UMEÅ UNIVERSITET
Institutionen för Datavetenskap
Laborationsrapport

10 juni 2016

Laboration 4

Objektorienterad programmeringsmetodik 5DV133

Obligatorisk uppgift 3 del 2

Grupp 15

Namn Susanne Kadelbach
Thomas Sarlin
Erik Dahlberg
Desireé Fredriksson

E-mail bio10skh@cs.umu.se
id15tsn@cs.umu.se
dv15dfn@cs.umu.se
c15edg@cs.umu.se

Handledare
Alexander Sutherland, Niklas Fries, Adam Dahlgren Lindström
Jonathan Westin, Erik Moström

Innehåll

1 Problemspecifikation

1.1 Problemsammanfattning

Uppgiften går ut på att implementera ett nätverk av trådlösa sensornoder som kommunicerar med varandra med hjälp av rumor-routingalgoritmen. I ett 50*50 noder stort nätverk, med 10 längdenheter mellan varje rad och kolumn, uppstår i varje nod en händelse med 0,01% chans vid varje tidssteg. Förfrågningar efter information om dessa händelser skickas var 400:e tidssteg ut från fyra, i början av körningen slumpmässigt utvalda, noder. Varje nod har en lista över sina grannar, de noder som förfrågningar och agenter får skickas vidare till, vilka innefattar noder inom deras räckvidd på 15 längdenheter. Förfrågningarna har 45 tidssteg på sig att hitta information om händelsen den söker innan den upphör att existera. Hittas informationen inom utsatt tid utgår den istället från nodens accepterade väntetid för svar på förfrågningen, vilket är 8*45 tidssteg från när den först skickades ut. Efter detta ger den upp och noden gör ett andra och sista försök med en ny förfrågning. För att öka chansen att finna informationen i tid skickas det med 50% chans ut agenter från varje händelse som detekteras. Dessa "meddelanden" har 50 steg på sig att sprida information, innehållande händelseID samt antal steg och vilket håll man ska gå gör att nå den, så att alla noder som fått besök av agenten kan dela med sig av denna information till förfrågningar som söker den. Agentens information uppdateras samtidigt genom att jämföra den mot nodens befintliga information och väljer ut de vägvisningarna med kortast väg till respektive händelse. Detta lagras för båda parter i tabeller. I varje tidssteg kan noden endast ta emot eller skicka ett meddelande, vilket innebär att noden har en kö med uppgifter som får vänta till nästkommande tidssteg. Om en förfrågan i tid finner den eftersökta händelsen och hinner tillbaka till noden den skapades i, ska information om händelsen skrivas ut: (plats, tid och id-nummer) följt av en radbrytning. Programmet körs i 10 000 tidssteg.

Arbetet genomförs i grupp av fyra och versionshantering sker via Git. I delmoment ett tas ett designförslag fram, som sedan ska implementeras i moment två.

1.2 Orginalspecifikation

Orginalspecifikationen för laborationen finns i sin helhet på kursens hemsida. [1]

[1] http://www8.cs.umu.se/kurser/5DV133/VT16/uppgifter/ou3/

2 Åtkomst och användarhandledning

2.1 Kompilering och körning

Under programmets gång har vi använt InteliJ version: 2016.1.2b med Java Development Kit (JDK) 1.8, för att skriva och kompilera programmet. Filerna till programmet är i format .java. I programmet finns det systemfiler och testfiler (testfilernas namn slutar med Test). Filerna till programmet finns i den medföljande ZIP-filen som heter ”OU3Java”. För att komma åt filerna högerklicka på Zip-filen och väljer sedan ”Extract Here”. Då skapas en mapp med namn ”OU3Java” där filerna finns. För att köra filerna kan man antingen använda java kompilator ex Intellij (rekommenderat JDK 1.7 eller högre), eller så kan man köra filerna via Linux Terminalen.

För att köra filerna via Linux terminal letas först mappen upp där filerna finns via terminalen. Efter att ha hittat filerna finns två sätt att köra filerna. Antingen så körs systemfilerna med testfilerna eller så körs enbart systemfilerna. Nedan finns beskrivning hur detta görs.

