02342 Distribuerede systemer Assignment 3

INDIVIDUEL PROGRAMMERINGSOPGAVE

Christian Budtz s134000

Indledning

I denne opgave skulle vi implementere en middleware platform, der understøtter styring af en intelligent bygning. Jeg har i min løsning forsøgt at koble publisher og subscribers så løst som muligt - også hvad angår teknologi hos hhv. subscriber og publisher.

Kravspecifikation

I opgaven er kun specificeret få krav, hvilket efterlader rige muligheder for kreativitet. Løsningen skal dog som minimum understøtte publish/subscribe paradigment og funktionerne publish(event) samt subscribe(event). Desuden skal kommunikationen mellem serveren og brugergrænsefladen fortsat ske med java rmi. Løsningen skal modellere et pub/sub system, der skal kunne servicere et intelligent hus.
Det er ikke beskrevet hvilke typer af events, som systemet skal kunne håndtere (tidligere havde vi temperaturmålinger), eller hvilken type af publishers eller subscribers, der kan forventes at tilkoble sig middlewaret.
Der er ikke stillet krav til usability, reliability, performance eller supportability.
For at et system skal være velfungerende i et intelligent hus, er det dog oplagt at elaborere kravspecifikationen noget.

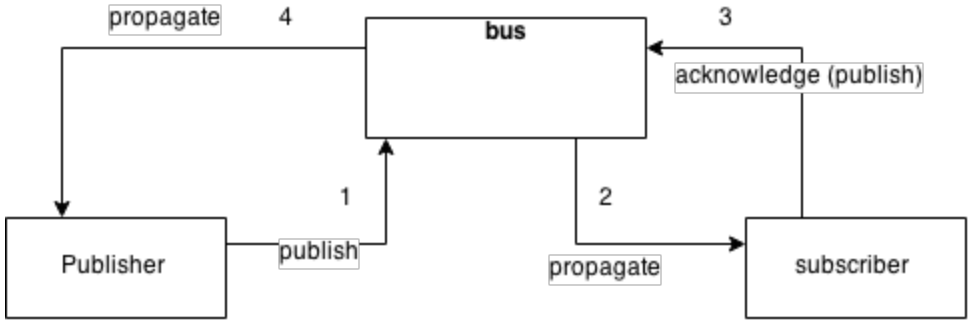
Usability

For at systemet kan afprøves, er det nødvendigt at implementere minimum én subscriber og én publisher. Oplagt er det at genanvende en temperatursensor-dummy og RMI-clienten fra tidligere. Det kan være svært at gennemskue flowet fra sensorer til temperaturgennemsnittet, hvorfor en form for logning af event-udbredelsen igennem systemet er ønskværdig.

Reliability

Et distribueret system er komplekst og kan fejle mange steder - Sensorer kan løbe tør for batteri, klienter kan miste forbindelsen,m.m. En løs kobling kan i nogen grad forhindre at en fejl får systemet til at gå ned, men kan ikke forhindre at fejl propageres i systemet - her er det nødvendigt med redundans.
I et intelligent hus, vil der kunne optræde en heterogen gruppe af publishers. Der kan eksempelvis være tale om sensorer (temperatur, fugtighed, åbne/lukke). Data fra sensorer kan være af mere eller mindre kritisk natur. Tabes enkelte temperatur-events, vil subscribers formentlig kunne fortsætte uden mærkbare fejl. Tabes information om at et vindue er blevet åbnet risikerer man at blive bestjålet fordi tyverialarmen ikke går af.
Publishers kan også være styringsenheder, der integrerer input og påvirker andre subscribers. Et eksempel kan være et lysstyringsmodul, der integrerer data om tusmørke og beboerens mobiltelefonplacering og herfra sender events til husets lamper om at tænde/slukke.
Således kan events have varierende grader af kritikalitet. Vil man garantere at events kommer frem, kan man anvende/konstruere en protokol, der garanterer at data afleveres (eks. TCP). Det vil til gengæld skabe et overhead, der skaber ekstra netværkstrafik. Temperatursensorer kan tænkes at være batteridrevne, og det ekstra overhead, som en sikker protokol skaber kan koste dyrt på batterilevetiden.
En alternativ løsning kan være at understøtte begge typer af events - altså et sikkert og et

ikke-sikkert publish/subscribe system i samme netværk.
I den nærværende løsning, har jeg valgt at designe en løsning, der ikke implementerer en garanti for modtagelse af events. Et sådant system vil således ikke være velegnet til events, hvor det skal garanteres at de når frem. Funktionaliteten kan dog bygges ovenpå - eksempelvis ved at en publisher subscriber på acknowledgements og subscribers afsender en acknowledgement ved modtagelse af en event - på samme måde som TCP lignende funktionalitet kan bygges ovenpå UDP.



Sikker aflevering af events. En publisher publicerer et event, der propageres til subscribers, der igen svarer at de har modtaget eventen. Løsningen kræver at publisheren ved hvilke subscribers der skal svare på et event.

