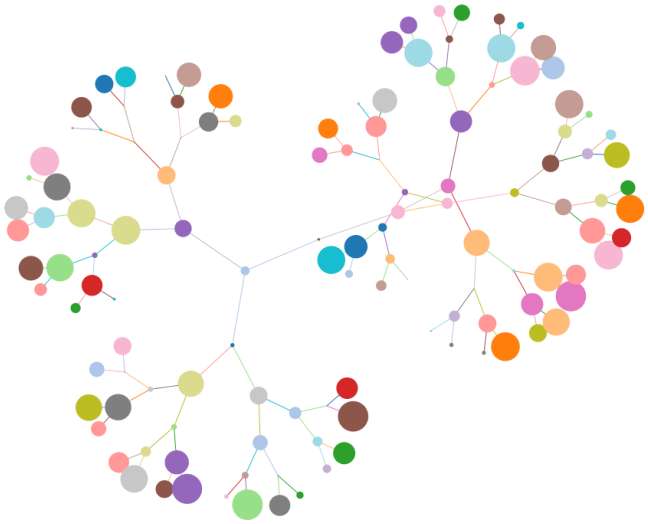
Slutrapport OU3 Grupp 16



Av

Dardan Dauti **(id16ddi)**, Jonas Sjödin **(id16jsn)**,
Pontus Eriksson **(dv16pen)**, Wilhelm Ackermann **(id16wan)**

Datum

2017-05-17

Kurs

Objektorienterad Programmeringsmetodik 5DV133

Kursansvarig

Anders Broberg

Handledare

Adam Dahlgren, Daniel Harr, Didrik Lindqvist,
Jakob Vesterlind, Sebastian Sandberg

Innehållsförteckning

Inledning

Vi har fått i uppgift att skapa ett program som har flera delar. Programmet ska skapa ett fält med noder där en samling event ska hända. Vid en händelse av event ska det sedan finnas en chans för fältet att skapa en Agent. Dessa agenter sprider händelser till vissa noder och instruktioner till andra noder om var dessa händelser befinner sig. Programmet skapar sedan förfrågningar som vi kallar för “Request”. Dessa söker efter dessa spår för att försöka hitta händelsen den söker efter.

I denna rapport beskriver vi hur vi har skapat programmet, hur delarna i programmet fungerar, testningsmetoder, hur vi har utgått ifrån instruktionerna angivna i specifikationen, samt hur vi har kompilerat- och kört programmet.

Rumour Routing

Detta nätverkssystem fungerar genom att ett fast antal noder är placerade i en graf. Slumpmässigt kan alla dessa noder skapa händelser på sin position. Vid jämna mellanrum skickar några av noderna ut förfrågningar efter en slumpmässig händelse. Dessa förfrågningar går då ett givet antal steg (varje steg är en förflyttning från en nod till en annan) på noderna för att hitta varsin satt händelse. Det finns även agenter i detta nodsystem, som har som uppgift att gå omkring mellan noderna och sätta ihop deras händelselistor men ökar då avståndet på händelserna med ett. Om en förfrågning hittar en händelse med ett avstånd som inte är noll följer den ett spår som lagts ut av agenten när de satt ihop händelselistorna. På så sätt kan den följa vägen till en händelse.

Efter ett tag när flera agenter gått omkring finns det väldigt många spår efter events i hela nodsystemet vilket kommer att medföra att väldigt många förfrågningar tar sig fram till sin satta händelse.

För mer info, se referenser [3].

Åtkomst och Användarhandledning

Kompilering

För att kompilera programmet med hjälp av filerna så använder vi dessa kommandon. Användaren förväntas bruka en av institutionens datorer på Umeå Universitet.

Egen implementation av programmet

```
javac <filnamn>.java
```

JUnit testerna

```
javac -cp /usr/share/java/junit4.jar <Fil som ska testas>.java <testfil>.java
```

Given implementation

```
javac main.java
```

Körning

Efter kompilering skall programmen köras. Körningen sker genom att tillkalla den fil vilket innehåller en main vilket i detta fall är filen main samtidigt som denna ligger i samma mapp som dess tillhörande class filer.

