

MicroQ

*Processororienterad programmering (1DT049) våren 2012.
Slutrapport för grupp 12*

**Björn Berggren 910705-0057
Marcus Enderskog 901025-2238
Simon Evertsson 911202-3370
Nanna Kjellin Lagerqvist 890103-1404**

2012-05-31

Innehållsförteckning

1	Inledning.....	4
1.1	Bakgrund	4
1.2	Syfte.....	4
1.3	Målsättning.....	4
1.4	Problemformulering	4
1.5	Avgränsningar.....	4
2	MicroQ.....	5
2.1	Funktionalitet	5
2.2	Användargränssnitt	6
2.2.1	Grafik	6
2.2.2	Erlang.....	7
2.3	Simuleringsresultat.....	Error! Bookmark not defined.
3	Programmeringsspråk	7
3.1	Fördelar	7
3.2	Nackdelar.....	7
3.3	Erlang.....	8
3.4	Java	Error! Bookmark not defined.
4	Systemarkitektur	8
4.1	Arkitektur java	8
4.1.1	Java	9
4.2	Arkitektur erlang	10
5	Samtidighet (concurrency)	10
5.1	Modeller	10
5.2	Samtidighet	11
5.3	Synkronisering	11
5.4	För- och nackdelar	11
5.5	Alternativ	11
5.6	Deadlocks	11
6	Algoritmer och datastrukturer	12
6.1	SimulationMain	12
6.2	Simulation.....	12
6.3	Communicator (java)	12
7	Förslag på förbättringar.....	13

8	Resultat.....	14
9	Reflektion	14
10	Installation och utveckling.....	14
10.1	Versioner	14
10.2	Ladda ner	14
10.3	Katalogstruktur	14
10.4	Kompilering	14
10.5	Automatiserade tester	14
10.6	Automatiserad generering av dokumentation.....	15
10.7	Start av systemet.....	15
11	Litteraturförteckning	16

1 Inledning

Projektet MicroQ har en nära koppling både till tekniken och till datavetenskapen men också till studenters vardag.

1.1 Bakgrund

Ett problem som ofta uppstår i studenters vardag är långa köer i lunchrummen. Detta problem kan lösas på olika sätt. Till exempel genom att öka antalet mikrovågsugnar i lunchrummet, begränsa tiden en student får värma sin mat eller sätta en begränsning på hur lång tid en student har på sig att, när det är möjligt, ta en ledig mikro.

I detta projekt har vi försökt få en uppfattning om hur dessa olika parametrar styr effektiviteten av mikroanvändningen. Detta har visualiserats med hjälp av en simulering.

1.2 Syfte

Det övergripande syftet med rapporten är att få kunskap och förståelse för hur en kö till mikrovågsugnar i studenters lunchrum (i fortsättningen kallad *mikrokö*) fungerar.

Vi vill undersöka hur olika parametrar så som antal mikrovågsugnar, antal studenter och hur lång tid maten ska värmas påverkar mikrokön.

1.3 Målsättning

Målet med rapporten är att ge ett underlag till att skapa en bättre miljö i studenters lunchrum. Genom ökat förståelse för hur en mikrokö fungerar kan eventuellt studenternas väntetider minimeras och utnyttjandet av varje mikro optimeras.

1.4 Problemformulering

För att besvara vårt syfte använder vi följande frågeställningar:

- Hur beror den genomsnittliga väntetiden i mikrokön av antalet mikrovågsugnar, hur ofta nya studenter anländer till kön samt hur länge varje student värmer sin mat?
- Hur länge står en mikro i genomsnitt tom, baserat på de givna parametrarna; hur många mikros som finns, hur ofta nya studenter anländer till kön samt hur länge varje student värmer sin mat?

1.5 Avgränsningar

Vi har valt att använda en simulering av en mikrokö för att besvara våra frågeställningar. Denna simulering använder givna parametrar som matas in av användaren innan simuleringen startas.