För att Kompilera och köra enbart system filer (allt görs via Linux terminalen):
1. Leta upp mappen där filerna ligger till programmet (mappen får inte innehålla Testfilerna)
2. Skriv i terminalen: javac *.java
- kommer att kompilera filerna som ska köras
3. Skriv i terminalen: java TestProgram
- Programmet kommer nu att köras
För att avbryta programkörning innan körningen är klar tryck ”Ctrl + c”.

För att Kompilera och köra både system och testfiler (allt görs via Linux terminalen):
1. Leta först upp mappen där alla filer ligger
2. Skriv i terminalen: javac*.java
3. Skriv i terminalen: -cp /usr/share/java/junit4.jar *.java
- kompilerar filerna som ska köras
4. Skriv i terminalen: java -cp :. /usr/share/java/junit4.jar org.junit.runner.JunitCore namnet på testfilen
- Programmet kommer nu att köras med testfilerna

I programmet finns det en klass TestProgram som kör hela programmet. Klassen TestProgram tar in 4 parametrar fieldSize, nodeDistance, nodeRange, endPoint och breakPoint. Vill man testa olika storlekar på programmet eller olika distanser är det fritt fram att prova på. Nedanför finns en beskrivning vad de olika parametrarna gör.

FieldSize – Storleken på kartan av noder, med storlek fieldSize gånger fieldSize. nodeDistance – distansen mellan noderna. NodeRange – distans radien för hur långt noder kan kommunicera med varandra. endPoint – hur många tidssteg programmet ska hålla på. breakPoint – vilket tidssteg som noderna ska skicka ut förfrågningar.

3 Systembeskrivning

3.1 Systemöversikt

Programmet utgör ett nätverk av noder som var och en representerar en händelse, ifall en händelse aktiveras på en nod så sprids informationen om denna händelse genom så kallad rumour-routing algoritmen, det vill säga att all information sprids via rykten. Dessa rykten sprids genom nätverket med hjälp av så kallade agentmeddelanden vilket är långlivade meddelanden vars uppgift är att gå runt igenom nätverket och uppdatera informationen hos noder. För att få respons från dessa händelser så skickas det ut förfrågningar som rör sig fram igenom nätverket och utnyttjar den information som delats av agenterna för att nå den händelse som förfrågades, ifall denna händelse hittats så ska denna förfrågan röra sig tillbaka igenom nätverket till ursprungsnoden och meddela nätverket händelsens position och vilken tidpunkt denna händelse skett. En javadoc-dokumentation har genererats från källkoden. `http://www8.cs.umu.se/~id15tsn/5DV133/OU4/index.html`

3.2 Klassbeskrivning

3.2.1 Network

Network ansvarar för att se till så att nätverket skapas och byggs upp som förväntat, storleken på fältet, distansen mellan noderna, radien av noderna för att sätta grannar samt hur programmet ska köras. Hur länge programmet ska pågå(i tidssteg) samt hur ofta förfrågningar ska skickas ut. Varje nod får sitt egna ID samt sina grannar tillagt.

Network är kärnan i hela programmet och har sin metod timeStep som får programmet att röra sig framåt. Denna metod låter varje nod aktiveras, aktiveringen av noden innebär att den har 0,01% chans det skapas en händelse på just den noden med ett avstånd på 0(steg) och 50% chans att ett agentmeddelande skickas ut från noden ifall en händelse har skapats. Efter att slumpen har fått avgöra om en händelse har skett så får noden en chans att antingen ta emot eller skicka ett meddelande. Efter aktiveringen av noder så kollar programmet om en brytpunkt är nådd, denna ställs in vid konstruktionen av nätverket. Ifall en brytpunkt nås så skickas det ut förfrågningar efter någon av de händelser som skett i nätverket. Varje gång timeStep körs kollar därför nätverket ifall några förfrågningar har återvänt med svar. Ifall ett meddelande har återvänt med information om en händelse så skrivs denna information ut. Ifall programmet har nått sitt slut i antalet tidssteg som ställdes in i början så skrivs information om hur många aktiviteter som skett, hur många förfrågningar som skickats/mottagits/gått förlorade osv.