Egen implementation av programmet

```
java <filnamn>
```

JUnit testerna

```
java -cp .:/usr/share/java/junit4.jar org.junit.runner.JUnitCore <testfil>
```

Given implementation

```
java main
```

In- och Utdata

Indata för programmet är konstanter inuti Config.java. Utdata är ett fält med noder.

Javadoc

Vi har laddat upp vår dokumentation på vår git. Länk: https://goo.gl/T3J9rA

Tolkning av Specifikationen

- Livslängd
  - Som uppfattat från specifikationen så har vi satt livslängden på agenter till 45 och livslängden på förfrågningarna till 50. Livslängden här representerar antalet steg som man får ta innan man ska räkna det som att en agent eller förfrågan ska ge upp.
- Hur de går omkring
  - Agent försöker att inte gå till de noder den redan har besökt. Request går helt slumpmässigt fram. Response följer samma väg som Request tog.
- Vem som hanterar meddelanden
  - Noderna skickar vidare agenter, requests och responses. De hanteras som så att noderna kommer skicka ut requests som går runt och söker efter meddelanden. Sedan skickar noderna vidare responses kommer att lämna tillbaka meddelandet till eventet som i sin tur skriver ut det.

Systembeskrivning

Vår implementation av Rumor Routing

Programmet använder sig av Rumor Routing, alltså att en agent skapar händelser och spår till händelsen runt en karta av noder, som en förfrågning sedan ska navigera till.[3]

Agenter rör sig mellan noder och har en viss livslängd. Den kommer med viss chans varenda gång den går lämna ifrån sig en händelse och hädanefter i alla de noder som den besöker så kommer denna händelse då refereras till. Informationen som den lämnar ifrån sig visar vilken nästa koordinat är för att ta sig till händelsen, samt hur långt ifrån den är. Det betyder att vid händelsen är distansen noll. När en agent passerar igenom nod som redan har information om en händelse kommer agenten i framtiden sätta ihop de olika data-samlingarna och fortsätta sprida de båda. Agenten har också förmågan att förkorta och uppdatera gamla spår ifall den har hittat en bättre rutt till händelsen, såvida den går in var den redan har varit.

Request är ett meddelande som en specifik händelse. Likt Agent har den en livslängd, men ifall den hittar ett spår till den händelsen den söker efter så får den fritt gå till den utan att ta hänsyn till dess liv. När en nod försöker skicka ett meddelande kollar noden först om meddelandet är på ett spår. Om så är fallet kommer meddelandet försökas skickas till nästkommande nod i spåret. Är meddelandet inte på något givet spår skickas det i en slumpad riktning i förhoppning om att på så sätt hitta ett spår. Vid varje steg tagit undan sparas den väg för vilket Requesten har tagit. Ifall en Request ej hittar ett event eller tar för lång tid på sig kommer nätverket att efterfråga samma event igen, dock endast en gång till, med förhoppning om att fler spår ska ha spridits till källan.

Response är en andra del av Request. Om Request har hittat en väg till en händelse kommer denna omvandlas till ett Response och börja traversera nätverket tillbaka till ursprungs noden. Den väg vilket Request sparade undan står som grund för vägvalet och kommer blint följas hela vägen.

Begränsningar i implementation

Eftersom vår lösning utgår ifrån att vi har alla konstanter i en fil som heter Config.java. Om dessa ska ändras måste programmet kompileras om vid varenda körning. Simuleringens prestanda försämras avsevärt desto fler noder. Om det ändras sådant att det är en högre chans att ett event eller en agent skapas, kommer prestandan försämras avsevärt.

Klassbeskrivning

Position

Denna klass är en position med x- och y-koordinater i vårt nodsystem. Positionens x - och y-koordinat sätts i konstruktorn och kan sedan hämtas med de publika metoderna **getX()** och **getY()**. Positionen innehåller även egenskrivna **equals()** och **hashCode()**.