Vigtigt er det dog at subscribe() og unsubscribe() beskeder når frem - uden garanti for dette, risikerer en subscriber at misse publications fordi beskederne ikke når frem.
I en pub/sub arkitektur er publishers og subscribers løst koblet gennem middleware-laget. Middleware-laget sørger for at beskeder når frem. Forsvinder publishers eller subscribers fra systemet, kan det fortsætte med at køre - med den manglende funktionalitet som det afføder. Bryder middleware-laget ned ophører systemet til gengæld med at fungere. Konstruerer man middlewarelaget med én eventbroker, der modtager alle events og videresender dem til subscribers, har man et single point of failure, der kan bringe systemet i knæ. Løsningen er den simpleste at implementere, men vil i et system, der kræver høj oppetid, formentlig være for usikker.
For at gardere sig mod totale nedbrud, kan man skabe redundans i middlewarelaget - eks. form af flere brokers. Vil man også kunne gardere sig mod en broker, der videresender fejlagtige beskeder, er det nødvendigt med minimum 2k+1 brokers for at håndtere op til k fejl ved at lade flertallet bestemme. Det stigende antal brokers vil til gengæld skabe ekstra trafik på netværket og skabe behov for protokoller, der håndterer kommunikationen indbyrdes. En mulig vej til at skabe større redundans kunne være at lade middleware laget være sammensat af alle nodes - både subscribers og publishers og distribuere informationen om subscribers til alle nodes - evt. som en distribueret hash table - altså en form for p2p-teknologi. På den måde skaber man større redundans, jo flere nodes (og dermed større kompleksitet) der tilføjes netværket. Denne løsning vil igen skabe et større behov for kommunikation mellem knuder med et dermed tilsvarende stigende energiforbrug. Man kunne evt, håndtere det øgede energiforbrug ved kun at lade knuder, der er el-net forbundne danne middlewarelaget - eller subsidiært kun de knuder, der har nok strøm tilbage på batteriet.
Vælger man en løsning med hvor bus-funktionaliteten er redundant og måske endda distribueret, bliver det nødvendigt at finde ud af hvordan knuder forbinder sig til netværket.

Man kan evt. vælge et fast område af IP-adresser hvor knuderne kan findes - eller hvis det foregår indenfor et sub-net kan man vælge at UDP-broadcaste på en foruddefineret port og lade aktive brokers svare på denne.
I et distribueret netværk er det desuden ikke garanteret at beskeder sendes og modtages i en ordnet rækkefølge - eller at alle subscribers modtager alle events. Er det nødvendigt med 'ordnede forhold' er det nødvendigt at implementere protokoller til synkronisering - eks. hvis man vil sikre sig at dørlåsen låste *EFTER* at døren blev lukket.
Det kan også være vigtigt at events, der er afsendt før en subscriber sender sin subscribe besked, ikke sendes til subscriberen eller omvendt at events ikke afsendes efter at unsubscribe beskeden er afsendt. Her kan en løsning være at anvende vector clocks til at etablere en rækkefølge af events.
Den løse kobling betyder desuden at der ikke automatisk er en kontrol af om subscribers eller publishers er online. Over tid risikerer man at publishers crasher uden at give en besked om unsubscribe og dermed efterlader en masse 'døde abonnementer'. For at imødegå dette vil det være nødvendigt at implementere en protokol, der spørger om publishers er i live og venter et passende interval før deres subscription fjernes.

Performance

Så længe der er tale om et mindre hjemmenetværk af nyere dato, vil den trafik der genereres af sensorer og aktuatorer være begrænset. Kun hvis der sendes anden data (streaming, filkopiering), kan der opstå flaskehalsproblemer og deraf packet loss på netværket - med nedsat performance af systemet som følge. Her kan quality of service politiker afhjælpe problemet. Processeringen af pakker vil ligeledes næppe skabe forsinkelser - de fleste eventtyper vil formentlig begrænse sig til meget lidt data - næppe større end end maksimumstørrelsen på UDP pakkerne.