3.2.2 Node

Node ansvarar för att hålla reda på de händelser som skett hos dem, hålla reda på information om andra händelser som dem fått från agenter. De ansvarar även för att se till så att meddelanden skickas vidare. En nod kan antingen läsa eller skicka ett meddelande per tidssteg. Den prioriterar att läsa/skicka förfrågningar före agenters framfart då agenter har längre levnadstid och lägre prioritet.

Ifall node får en förfrågan tillagt i sin kö så kollar den första tidssteget igenom sin egen tabell över händelser och kollar om den vet något om denna händelse. Ifall händelsens steg är 0 så innebär det att händelsen skett på denna nod och noden ger då information om denna händelse till förfrågan och refererar till förfrågan att den ska skickas tillbaka. Ifall händelsen har mer än 0 steg så får förfrågan nästa position i riktningen den bör gå för att hitta händelsen. Sista fallet, om noden inte har någon aning om vilket håll förfrågan ska röra sig åt så ger den sin lista på grannar och låter förfrågan välja fritt från denna lista.

Ifall node får ett agentmeddelande tillagt i sin kö, så läser den av dess värden i tabellen över händelser. ifall någon händelse har bättre distans så läggs dess steglängd samt nästa position till på nodens egen händelse. Ifall agenten har en händelse som inte noden har så läggs denna till i tabellen. Sedan delar noden sin lista till agenten så den kan göra samma sak med sin lista och på så vis har båda en uppdaterad lista. Nästa tidssteg så ger noden sin information till Agenten så den kan markera den som besökt samt få tag i dess grannar för att kunna slumpa en ny destination.

3.2.3 Time

Ansvarar för att hålla reda på antalet tidssteg som pågått i nätverket samt kunna säga till om en brytpunkt är nådd eller slutet på programmet är nått. Har funktioner för att öka tiden, returnera tiden samt sätta tiden till ett specifikt värde

3.2.4 NetworkStatus

Ansvarar för att hålla reda på hur många händelser som aktiveras, hur många agentmeddelanden som skickats ut. Även hålla reda på allt som har med förfrågningar och svar från förfrågningar att göra, ifall någon förfrågan har blivit till ex skickad, förlorad eller mottagen. Den gör detta med ett antal increment metoder. Slutligen har den en metod för att skriva ut all information om nätverket.

3.2.5 Response

En privat klass för nätverket. Den håller koll på när en specifik förfrågning har skickats, till vilken nod den skickats och vilken händelse den har frågat efter. Detta för att nätverket ska kunna veta när meddelanden har gått runt för länge utan att hitta svar och behöver skickas igen och då till vilken nod och vilken händelse den ska söka efter. Slutligen för att veta när den bör sluta vänta på svar från en specifik förfrågan.

3.2.6 Position

Ansvarar för att representera specifika positioner för noder i nätverket med hjälp utav ett x och y värde.

3.2.7 Action

Representerar en händelse som skett på en specifik nod, innehåller vilken position och tidpunkt händelsen skett. Har även en stegräknare och en positionspekare för att Agenter/Noder ska kunna referera till denna händelse, hur långt bort den är och vilken nästa position är för att röra sig i riktningen mot den.

3.2.8 Message

Abstrakt klass som representerar ett meddelande, implementeras av AgentMessage och RequestMessage.

3.2.9 AgentMessage

Representerar ett agentmeddelande vars uppgift är att sprida information om händelser som sker i nätverket. Dem har 50% chans att skapas då en händelse uppstår. Ifall en agent skickas till en nod så läser noden av informationen i agenten tabell och kopierar de informationen om de händelser som innehåller bättre värde alternativt inte fanns på noden sen tidigare. Efter detta returnerar noden sin tabell av händelser till agenten. Agenten uppdaterar då sin tabell med de bästa värdena från noden samt lägger till information om händelser som informationsteknologi agenten sett sen tidigare.

Agenten markerar noden den var på som föregående position samt sparar att den är besökt. Då noden skickar vidare agenten uppdaterar agenten sina händelser i tabellen med senaste positionen samt ökar distans värdet.

När agentmeddelandet rört sig 50 steg så slutar den sprida information.