Time

Denna klass är en tidshanteringsklass med endast publika metoder. När en **Time** instansieras sätts tiden till 0. Tiden kan sedan ökas med 1 genom att köra **increment()** och värdet kan hämtas med metoden **getTime()**.

Config

Denna klass är en konstant klass som innehåller konstanter. Dessa konstanter kan hämtas utan att **Config** instansieras eftersom att klassen är konstant. Konstanterna som finns i den här klassen är sannolikheten för att ett **Event** och en **Agent** ska skapas och de är skrivna i bråkform. Därför måste användaren hämta både täljaren och nämnaren när sannolikhet för någonting ska beräknas. Även maxlivet för en **Agent** och en **Request** kan hämtas här samt storleken på nodsystemet. Här är konstanter som maxtiden för hur länge programmet ska köras satta samt hur många noder som ska skicka ut förfrågningar. Avståndet mellan noder är också specificerat här samt hur lång räckvidden nod har, d.vs. hur långt en nod kan max ha till andra noder för att få kalla dem sina grannar. Alla dessa konstanter kan hämtas genom att köra olika publika **get()** metoder.

Randomizer

Denna klass är en slumpklass som kan genom dess två publika booleanska metoder **eventTrigger()** och **agentTrigger()** ta fram en boolean om ett event eller en agent ska skapas. Detta gör den genom att hämta konstanterna om sannolikhet för att en **Agent** och ett **Event** ska skapas i **Config**-klassen.

Node

Denna klass är en nod i vårt nodsystem. Den har olika listor och kartor med hashvärden för nodens grannar och **Events** som finns i noden. Om en nod skickar en **Request** och inte får ett svar inom 8 * livslängden för en **Request** skickar den ut en likadan **Request** en gång till. Detta hashas också och sparas i en hashmap. Klassen har också en kö som hanterar inkommande meddelanden som **Agent**, **Request** och **Response**. Denna kö plockar metoden **sendMessage()** meddelanden ifrån när den skickar dem. Sedan har klassen ett specifikt ID som sätts när klassen instansieras och det är detta som talar om för andra noder och meddelanden vilken node de pratar med. Sist finns en tidsvariabel vid namn **busy** som sätts till nuvarande tiden om ett meddelande mottagits eller skickats då en nod endast kan behandla en sak per tidssteg.

I konstruktorn sätts nodens **position** och **id**. Metoden **getMessage()** hämtar meddelandena för att **Network**-klassen ska kunna rita ut nodsystemet. Sedan finns metoden **addNotRecieved()** som lägger till en Request i **notRecieved-**kön och hämtas om meddelandet inte kommit fram på 8 * livslängden. Sedan finns flera metoder som **addNeighbour()**, **addEvent()**, **getId()**, **getPosition()**, **getEvents()** och **addMessages()** som gör precis det som namnen antyder. Metoden **setBusy()** sätter **busy**-variabeln till tiden den tar som parameter och **nodeIsBusy()** kollar om tiden som den tar som parameter är lika med **busy**-variabeln. Är den det är noden upptagen.

Den sista metoden **attemptSendMessage()** är egentligen fyra metoder. Först kollar metoden om det finns ett meddelande som ska skickas igen i **notRecieved**-HashMapen. Om det finns det läggs det meddelandet till i meddelandekön och tas bort från HashMapen. Sedan kollar metoden om det första meddelandet i meddelandekön är dött och tar isåfall bort det första meddelandet tills kön är tom eller det första meddelandet inte är dött. Efter detta körs en koll om meddelandelistan är tom eller om noden är upptagen. Är den inte det försöker metoden skicka ett meddelande till en annan nod genom att köra de tre metoderna **sendAgent()**, **sendRequest()**, **sendResponse()**.

sendAgent() kollar först om noden har några grannar som agenten inte skickats till och lägger isåfall dess grannar i en lista. Seda körs en koll om de obesökta grannoderlistan är tom om. Om den inte är det slumpas en av de noderna fram och metoden försöker skicka agenten vidare till den noden så länge som den inte är upptagen. Om agenten besökt alla grannoder så försöker metoden skicka den vidare till en slumpad grannod och lyckas så länge den noden inte är upptagen.