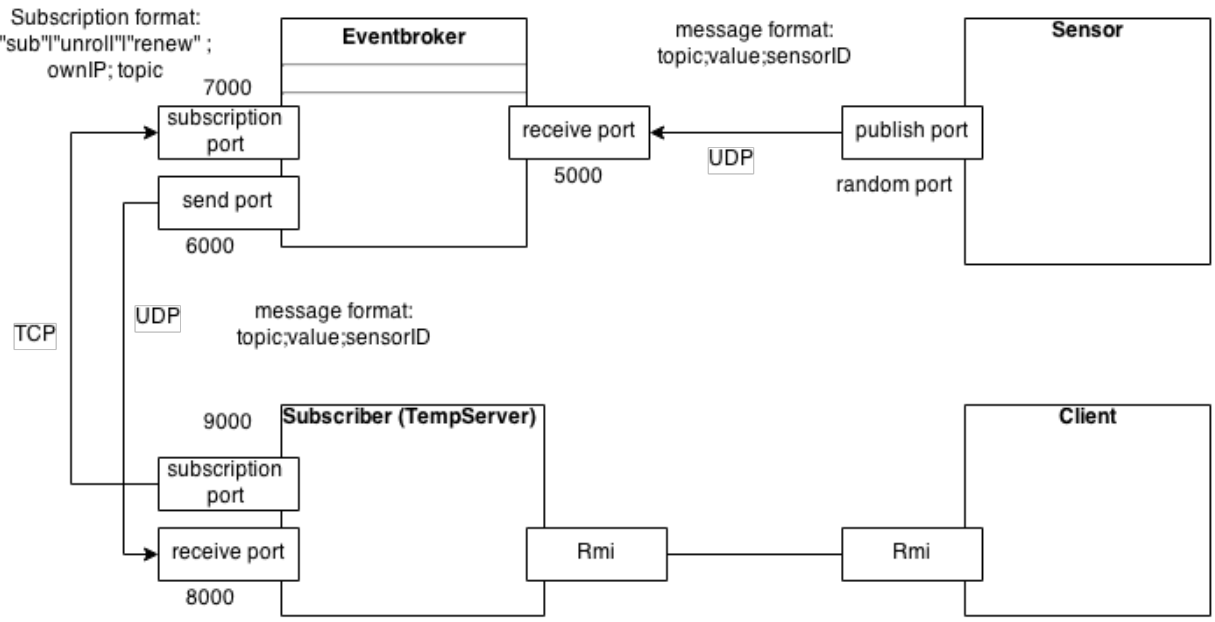
Supportability

I opgaven er lagt op til at der anvendes rmi til brugergrænsefladen. Det kan senere vise sig at være uhensigtsmæssigt, da rmi teknologien muligvis er på vej ud og man er bundet af teknologien i begge ender af forbindelsen. Et alternativ kunne være en RESTful webservice og JSON over HTTP, der ville afkoble afhængigheden af java. Vil man gerne implementere en letvægts komponent (eks. en arduino med display og knapper til at justere temperaturen), vil det ekstra overhead til en jvm desuden gøre det meget besværligt. I opgaven er lagt op til at man skal understøtte subscribe() og publish() funktionen. Jeg har valgt at tolke det, som at middlewaret skal kunne tage imod en subscription (i et ikke defineret format) og en event der publiceres og ikke nødvendigvis som et rmi kald.

Design

Ud fra mine omfattende overvejelser om mulige implementeringer af middleware-laget besluttede jeg at designe et middleware-lag, der som nævnt var så teknologi (java) uafhængigt som muligt. En java binding ville skabe problemer for de tilkoblede publishers og subscribers idet de skulle implementere en jvm - der igen ville skabe et unødvendigt overhead.
Jeg valgte en løsning med én event broker. Dette gav en simpel implementering og i den givne kontekst (det intelligente hus) vil det formentlig være acceptabelt at have et system, der kan være nede i kortere perioder (man bør dog nok finde en anden løsning til tyverialarmen). Da

kravspecifikationen desuden var meget begrænset, betragtede jeg det som en prototypeløsning og valgte den simple vej igennem.



Forbindelser mellem de forskellige delkomponenter i systemet. Sensorer sender events til brokeren over UDP. Events videresendes til relevante subscribers ligeledes over UDP. Da subscriptions ikke bør gå tabt, forbinder subscriberen til Eventbrokeren med TCP. Klienten der gerne vil have en gennemsnits temperatur, forbinder til temperaturserveren med rmi.

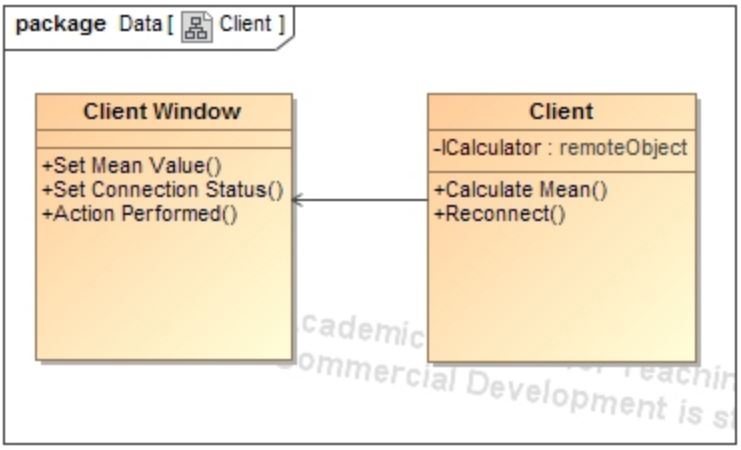
Protokoller

Jeg valgte UDP til at sende events til event brokeren og videre til subscriberen, da jeg valgte at implementere en løsning, hvor jeg ikke garanterer for at events viderebringes. UDP er en stateless letvægtsprotokol med meget lille overhead og dermed lille energi forbrug. Jeg valgte TCP til subscriptions for dog at garantere at subscriptions når frem. For overskuelighedens skyld, har jeg brugt én separat socket til hver forbindelse - selvom UDP og TCP sockets kan bruges til full duplex.
For at afkoble rmi fra event brokeren, valgte jeg at opsætte en temperatur-server, der optræder som subscriber. For fortsat at leve op til kravet om at anvende rmi til brugergrænsefladen, kommunikerer klienten fortsat med temperatur-serveren via rmi.
Tanken er at min temperatur-server modsvarer en temperaturstyringsenhed, der modtager temperatur-events og integrerer data til at styre solafskærmning, radiatorer/varmepumper, air condition osv. via events den selv publicerer. For at indstille temperaturstyringen tilkobler brugeren sig via rmi - enten på et mobilt device (der kan køre rmi - eks. et android device) eller måske via et smart TV.
For at undgå at subscribers forlader netværket uden en unsubscribe besked (eks. ved et crash - eller brud på netværk) og efterlader en død subscription, der får brokeren til at videresende pakker til subscriberen selvom den har forladt netværket, har jeg tilføjet et krav til subscribers om at de som minimum skal subscribe eller renew'e minimum hver time - gør de ikke dette, fjernes alle deres subscriptions fra eventbrokeren.