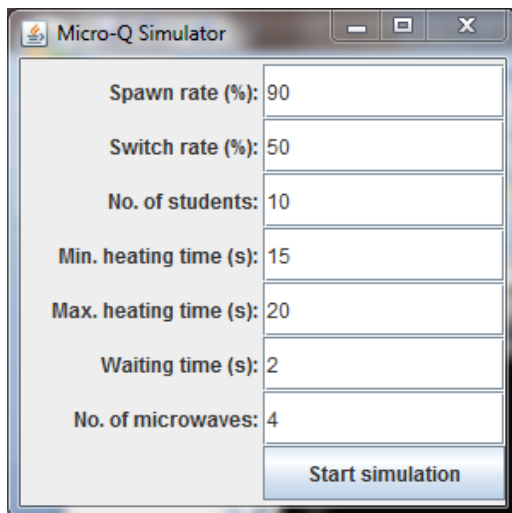
Dessa parametrar är

- *Spawn Rate* - Hur ofta nya studenter anländer till kön. Egentligen sannolikheten att en ny student anländer, anges i procent.
- *Switch Rate* - Hur ofta en student byter tillstånd. Med tillstånd menas om studenten är närvarande och redo att värma, eller frånvarande.
- *Number of Students* - Antalet studenter som ska gå genom simuleringen.
- *Waiting Time* - Väntetid, det vill säga så länge som en student kan vara frånvarande utan att förlora sin plats först i kön.
- *Minimum Heating Time, Maximum Heating Time* - Ett tidsintervall inom vilken tiden för att värma varje matlåda sedan slumpas fram, till exempel 2,00 till 5,00 minuter.
- *No. Of Microwaves* - Antal mikrovågsugnar

2 MicroQ

Här följer en beskrivning av hur systemet kan användas.

2.1 Funktionalitet



The image shows a software window titled "Micro-Q Simulator". It contains a list of parameters with corresponding input fields:

Spawn rate (%):	90
Switch rate (%):	50
No. of students:	10
Min. heating time (s):	15
Max. heating time (s):	20
Waiting time (s):	2
No. of microwaves:	4

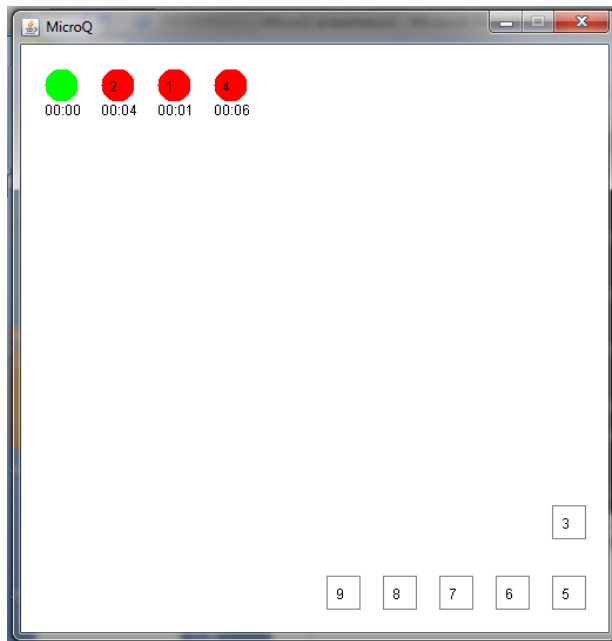
At the bottom of the window is a button labeled "Start simulation".

Figur 2-1, Ruta där användaren får mata in parametrar till simuleringen.

Systemet kan användas för att skapa en simulering av en mikrokö. Användaren får själv mata in ett antal parametrar (se avsnitt 1.5 *Avgränsningar*, ovan) innan simuleringen startas (Fig. 2-1). När simuleringen startas kan användaren följa systemets exekvering visuellt dels genom en grafisk representation av lediga och upptagna mikrovågsugnar och kön av studenter.

2.2 Användargränssnitt

Systemet har både ett graphical user interface (i fortsättningen kallat GUI) och ett terminalfönster för statusmeddelanden.