Metoden **sendRequest()** kollar först noden innehåller rätt event som requesten letar efter om den gör det körs en koll om detta är det första eventet av detta slag som skapades, d.v.s om eventets distance är lika med 0. Om det är det så skapas ett **Response** som ska gå samma väg som requesten gick fast bakvänt. Sedan tas requesten bort och metoden avbryts. Om det inte är det första eventet av denna typ så sätts requesten till att vara på ett spår till rätt event. Då körs en koll om nästa nod som den ska gå till är upptagen. Är den inte den så skickas meddelandet dit. Om denna node inte innehåller det **Event** som letas efter så försöker noden skickas det vidare till en slumpmässig granne och lyckas om den inte är upptagen.

Metoden **sendResponse()** kollar först responsen kan ta ett steg till. Om den kan och nästa nod inte är upptagen så skickas den dit. Om responsen inte kan ta ett steg till är denna nod dess slutdestination. Då körs en koll om denna responses ID finns i listan med inte tillbakakomna noder. Om den gör det så tas den bort då requesten har fått ett svar. Sedan tas det meddelandet bort från messages listan.

Sist finns en metod hashCode som skriver över standard-hashCode och returnerar nodens unika id.

Event

Denna klass kan skapas av slump varje tidssteg i varje nod. När ett event skapas finns också en chans att en **Agent** skapas. Denna klass har två konstruktorer där den ena kopierar en annan **Agent** och den andra skapar en ny **Agent** och tar parametrar för detta. Klassen har sedan metoderna **getPrevPosition()**, **setPrevPosition()**, **incrementDistance()**, **getEventID()** och **getDistance()** som gör det som namnen antyder. Sedan finns metoderna isFinalEvent som kollar om distance är 0 och om den är det returnerar den true och metoden getMessage som returnerar en String som består av positionen, tiden och dess eventID.

Message

Denna klass är superklass till **Request**, **Response** och **Network**. Den har en livslängd som sätts i konstruktorn och metoderna **age()** som får life-variabeln som börjar på maxLiver att minska med 1 samt metoden **isAlive()** som kollar om livet är större än 0 och returnerar true om den är det och annars false.

Request

Denna metod är förfrågningsdelen av en Query. Den sätter olika värden genom konstruktorn och lägger till den nuvarande nodPositionen som besökt. Sedan finns metoderna **getEventID()**, **getRequestID()**, **onPath()**, **isOnPath()** och **getPathTaken()** som gör det namnen antyder. Metoden **move()** lägger till nodID:t i sin path och åldrar meddelandet med 1. Sedan finns metoden **isAlive()** som skriver över messages metod med samma namn. Den returnerar true om messages metod med samma namn returnerar true eller om den är på en väg till ett finalEvent.

Response

Denna klass är svarsdelen av en Query. Den tar olika värden som parametrar och sätter de till lokala värden. Den tar bland annat hemvägs-stacken som parameter och det är den som den följer för att gå hem. Den har metoderna **getResponseID(), move(), getNextStep(), hasNextStep()** och **getEventMessage()**. Metoden isAlive finns också och den skriver över messages isAlive och returnerar endast true.