Når en subscriber ønsker at subscribe, forny sin lease eller unsubscribe, skal det ske med en streng sendt over TCP og formateret i UTF-8. Strengen skal være sammensat af en beskedtype ("sub","renew" eller "unroll"), en ip-adresse (man kan sende subscriptions for andre) og et topic adskilt af ";".
En event skal ligeledes være en streng, der er formateret i UTF-8, men den skal sendes som UDP-pakke. Strengen skal være sammensat af topic, value og sensorID og igen skal de være adskilt af ";".

Klient

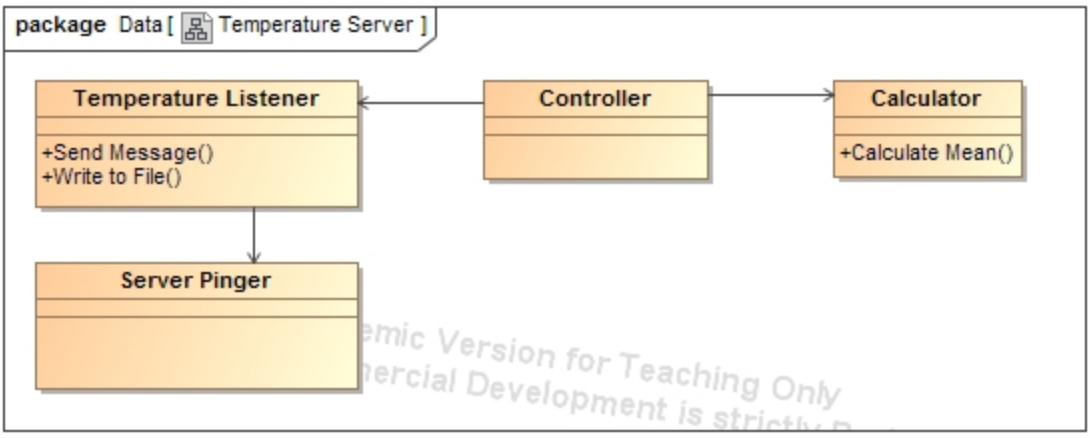
Klienten består af en controller og et view. View'et præsenterer data til brugeren og viser om man er tilkoblet en rmi-server. Her er mulighed for at hente gennemsnitstemperaturen fra serveren via et klik. Klienten tillader at forsøge at genforbinde til serveren hvis forbindelsen går tabt.



Rmi klient design diagram. Kun væsentlig funktionalitet er medtaget.

Temperaturserver

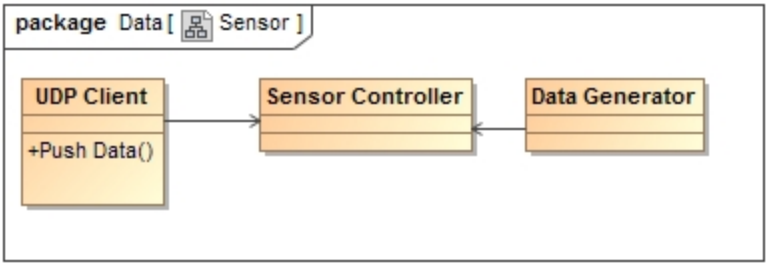
Består af 2 delkomponenter, der kører i hver deres tråd. RMI serverkomponenten (Calculator) vedligeholder forbindelse med eventuelle klienter. TempListener komponenten lytter efter nye events over UDP. TempListner spawner desuden en tråd der sørger for at oprette en subscription og fornyer subscriptions hver 5 sekunder over TCP. De to delkomponenter tilgår data via en statisk hjælperklasse - PropertyHelper der sørger for at tilgangen til data på disken er beskyttet med en read/write lock - således at listeneren kan skrive og calculatoren kan læse uden konflikter.



Temperatur server.

Sensorer

Består af to delkomponenter der startes i hver deres tråd af en controller. Controlleren holder en liste af temperaturdata i en kø. Køen er 'volatile', således at samtidige skrive og læse operationer ikke skaber korrupte data. Datagenerator tråden genererer tilfældige måledata for at simulere en temperatursensor, der sendes til controlleren. UDP klientdelen holder øje med datalisten i controlleren og afsender data, når de er tilgængelige.