3.2.10 RequestMessage

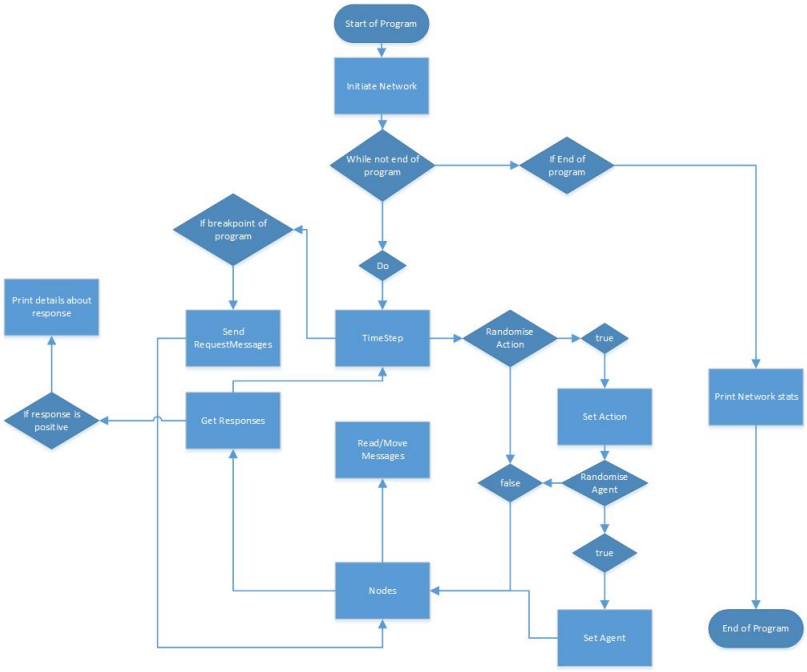
Klassen representerar en förfrågan som får när den skapas på en nod information om vilken tid den startar, vilken händelse den söker efter och ett eget id.

Förfrågan rör sig från ursprungsnoden i ett slumpmönster och letar efter information, på de noder den besöker, var händelsen har inträffat. En förfråga har upp till 45 tidssteg för att hitta information om händelsen den söker. Ifall den går 45 steg utan att hitta information om händelsen kommer förfrågan förkastas. Ifall den hittar information om händelsen den letar efter kommer förfrågan följa noderna till händelsen. När den har hittat noden där händelsen har inträffat så sparar den ner all information den behöver och rör sig tillbaka samma väg den har kommit till noden förfrågan utgick ifrån.

3.3 Klassdiagram

Figur 1: Klassdiagram över ingående klasser till lösningen

3.4 Flowchart



Figur 2: Flowchart över programmets exekveringsflöde

4 Lösningens tolkningar och begränsningar

4.1 Designval och tolkningar

Nodens "toDoList", innehållande meddelanden som ska hanteras i ett senare tidssteg, har fått en prioritetsordning. En "request" har förtur mot "agent", med anledningen att dessa har en livslängd begränsad av tidssteg. Agenten, däremot, har ett fast antal steg innan den upphör att existera. Designvalet har gjorts för att öka möjligheten att ett request hinner utföra sin uppgift innan tiden rinner ut.

4.2 Begränsningar

Noderna och agenternas tabeller innehåller händelser, vilket gör att dessa blir relativt minneskrävande. Ett alternativ är att använda sig av dess position som referens, men utifrån vår design är det enklare att skicka information med hjälp av händelser istället för strängar.

5 Testkörningar

För att få så många och så bra tester som möjligt, valde gruppen att vi skulle skriva tester under tre steg i processen. Innan vi hade skrivit klasserna, undertiden vi skrev klasserna och efter att klasserna var klara. Vi valde denna metod på grund av att ingen av oss hade innan skrivit tester för klasser som inte var gjorda. Här valde vi även att ingen skulle skriva tester för de klasserna som vi själva skulle skriva. För att göra detta delade vi upp oss i två mindre grupper som fick hälften av klasserna var. Detta gick bra för vissa av klasserna med andra mer komplexa klasser blev svårare. Till exempel var det svårt att skriva för klassen Message eftersom vi inte riktigt hade bestämt hur meddelandena skulle röra sig i kartan. Vi ville även skriva tester medan vi själva skrev klasserna så att vi kunde försäkra oss om att metoder fungerade som förväntat. Slutligen så skrev vi tester när klasserna var klara om vi hade missat något eller om vi tyckte det behövdes fler.