Network

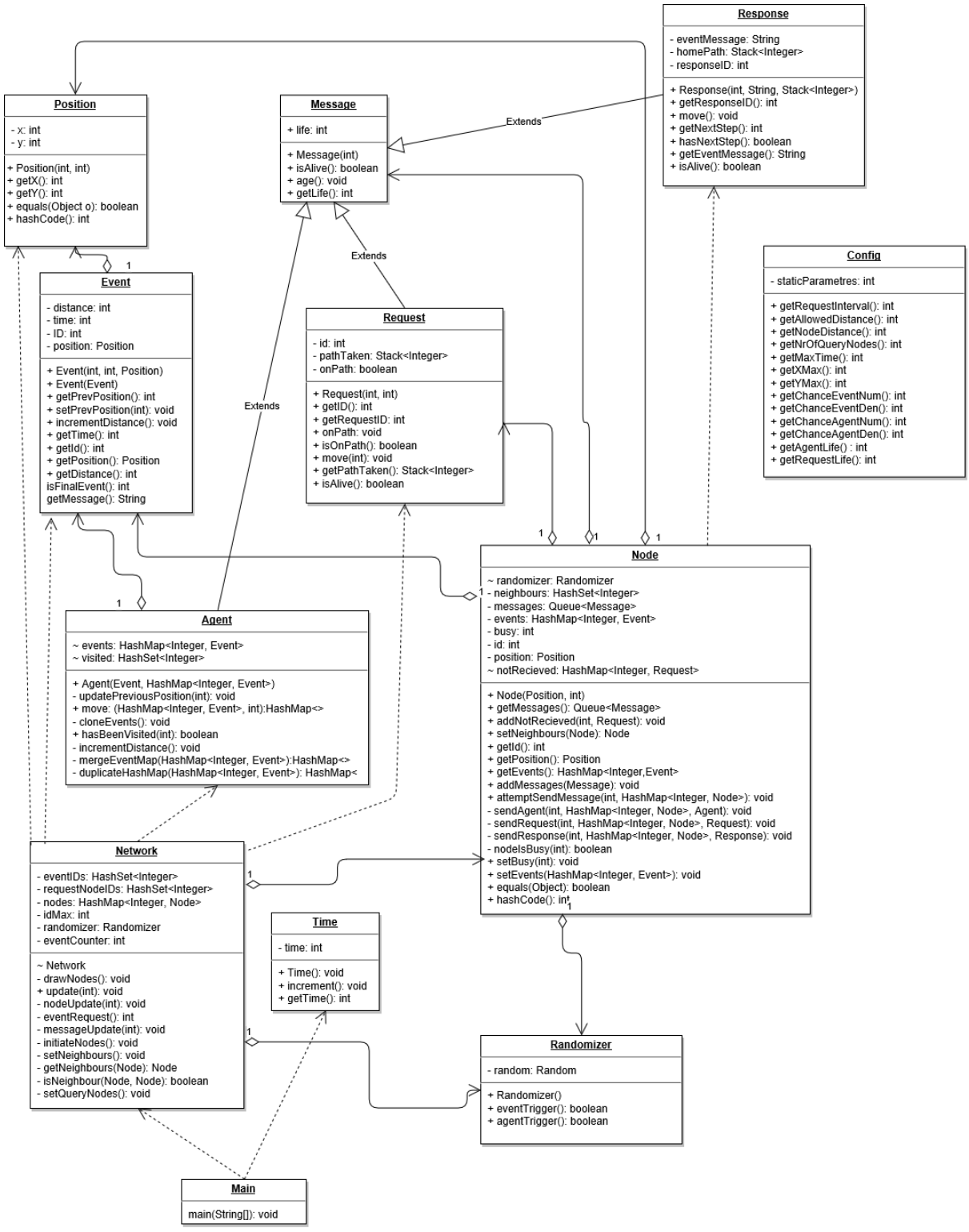
Denna klass styr själva nodnätverket. Konstruktorn kör först metoden **initiateNodes()** som skapar ett rutnät av noder och sätter ett ID på varje nod. Sedan körs metoden **setNeighbours()** som sätter alla grannar till noderna genom att loopa igenom alla noderna och kör **getNeighbours()** på varje nod som sätter alla grannarna till en nod. Sist körs **setQueryNodes()**. Den metoden sätter ett antal unika queryNodes som sedan har som ansvar att skicka ut responses. Sedan har konstruktorn körs färdigt. Sedan finns metoden **update()** som tar tiden som parameter och kör metoden **nodeUpdate()**. Den metoden kör **eventTrigger()** för alla noder och den returnerar true så skapar den ett **Event** på den noden. Om ett event har skapats så körs **agentTrigger()** och om den returnerar true så skapas en