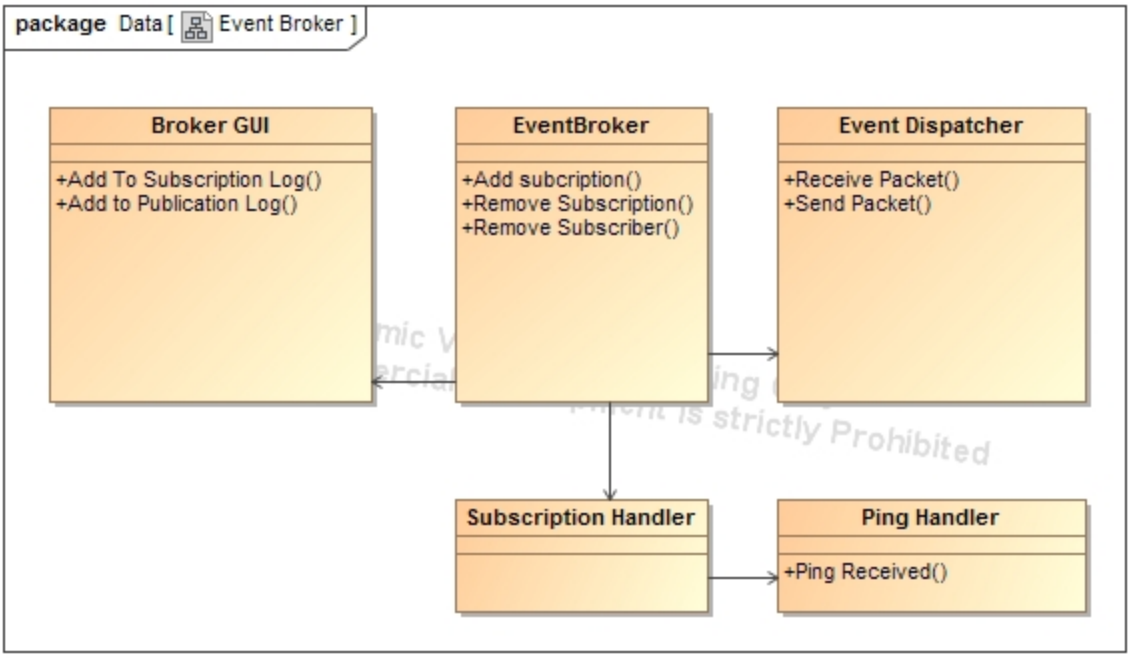


Sensor.

Event broker

Event brokeren har 2 hovedkomponenter - en event dispatcher der lytter på nye pakker på en udp socket og videresender dem og en subscriptionhandler der opdaterer subscriberlisten. Selve controlleren (Broker) holder et hashmap af linkedlists, hvor key er topic og value er en liste over subscribers for det topic. Det betyder at man i konstant tid kan finde listen af subscribers for et givent topic. Listen kan så gennemløbes og alle subscribers modtage en kopi af en relevant pakke.
Event dispatcheren gør netop dette. Når den modtager en udp pakke med et givent topic, slås subscriberne op og alle sendes en kopi af pakken.
Subscriptionhandleren håndterer subscribe(), unsubscribe() og renew(). Modtages en subscribe() besked tilføjes subscriberen på subscriber listen. Subscription handleren har en privat classe der kører i sin egen tråd - Pinghandler. Hver time tilføjes alle subscribers til en liste over tråde der er markeret til oprydning. Fjerner subscription handleren ikke subscriberen fra oprydningslisten inden for en time, annuleres alle subscriberens subscriptions. Subscriptionhandleren fjerner en subscriber fra oprydningslisten hvergang den hører fra den - enten via subscribe eller renew.

For at kunne følge med i aktiviteten på eventbrokeren har jeg tilføjet en gui med to logvinduer, der viser trafikken i eventdispatcheren (højre vindue) og i subscriptionhandleren (venstre vindue.

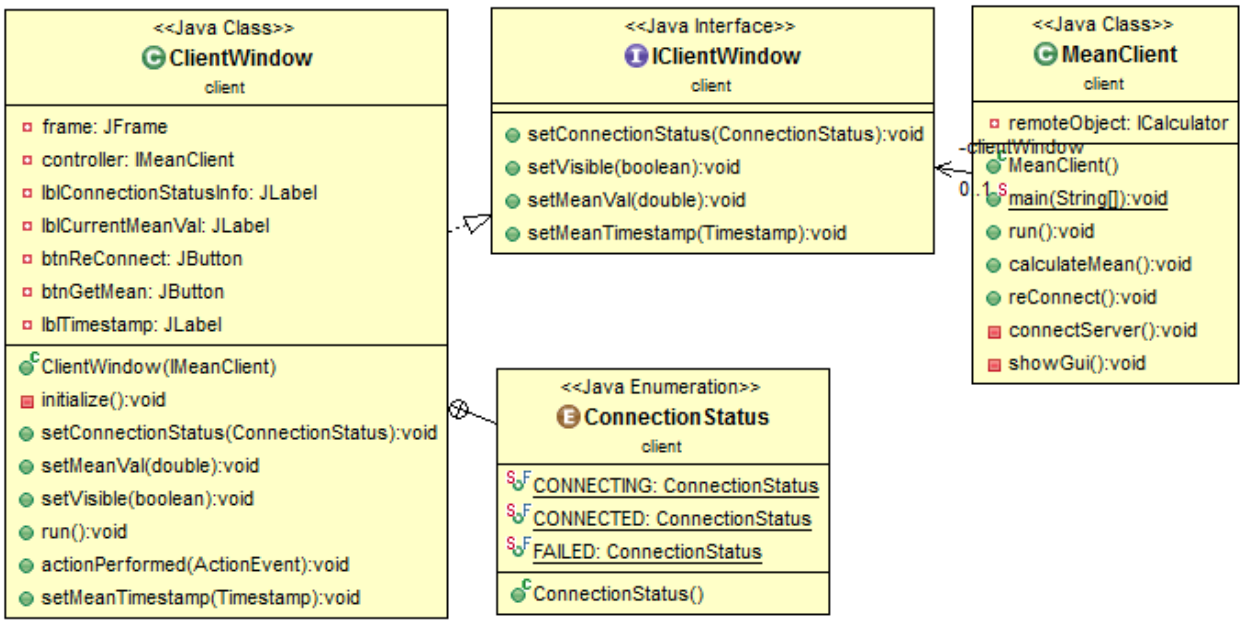


Event Broker.