Figur 2-2, Den grafiska representationen av simuleringen

2.2.1 Grafik

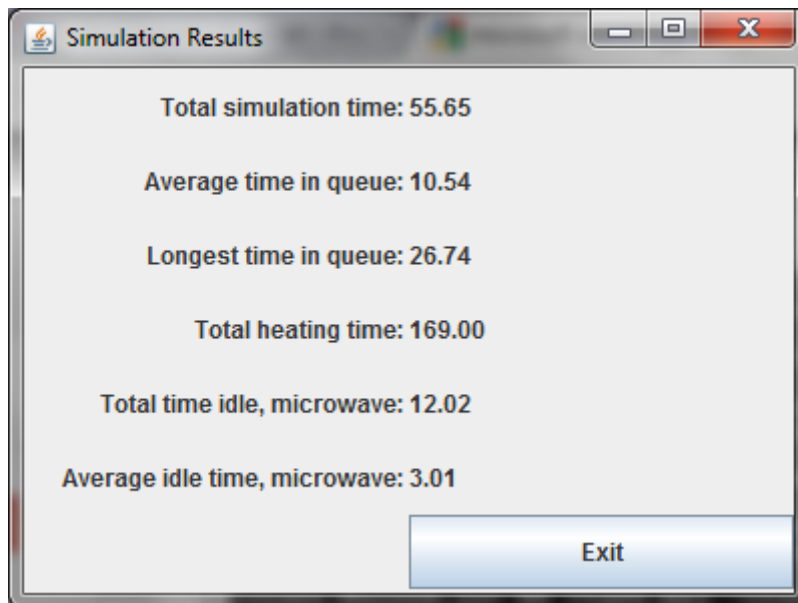
GUI:t(Figur 2-2) är helt och hållet byggt i Java. Detta eftersom det finns stöd för grafik i form av Javas färdigskrivna grafikbibliotek och verktyg för att tillverka GUI-layouter till IDE:n Eclipse¹. (Eclipse)

När programmet startar så visas en ruta där användaren ombeds att mata in simuleringsparametrar. När användaren har matat in alla parametrar klickar denne på knappen "Start Simulation" för att starta simuleringen och stänga ner inmatningsrutan.

När simuleringen startas så öppnas en ny ruta vilken representerar simuleringen. I denna ruta visas ett antal cirklar som representerar mikrovågsugnar, med tillhörande tid i formatet MM:SS . Antalet cirklar är lika många som det antal mikrovågsugnar användaren angav i inmatningsrutan. Cirkarna ändrar färg beroende på om de är lediga (grön färg) eller upptagna (röd färg). Tiden som visas är den värmtid som återstår för tillfället. Då tiden visar "00:00" är mikron ledig.

Vidare visas en representation av kön i form av rektanglar med en unik siffra per rektangel. Rektanglarna är uppräddade under mikrocirkklarna och representerar de tio första matlådorna i kön. Den rektangel som befinner sig längst till höger är den matlåda som befinner sig först i kö. När en mikro blir ledig, och om studenten som lådan tillhör är närvarande i lunchrummet, plockas lådan bort ur kön och en ledig mikro blir upptagen. Skulle inte studenten hinna tillbaka till lunchrummet inom den väntetid som användaren angivit så hamnar lådan automatiskt sist i kön.

¹ The Eclipse Foundation open source community website, <http://www.eclipse.org/>, 2012 05 23



Figur 3, Resultatrutan som innehåller resultatet från simuleringen.

När alla studenter har värmt sin mat beräknas simuleringen vara avslutad och en ruta med diverse simuleringsdata visas. När användaren är färdig med datat och vill avsluta programmet klickar denne på knappen "Exit".

2.2.2 Erlangkonsolen

Erlangkonsolen visar enbart meddelanden då eventuella fel uppstår. Till exempel om kontakten med Java skulle upphöra. För övrigt så visas endast meddelanden under uppstarten av systemet samt då programmet avslutas, som bekräftar att allt genomförts korrekt.

3 Programmeringsspråk

Vi har använt två programspråk. Detta val gjorde vi dels för att lära oss att skriva ett program i två olika språk och få delarna att fungera tillsammans, men också för att Erlang och Java har olika bra stöd för implementation av olika delar av programmet.

3.1 Fördelar

Vi har mest känt att det varit fördelar med att arbeta i två språk. Det gav en naturlig uppdelning av arbetsuppgifter inom gruppen och på så vis har varje persons kompetens utnyttjats, samtidigt som gruppen fått en chans att öva extra på kommunikation då det varit ett gemensamt ansvar att alla gruppmedlemmar hålls uppdaterade om vad som händer i projektets olika delar.