5.1 Network

Det första testet kolla att inga fel kastas då ett nätverk skapas med korrekta argument, följt av tester som försöker skapa nätverk med inkorrekta argument. Det testas även att nätverket verkligen skapar noder och kopplar tillsamman dessa korrekt genom att utnyttja Pythagoras sats, även vid olika sorters radie på nodernas "mottagningsområde" ska grannarna läggas till korrekt, både i antal samt så att rätt noder är sammankopplade. Slutligen testas att nätverket utnyttjar network-stats för utskrift om vad som hänt under programmets lopp samt att programmet avslutas som tänkt när slutpunkt nåtts.

Resultat:
Nätverket lyckades kasta fel som förväntats vid fel argument samt att noderna har kopplats ihop med de grannar som är innaför deras radie samt har fått rätt grannar tillagda i mönster, ifall grannar som är utanför nätverket försöks läggas till så händer inget och programmet rullar vidare. Utskriften vid respons från förfrågan testas i det samlade testProgrammet för hela nätverket som heter "TestProgram".

5.2 Time

Testar så att times konstruktor kastar fel som förväntat ifall time initieras felaktigt. Testar ifall programmet är funktionabelt att kunna öka antalet tidssteg samt returnera korrekt tid med getters efter ett antal ökningar. Även isEnd samt isBreakPoint testas så dem fungerar som förväntat.

Resultat:
Alla värden ökas och returneras som förväntat, konstruktören kastar fel ifall dem upptäcks.

5.3 Node

Testar att Node kan skapas, ta emot och skicka information om sin id och position. Kollar även så att equal metoden fungerar och att den kan ta emot en-

staka händelser. Testar att noden endast tar emot ett meddelande eller skickar ett meddelande när TakeNextMessage blir kallad. Att den kan hantera Agent-meddelanden med eller utan värden i sin tabell. Kontrollerar även att noden uppdaterar sin tabell om en agent skulle ha bättre information om en händelse eller om den skulle ha händelser som inte noden har. Slutligen så testas om ett händelse-meddelande frågar efter en händelse som noden har information om så skickar noden vägen till händelsen eller om den inträffade på noden så skickar den händelsen.

Resultat:
Testarna visade att noden fungerar som den ska och att den enbart kan ta emot ett meddelande eller skicka ett meddelande i taget. Kan även säkerställa att noden fungerar korrekt med agent-meddelanden och händelse-meddelanden.

5.4 AgentMessage

Testar ifall agent-meddelanden har möjlighet att skapas, röra sig samt sprida information om händelser. Det testas även att ifall en nod har en bättre information om en händelse ska dessa sparas i agentmeddelandet och om noden har information om händelser som inte agenten vetat om så sparas även denna information i agentmeddelandet. Det testas även med det omvända påståendet, att agenten skulle vara den med bättre och mer uppdaterad information då ska den sprida sin information till noden utan att påverka nodens andra aktuella värden. Slutligen testas så att efter 50 steg ska agenten sluta läggas till i köer på noder.

Resultat:
Alla test visade på att agentmeddelande fungerade som förväntat.

5.5 RequestMessage

Testar huruvida förfrågan har möjlighet att skapas, röra sig samt ta emot responser från händelser och returnera dessa till ursprungsnoden. Sista testet är för att se så att förfrågan slutar skickas vidare ifall den gått 45 steg utan att hitta information om en händelse.

Resultat:
Alla test visade på att förfrågan fungerade som förväntat.

5.6 Position

Testar att nya positioner kan skapas och att den kan skicka sin X och Y koordinater. Kontrollerar även så att equal funktion fungerar.

Resultat:
Testerna visade att position fungerar som förväntat.

5.7 TestProgram

Test programmet kör enligt beskrivningen från uppgiften:
Tidssteg: 10000
Förfrågningar: var 400 steg
Fältstorlek: 50x50 noder
Chans att händelse skapas: 0.01%
Chans att agent skapas på händelse: 50%
En bild på utskriften efter programmkörning finns i Bilaga 1.