Implementering

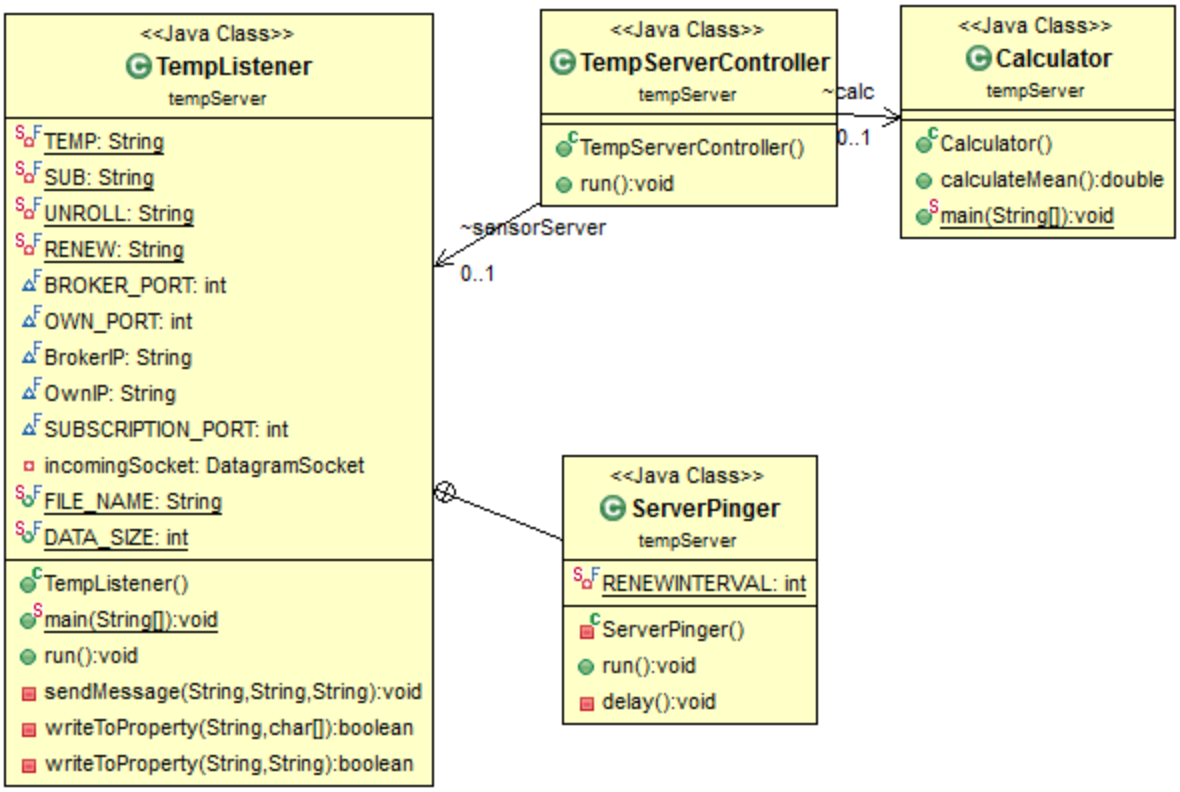
Klient

Klienten er implementeret med en MeanClient klasse der instantieres og forsøger at forbinde til en rmi server. Den modtager et remote object: Calculator, hvorpå man kan kalde calculate mean. Når brugeren trykker på Get Mean, fanges eventen og metoden calculateMean kaldes på MainClient, der igen kører calculateMean() på remote objektet. Herefter returneres gennemsnittet, der kan sendes tilbage til gui'en og vises til brugeren. Mistes forbindelse til rmi serveren, kan brugeren trykke på reconnect, hvorefter kaldet propageres til MeanClient, der vil forsøge at genetablere forbindelsen via reConnect() metoden.



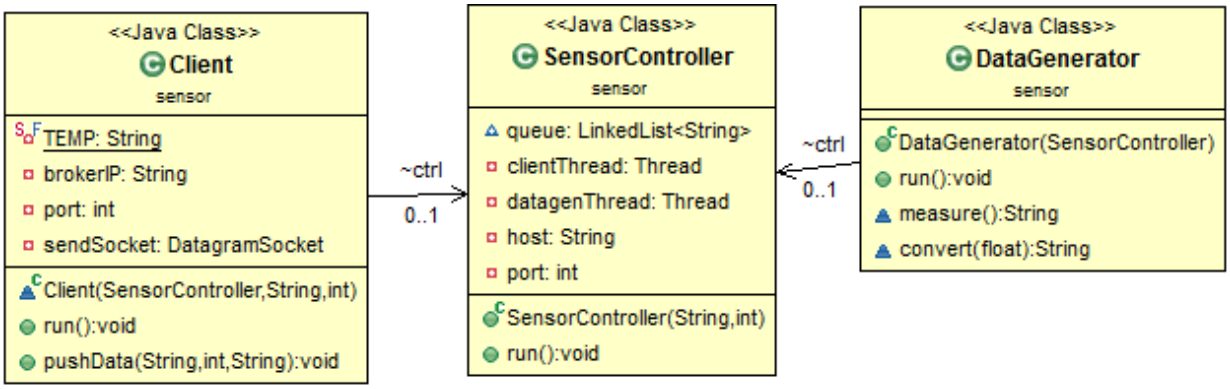
Temperaturserver

Temperatur serveren har en central controller - TempServerController der holder styr på rmi forbindelsen gennem sit remote object - Calculator og desuden starter en ny tråd med en eventlistener - TempListener. TempListener venter på UDP pakker og logger disse til en fil på disken - temperature.properties. Til dette anvendes en statisk hjælperklasse - Propertyhelper, der samtidig beskytter temperature.properties mod samtidige læse/skrive operationer.