3.2 Nackdelar

Eftersom vi valde att implementera programmet i två språk så blev vi tvungna att hantera kommunikationen däremellan. När vi bestämde oss för just språken Erlang och Java så hade vi tänkt använda verktyget Erlide² till Eclipse, men vi fick det helt enkelt inte att fungera. Eftersom det riskerade att ta för mycket tid och fokus från kärnan i projektet så löste vi det genom att utveckla erlangdelen separat.

² Erlide – the IDE. Powered by Eclipse (<http://erlide.org/>)

En annan nackdel är att vår systemarkitektur blev mer komplex än den kanske kunde blivit om vi hade skrivit programmet i enbart ett språk.

3.3 Erlang

Erlang är ett funktionellt högnivåspråk utvecklat av telekomföretaget Ericsson i slutet av 1980-talet. *"Erlang fungerar bra som bas för system som ska vara felsäkra och fungera hela tiden, vara skalbara och distribuerade och hantera många simultana händelser"*.³ (Larsson, 2009)

Vi valde att använda Erlang eftersom det har lättviktsprocesser, *"Erlang processes are much more lightweight than OS threads"*⁴, vilket gör det enkelt att skapa nya processer som får sköta olika delar av programmet. Erlang har även actormodellen⁵ som är en modell för asynkron kommunikation och implementerar en mycket kodeffektiv struktur för att skicka meddelanden mellan processer. Erlang används till de delar av programmet som består av flera processer och erlangdelen av programmet sköter även studenternas tillstånd. (Marklund, 2012)

3.4 Java

Java är även det ett högnivåspråk men till skillnad från Erlang är det inte ett funktionellt utan ett imperativt objektorienterat programmeringsspråk. Java har inbyggt stöd för synkronisering⁶ av delade resurser vid användande av trådar⁷, samt för att implementera ett grafiskt användarinterface. I projektet använder vi Java till de delar av programmet som hanterar kön, mikrovågsugnar samt tidsmätning. (Lampka & Marklund, 2012) (Oracle.com)

Vi valde att använda Java eftersom språket tillhandahåller stöd för automatiserade tester, JUnit⁸, samt automatiserad generering av dokumentation, Javadoc⁹. Språket innehåller också många färdigskrivna bibliotek som vi använt till exempelvis köhantering och de grafiska delarna i programmet. Ericsson har även skrivit ett gränssnitt¹⁰ mellan Erlang och Java vilket möjliggjorde kommunikation mellan Java och Erlang. Detta var en förutsättning för att kunna implementera programmets delar i två skilda språk. (JUnit.org) (Oracle.com) (Ericsson AB, 2012)

4 Systemarkitektur

Systemets olika delar samt samverkan däremellan.

4.1 Arkitektur Java

Java kommunicerar med Erlang via mailboxes¹¹. Intern kommunikation mellan trådarna i Javadelen av programmet sker med hjälp av shared resources¹² i form av statiska variabler och metodanrop. (Ericsson AB) (Marklund, 2012)