Agent. Sedan kör metoden **messageUpdate()**. Den metoden kollar om det är dags att skapa requests. Om det är det så skapar den en request i varje queryNode. Sedan så kör den alla noders **attemptSendMessage()**. Metoden **drawNodes()** är bortkommenterad men den är en grafisk visning av hur nodsystemed ser ut som kan köras om användaren vill. För att få snabbare utritning sätt sleep till ett mindre värde.

Main

Denna klass till största delen är ett exempel på hur en större variant av programmet skulle kunna se ut, och ansvarar då också för att ha en ytlig instruktion hur programmet körs. Vår implementation körs i en for-loop som går 10000 tidssteg och för varje omgång körs **update()**-metoden i **Network** och tiden ökas med 1.

Klassdiagram



https://goo.gl/uvVqui

Diskussion

Det vi märkte ganska tidigt var att vår design som vi hade gjort innan inte skulle fungera till hundra procent, eller att det skulle bli onödigt komplicerat, därför fick vi skriva om hela programmet och tänka om hur vi skulle lösa uppgiften. Eftersom java använder sig av code by reference så länge som det inte är de mest grundläggande datatyper så kunde vi hasha allt i tabeller och spara ID. Det gjorde det mycket enklare för Request att söka efter rätt händelse.

Hade vi gjort om uppgiften hade vi nog försökt få ännu mer klargörelse för hur programmet ska se ut. Mer utförliga tester i designfasen samt att ändra hur vi byggde upp arbetsprocessen på. Vi i gruppen gjorde delar av programmen individuellt och testade andra delar av programmet efteråt. Det hade gått att göra testen först, och sedan skapa programmet utifrån det istället. Genom att tidigt implementera designen hade vi kanske slösat mindre tid med debugging.

Testning

För att garantera hög kodkvalitet måste vårt system prövas. Dessa tester skall utföras under arbetsprocessen och vi ska följa de principer för kodkvalitet given av *Anders Broberg*[2] under sina föreläsningar. Bland annat har vi fokuserat på kvalité av kod, inte kvantitet och varit aktiva med gemensamt ansvar för projektet.

För att underlätta för alla under testning har vi gjort vår kod så lättläst som möjligt, samt sett till att kommentera alla metoder och klasser väl.

Program-testningen under utvecklingen har sett ut sådant att en person har skrivit en klass eller en metod, och en annan person har i sin tur testat det, på så vis såg vi till att kodaren inte har missat buggar i systemet på grund av tunnelsyn. Med detta arbetssätt tillsammans med principen att ha kommenterat sin kod efter sig såg vi att det blir enkelt för andra i gruppen att dyka in i andras kod och vi slapp ödsla tid när klasserna skulle implementeras samman.

Efter utvecklingen av programmet gör vi ett större test där vi kollar igenom alla metoder i programmet med hjälp av JUnit och skapar sedan en egen implementation som vi använder till test. Detta test kommer att testa hela programmet och ser till att vi inte missat detaljer i enhets-testerna. Vi gör detta eftersom även fast man testar en metod så kan ett enhetstest fortfarande gå igenom ifall det endast funkar vid just det specifika tillfället.

En testkörning av vårt färdiga program ser ut som så att vi sätter alla värden som behövs för att köra vårt program. För att konfigurera bland annat storlek på fält, antal liv som agenter och requests har samt antalet steg som en som man kan gå innan agenten eller requesten dör så sätts alla variabler till värden i vår Config-klass.
Vi sätter alla värden enligt specifikationen och kör programmet och får ut responses med plats för inträffat event, tidpunkt samt ett unikt ID för händelsen. Vi förväntar oss inget specifikt utan ser när vi kör att det är logiska siffror vi får ut.