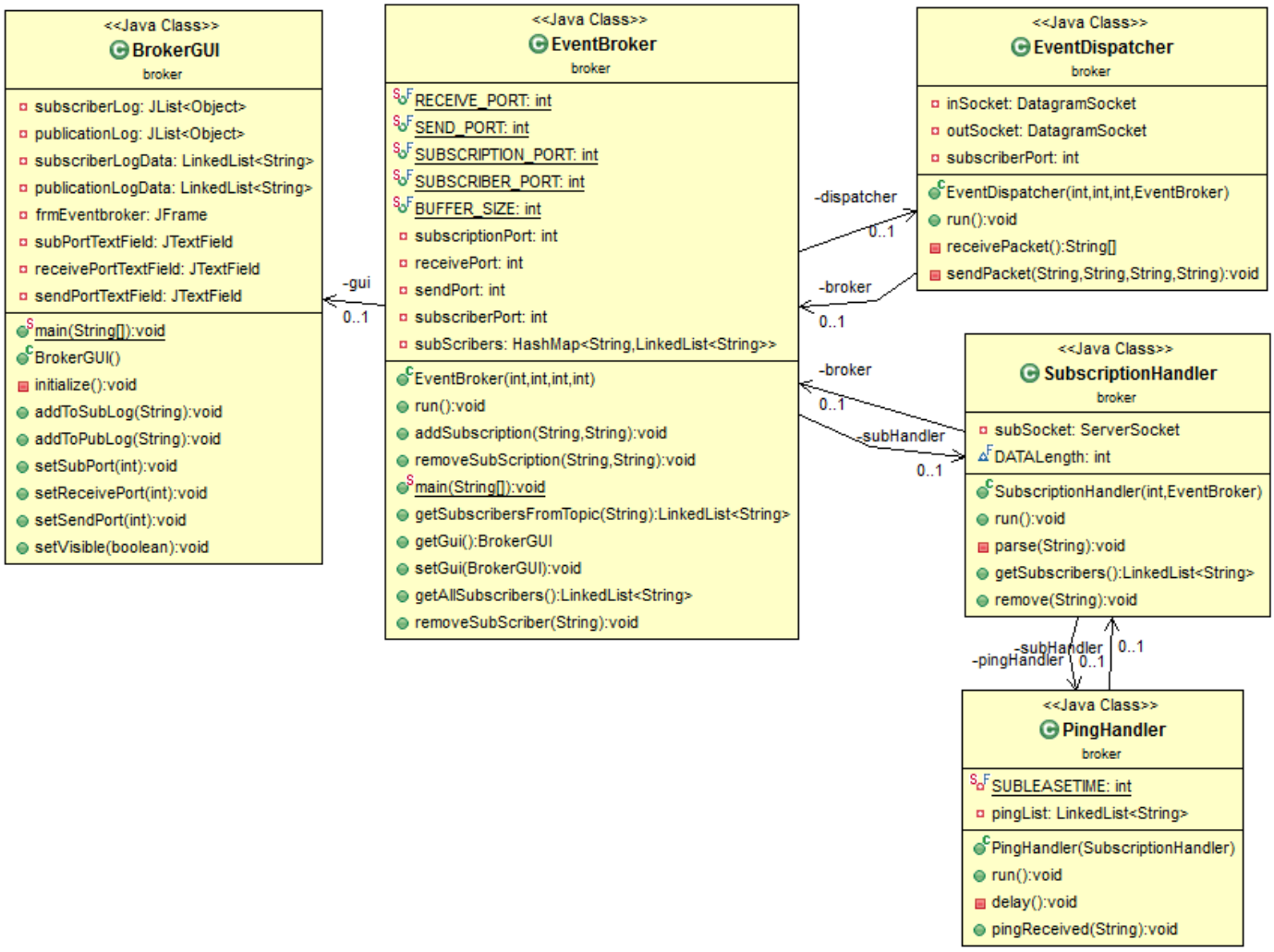
Sensorer

Sensorerene har en Controller - SensorController, der starter to nye tråde med en DataGenerator og en (UDP) Client. DataGenerator genererer tilfældige data i sin measure() metode og data skrives en volatile queue (der er implementeret som en linked list). På den måde kan Client læse fra køen uden at få korrupte data. Data sendes til Eventbrokeren via pushData metoden.



