forbundet 3 klienteer til hver server og tjekket at samtlige kan se hinandens ændringer.

Dette kan opnåes lokalt, da der bruges dynamiske port bindinger.

Using Ex11 to maintain consistancy

Vores design fungerer således at enhver server kan holder en mængde af klienter, som den tager sig af. Den lytter således på en bestemt port, og venter på at klienterne sender requests til denne.

Når serveren modtager en request, processerer den ikke requesten. Serveren forwarder derimod Requesten til en TOTAL MultiCastQueue (Den der blev udvilket til Ex11). Dette sikrer da at alle klient-requests bliver afleveret på samme måde til alle servere.

Når en server modtager en request fra MultiCastQueuen, kan den altså antage at alle andre server også har modtaget denne, hvormed den kan ændre sin interne datastruktur til at afspejle dette (I samme stil som Standalone serveren gør det). På denne måde vil serverne altså altid holdes konsistente.

En lille detalje er dog at der skal sendes et acknowledgeEvent tilbage til klienten, og da ikke alle servere er forbundet til alle klienter, er det kun den oprindelige server, der kan udsende det (Serveren der oprindeligt modtog klientens request). Derfor tager noget specifik kode sig af at finde frem til hvem der 'ejer' klienten, således at denne server kan sende et acknowlegdeEvent tilbage.

I samme stil, tager den specifikke server sig også af join og leave for sine klienter.

Vi har testet at dette krav er opfyldt ved at starte to servere og så sende variabler til den ene og tjekke at disse kan læses i den anden (og omvendt).

Tolerate that any server with no client leaves

Dette er intet problem, da den kun er forbundet til de andre servere igennem MultiCastQueue, og denne nemt kan afbryde forbindelsen vha. leaveGroup().

Tolerate that a new server joins, after other servers has run

Dette var en mindre udfordring, da det kræver at den nyligt join'ede server bliver initialiseret til en tilstand, der tilsvarer tilstanden i de andre servere. Vi

har valgt at gøre dette, ved at benytte vores MultiCastQueueHistory decorator. Således vil en historik, der indeholder alt hvad der er sket siden starten, bliver sendt til den nye servere meget hurtigt.

Vi har testet at dette krav er opfyldt ved at starte en server, sætte nogle variabler og derefter forbinde en anden server og tjekke at denne så kender de givne variablers værdi.

Guarantee all four client-centric consistency flavours

Vi har opfyldt alle fire krav, ved brug af total queuen (da alle operationer går igennem denne, og dermed bliver udført i en strengt total orden).

Vi kan også sige at de fire 'client-centric consistency'-krav alle opretholdes pga. at det altid kun er en tråd, der tilgår den datastruktur, der indeholder variablerne med navne og værdier.

monotonic reads: Vi overholder monotonic reads, fordi læsninger umuligt kan læse en ældre værdi fra datastrukturen end den mest nyelige. Vi beholder altså ikke gamle værdier.

monotonic writes: Vi overholder monotonic writes, fordi der kun er en tråd som skriver til datastrukturen og denne arbejder serielt: Værdier skrives færdigt før de kan overskrives.

read your writes: Vi overholder 'Read your Writes' fordi det altid er den samme tråd, der læser og skriver til datastrukturen. Dermed vil en skreven værdi altid medføre en mulig læsning af den værdi (eller en senere skreven værdi).

write follow reads: Vi overholder 'Write follow reads', af samme grunde som vi overholder de tidliger consistency krav: kun en process tilgår datastrukturen og alle tilgange færdiggøres før en ny begynder.

Questions

Derudover skulle vi besvare følgende spørgsmål;

1. Do your implementation tolerate that two servers without clients leave at the same time? Why / Why not?

Da det eneste, der forbinder vores servere, er multicastqueuen, og da denne nemt kan afbryde forbindelsen, også concurrently, kan det sagtens lade sig gøre at afbryde forbindelsen fra to servere samtidig.

2. Do your implementation implement the semantics of an atomic section? If it does, how did you implement it?

Vi har ikke implementeret atomic section, men vi kunne hurtigt gøre det, ved at lave en speciel message type, der blot holder en liste af events, der så modtages som én, og dermed også udføres som én.

Usage Instructions

Kompilering

Ant can bruges til at kompilere koden. Derefter kan man bruge java i terminalen fra build folderen.

Dvs. man kan starte vores regne server med 'java ServerTUIDistributed' og clienten kan startes med 'java ClientTUI'

Eksempel for at kompilere og køre i Windows:

```
...\kode>ant build
Buildfile: ...\kode\build.xml

prepare-build:

build-src:

build-test:
    [javac] ...

build:

BUILD SUCCESSFUL
Total time: 0 seconds

...\kode>cd build

...\kode\build>java ServerTUIDistributed
Exit: Gracefully logs out
Crash: Makes the server crash.

Enter address of another server (ENTER for standalone):
```

Alternativt, kan man køre de to bash scripts 'server.sh' og 'client.sh' for at køre de tilsvarende TUI's.

Når en server startes, er der en forespørgsel om en server-adresse. Hvis man undlader at skrive en adresse og blot trykker ENTER, vil serveren lave en gruppe self.

For at forbinde til en eksisterende gruppe af servere skal man indtaste `<address>:<port>`.

Da porte oprettes dynamisk, skal man være opmærksom på at forbinde til den server port, som er blevet bundet i den kendte peer.

I ClientTUI bliver man ligeledes spurgt om en server adresse. Her skal man igen bruge `<address>:<port>`. Bemærk at dette er en anden port end 'server til server' porten.

Server opsætning

Den første server der startes (den der skal lave server gruppen), skal ikke gives noget argument, ved beskeden;

```
Enter address of another server (ENTER for standalone):
```

Her skal vi altså simpelt nok bare klikke '<ENTER>'.

For en server der skal joine en server gruppe, skal vi på givne tidspunkt skrive server group addressen på den server vi ønsker at joine her; Denne findes nemt ved at læse outputtet fra en allerede kørende server, og ser noglelunde således ud;

```
...
Created/Joined server group at: llama04/10.11.82.4:53261
...
```

Det er netop denne addresse der skal bruges, til at blive en del af server gruppen. Dette skal indtastes med både addresse og navn, på det sædvanlige '<address>:<port>' format. Man skal huske at kigge efter beskeden; 'Server running', der indikere at serveren nu faktisk køre.

Klient opsætning

For at kontakte til en server med en client skal man således også kende noget information fra serveren, nemlig på hvilken port, den specifikke server, servicere clienter, denne information er ligeledes nem at læse, og ser noglelunde således ud;

```
...
Listening for clients at: llama04/10.11.82.4:46814
...
```

I clienten skal denne addresse, skrives på samme format, som tidligere, når beskeden;

```
Enter address of a server:
```

Vises på skærmen, herefter vil clienten udskrive information om, hvorvidt det lykkedes at kontakte til serveren eller ej. Noter venligst, at client porten, er anderledes en servergroup porten.

Conclusion

Vi konkluderer at vi har løst opgaven tilfredsstillende, da vi opfylder alle de opstilte krav.