Nätverket är uppbyggt så att det skriver ut ett fält som är 50x50 noder stort, och jobbar med att uppdatera bland annat noder och skapar eventuella agenter och event. För en visuell representering av Rumor-Routing systemet när det körs sätter vi de olika tillstånden på noderna till olika slags tecken. För varje tidssteg som gås kommer även fältet att och den visuella representationen uppdateras.
Programmet saktas ned så att varje steg visas i 100 millisekunder för att man i den visuella representationen enklare ska se alla stegen. Hade vi inte haft med denna funktion för att enklare se stegen hade det som nu tar ungefär 10 minuter att se från start till slut, istället tagit runt 6 sekunder.

Testkörning

När vi i gruppen har kört programmen så har vi fått runt 80 svar per körning.

```
x: 230 y: 250 time: 6501 eventID: 1582
x: 370 y: 300 time: 4422 eventID: 1092
x: 10 y: 40 time: 3838 eventID: 959
x: 350 y: 250 time: 3015 eventID: 736
x: 320 y: 310 time: 1176 eventID: 285
x: 340 y: 490 time: 3390 eventID: 832
x: 60 y: 60 time: 6696 eventID: 1632
New run!

Process finished with exit code 0
```

Vid körning kommer x och y koordinat komma fram för varenda respons, tid, samt eventID. När testet har gjorts kan vi mäta antal svar vi har fått:

```
3184 chars, 85 line breaks
```

I detta fall fick vi 85 stycken rader som skrevs vilket resulterar i 85st svar på förfrågningar.

Resultat

Det finns 2500 noder, slumpen är 1/10000 att ett event ska skapas per nod. Alltså var fjärde gång så borde det skapas ett event. 50% av gångerna som en event skapas, så skapas också en Agent. alltså hälften av gångerna (1250). En agent tar 50 steg. 62500 steg kommer att tas av agenter. Om vi delar det på antal noder, så kommer de förmodligen besökas 25 gånger per nod. Det betyder att i slutet så kommer hela fältet vara i princip ihopkopplat. Vi tycker att ~80 responses är rimligt enligt den matematiken. Vi provade också med hälften av tiden och märkte att det stegar något exponentiellt, då vi fick runt ~34 efter flera tester.

Arbetskälla och Git-repo

Vi har använt oss av kod-miljön IntelliJ för att skriva programmet. Eftersom vi är flera i projektetet har vi även använt oss av Gitlab, en *“lagringstjänst för programmerare”*, där vi lagrar kodfilerna. Gitlab fungerar som så att man hämtar hem alla kodfiler och jobbar sedan på den man ska skriva. När man jobbat klart för dagen så slänger man upp den uppdaterade versionen på filen och ett litet meddelande på vad som skrivits just då. På detta sätt kan vi enkelt jobba med uppdaterade versioner av programmet när vi delat upp arbetet. Har man råkat göra fel kan man gå tillbaka till gamla versioner för att se vad som ändrats och hämta tillbaka versioner.

Länk till Gitlab-repo: https://git.cs.umu.se/id16jsn/grupp16ou3

Referenser

[1] Bild framsida: Noder, https://goo.gl/EqwOM8 [Hämtad 2017-04-28]

[2] A. Broberg, “Mer om kodkvalitet,” *pmet_vt16.key - F10_16.pdf*, 2017-02-24 09:04:36. [PDF] Tillgänglig: https://goo.gl/EH8pwv [Hämtad 2017-05-16]

[3] D. Braginsky, D. Estrin. “WSNA’02: Rumor Routing Algorithm For Sensor Networks,” *Rumor Routing Algorithm For Sensor Networks*, 2002-10-28 [PDF] Tillgänglig: https://goo.gl/XbaOpF [Hämtad: 2017-05-17]