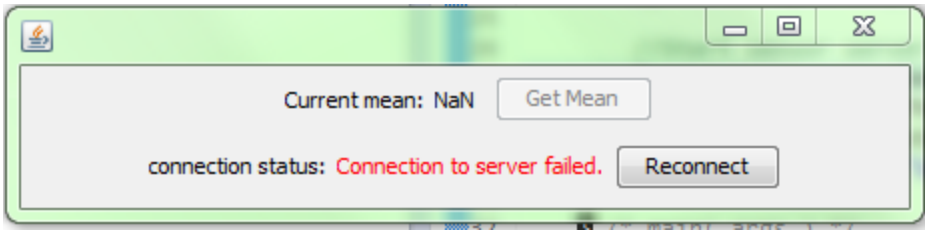
Event broker

Eventbrokeren har en Controller - EventBroker, der sørger for at starte BrokerGUI, EventDispatcher og subscriptionHandler i hver deres tråd. EventBrokeren holder listen over subscribers - der som nævnt er implementeret som et hashmap af linked lists. Key er topic for subscriberen og value den Linkede liste, der indeholder subscribers for det pågældende emne. Mappet er volatile for at undgå læse/skrive konflikter. EventBrokeren udstiller addSubscription(), removeSubscription(), getSubscribersFromTopic() og removeSubscriber(), som SubscriptionHandler anvender til at opdatere subScribers. getSubscribersFromTopic() returnerer den linkede liste af subscribers, der er knyttet til topic- key'en i hashmappet. removeSubscriber() fjerner alle subscriptions knyttet til en given ip-adresse.
EventDispatcheren venter på UDP-pakker. Når de ankommer, afkodes de i receievePacket() - emnet slås op i subScribers i EventBroker og alle på den linkede liste, der returneres, modtager en ny UDP pakke via metoden sendPacket().
SubscriptionHandler venter på TCP forbindelser fra clienter. Den starter en tråd med en pingHandler der holder styr på subscriber timeouts. Når en forbindelse oprettes, parses pakken i parse() og beskeden håndteres alt efter om det er en sub, unroll eller renew. Er der tale om en sub eller renew kaldes pingReceived() i PingHandler og den subscriber, der sendte pakken fjernes fra oprydnings listen (pingList i pingHandleren). Ved en sub besked tilføjes subscriberen til subScribers i EventBroker via addSubscription() metoden. Er der tale om en unroll fjernes subscriberen fra listen.
PingHandler kører sit loop 1 gang i timen. Alle subscribers tilføjes oprydningslisten - pingList hver time - er de ikke fjernet af SubscriptionHandleren inden der e rgået en time, fjernes alle subscribrens subscriptions fra subScribers i EventBroker.
Gui'en har udover et sæt tekstfelter, der viser relevante porte, to log vinduer, hhv for subscriptions og for publications. Når der kommer en ny subscription, en subscription annuleres eller en subscriber sender en renew besked, vises dette i venstre vindue. Tilsvarende vises en besked i højre vindue, når der modtages en event og den videresendes til subscribers.



Test

Systemet er en prototype og således kun konceptuelt testet. Alle komponenter kan dog startes i vilkårlig rækkefølge og bringe systemet i fungerende tilstand. Selvom om man starter rmi-klienten op først, er det muligt at forbinde til rmi serveren senere. Starter man Temperaturserveren op først, vil den kaste en exception, men forsøge at etablere forbindelse, indtil en broker kommer online. Starter man sensorer op først vil deres udp pakker blot gå tabt. Alle kombinationsmuligheder af partielle nedbrud undervejs er afprøvet. Går RMI serveren ned, er det nødvendigt at genstarte klienten, men ellers kan systemet godt fortsætte når delkomponenterne kommer op igen.



Rmi klienten er startet op uden en server. Starter man serveren op, kan man reconnecte.

Systemet fremstår således relativt robust - hvis man betragter det som en prototype, men man kan formentlig med begrænset opfindsomhed sagtens bringe systemet i knæ. Systemet er kun brugertestet, og unit tester man det systematisk, kan man formentlig afdække adskillige bugs.

Konklusion

Hele løsningen kunne have været implementeret som rmi kald (publish(event)), subscribe(topic)) til en central rmi-server - hvilket kunne have været implementeret betydeligt hurtigere. Jeg valgte at gå en lidt mere besværlig vej for at forsøge at implementere en letvægtsprotokol (UDP), der er teknologi (java) uafhængig. Processen var særdeles lærerig, og det at jeg var nødt til at definere (og omdefinere) egne protokoller for samspillet gav et godt indblik i de problemer, der let opstår når mange komponenter skal samarbejde.
Det ses i sammensætningen af løsningen at det er et forstudie, og en restrukturering af koden vil formentlig være gavnlig.
I en fremtidig løsning af samme problem, ville det være spændende at forsøge at distribuere middlewarelaget til netværkets knuder, således at man opnår stigende redundans når man tilføjer netværket flere sensorer. Implementeringen af samspillet mellem disse - eks. election protokoller og distribuerede hashtables ville dog have taget for lang tid i denne omgang.
Alt i alt opnåede jeg at sammensætte en fleksibel og velfungerende prototype, der ville kunne bruges i et virkeligt setup og med få modifikationer opnå en rimelig pålidelighed.

Brugervejledning

For at køre programmet skal man køre følgende mains som ligger i default package:

1. BrokerMain.java starter eventbrokeren op. (Kører en broker i forvejen fejler dette)
2. SensorMain.java starter 10 sensorer som hver sender tilfældige værdier mellem 14 og 24 med et mellemrum på 3 sekunder, indtil de lukkes.
3. TempServerMain.java starter temperaturserveren, der modtager events fra sensorerne og lagrer disse i en fil. Desuden sættes RMI serveren op.
4. ClientMain.java starter en grafisk brugergrænseflade, hvor man kan hente gennemsnit af alle målinger fra temperaturserveren. Hvis serveren ikke er startet kan man få den til at forsøge at reconnecte.

Værdier gemmes i en properties fil kaldt temperature.properties som oprettes i projektfolderen, som kan slettes/tømmes hvis man vil gøre et nyt forsøg.