Test:	1	2	3	4	5	6	7	8	9	10	Medel
Händelser skickade	2504	2456	2492	2453	2509	2499	2446	2497	2599	2528	2498.3
Agenter skickade	1284	1265	1208	1288	1294	1268	1196	1252	1264	1252	1257.1
Förfrågan skickade	96	96	96	96	96	96	96	96	96	96	96
Förfrågan återsskickae	29	34	29	25	24	32	35	21	38	25	29.2
Misslyckade förfrågan	20	24	24	17	18	19	25	16	27	18	20.8
Aktiva förfrågan vid slutet av program met	0	0	0	0	0	0	0	0	0	0	0
Antalet responser från händelser	75	71	72	79	78	75	70	80	67	77	74.4
Meddeltid för respons	149	135	149	129	113	122	125	124	144	127	131.7

Figur 3:Statistik från 10 olika testkörningar samt medelvärdena från dessa testkörningar.

Resultat:
Programmet verkar fungera som förväntat. Vid närmare undersökning av resultaten kan man observera att väldigt många förfrågningar återvänder. Vanligtvis ligger nivån för återvända förfrågningar kring 20 till 40 %. Vårt resultat ligger kring 70 %. Detta kan förklaras genom att vi använder en metod som kallas för isVisited. Metoden ska hjälpa att sprida information i nätverket genom att förhindrar att agenter inte besöker noder som de redan har varit på om det finns obesökta alternativ att gå på.

5.8 Arbetsfördelning

Inledande designprocess genomfördes i grupp, med stort fokus på att samtliga gruppmedlemmar tolkade uppgiften likadant. Alla deltog aktivt och bidrog till att utforma projektet. Under arbetets gång lades stor vikt på att frågetecken och förändringar diskuterades noga, med anledning av att alla gruppmedlemmar jobbade åt samma håll. Mycket av kodskrivningen producerades av denna anledning runt samma bord, för att enklare hjälpa varandra och säkerställa att ändringar som görs anpassas i resterande delar av programmet.

Tabell 1: Arbetsfördelningen uppdelad i programklasserna och deras tester. Beskriver ungefärligt, men till följd av mycket samarbete inte uteslutande, vem som har gjort vad.

	Test	Klass
Network+interna klasser	Thomas	Thomas
Node	Susanne, Desireé, Erik	Erik
Message	Susanne	Susanne
Request	Thomas Susanne,	Desireé
Position	Susanne, Desireé	Erik
Agent	Thomas	Susanne, Desireé
Time	Thomas	Susanne
Action	Susanne, Desireé	Erik
TestProgram		Thomas

Tabell 2: Arbetsfördelning under rapportskrivning

Rapport	Namn
Inledning	Desireé
Programbeskrivning	Thomas
Klassbeskrivning	Thomas
Designval och tolkningar	Desireé
Begränsningar	Desireé
Programkompilering och körning	Erik
Testbeskrivning	Erik
Arbetsfördelning	Desireé
Överföring till Latex, korrekturläsning samt infogandet av länkar och bifogade filer	Susanne
Klassdiagram, flödesschema	Thomas

6 Bilaga 1

```
Response sent at: 9200 Response received at: 9378 Response time: 178 resent:false
Action ID: 2452:7915 Position:(49,2) Action created at: 7915

Response sent at: 9200 Response received at: 9388 Response time: 188 resent:false
Action ID: 513:3035 Position:(10,13) Action created at: 3035

Response sent at: 9600 Response received at: 9684 Response time: 84 resent:false
Action ID: 1313:5040 Position:(26,13) Action created at: 5040

Response sent at: 9600 Response received at: 9696 Response time: 96 resent:false
Action ID: 1374:2103 Position:(27,24) Action created at: 2103

Response sent at: 9600 Response received at: 9728 Response time: 128 resent:false
Action ID: 789:2587 Position:(15,39) Action created at: 2587
Disconnected from the target VM, address: '127.0.0.1:63195', transport: 'socket'


--------------NETWORK SUMMARY---------------
Actions sent: 2446
Agents sent: 1196
Requests sent: 96
Requests resent: 35
Requests failed: 25
Requests active at end of program: 0
Responses received: 70
Average Response time: 125
----------------------------------------
End of program
```

Figur 4: Utskrift av testprogrammet