³ <http://www.idg.se/2.1085/1.213789/svensk-sprakdoldis-gor-succ%C3%A9>

⁴ karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang* s. 18

⁵ karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang* s. 14-17

⁶ <http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

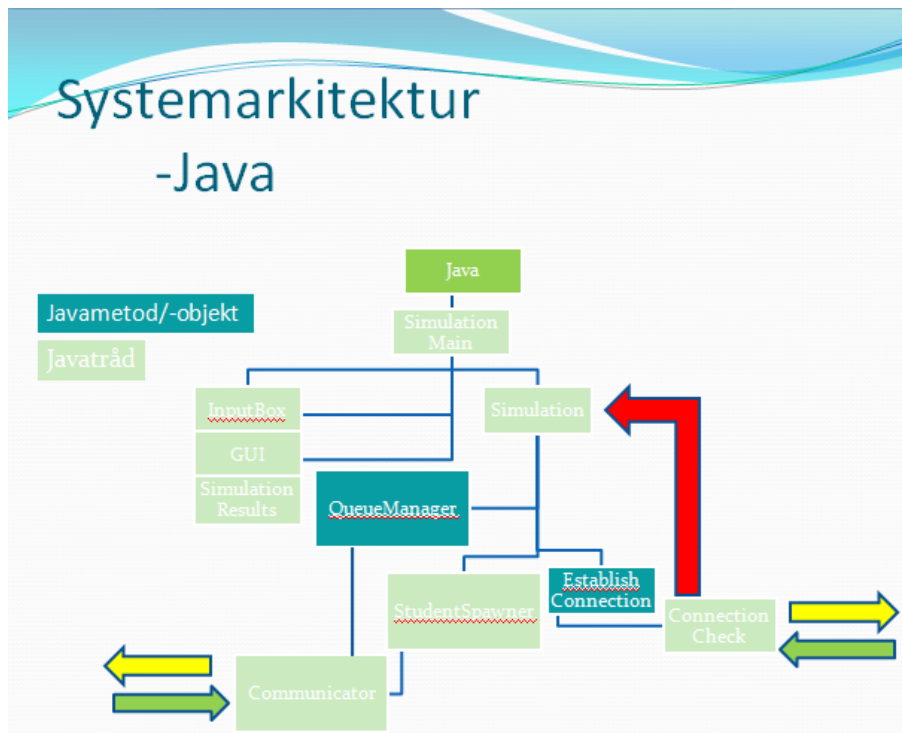
⁷ kai.lampka@it.uu.se, karl.marklund@it.uu.se, Uppsala Universitet 27 01 2012, *Chapter 4: Threads*

⁸ <http://kentbeck.github.com/junit/javadoc/latest/index.html?overview-summary.html>

⁹ <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>

¹⁰ <http://www.erlang.org/doc/apps/jinterface/index.html>

¹¹ mailboxes <http://www.erlang.org/documentation/doc-5.0.1/lib/jinterface-1.22/doc/html/com/ericsson/otp/erlang/OtpNode.Mailboxes.html>



Figur 4, Systemarkitekturen i Java

4.1.1 Java

SimulationMain (se Fig 4) är det första som körs i Java när programmet startar och det är här den grafiska delen av simuleringen körs. Till att börja med så startas en ny InputBox-tråd där användaren får mata in simuleringsparametrar. När användaren har matat in datat så startar GUI:t, simuleringen och datasamlaren.

I simuleringen sätts parametrarna som angetts av användaren. Sedan försöker kommunikation med Erlang upprättas. När uppkoppling erhållits startas en ConnectionCheck som ska kontrollera förbindelsen mellan Java och Erlang. Skulle förbindelsen tappas så meddelar ConnectionCheck SimulationMain att programmet ska avslutas. Simulation skapar även en ny instans av QueueManager, som håller reda på kön, samt startar en StudentSpawner vilken slumpmässigt genererar nya studenter.

Då en ny student eller om en redan existerande students tillstånd skall förfrågas så startas en ny Communicator, vilket är den del i Java som kommunicerar med Erlang.

¹² Marklund, Process Oriented Programming (1DT049), The Process Concept and Inter Process Communication s 34-41, 2012 01 24

4.2 Arkitektur Erlang

Erlang kommunicerar både internt (mellan processer) och externt (mellan Erlang och Java) med hjälp av message passing¹³. Om en erlangprocess vill skicka ett meddelande till Java så måste den känna till vilken mailbox den ska skicka till samt vilken javanod som den tillhör. (Marklund, 2012)



Figur 5, Systemarkitekturen i Erlang

När Erlang delen (Fig. 5) startas så börjar den en initiering. Där upprättar den kommunikation med Java och startar sedan en liknande ConnectionCheck som fungerar på samma sätt som ConnectionCheck i Java. Efter det startar Communicator vilket är den del i Erlang som skickar och tar emot meddelanden. Meddelanden kommer dels från Java och dels från tillståndprocesserna i Erlang. Tillståndprocesserna skapas efter att Java har bett om det. Då körs ClientSpawner vilken skapar en ny tillståndprocess och en tillhörande tillståndsbytarprocess. Tillståndprocessen håller ett tillsånd och svarar på "redo-förfrågningar" från Communicator när den är i sitt redo-tillstånd. Tillståndet byts av tillståndsbytare.

5 Samtidighet (concurrency)

Här diskuteras de modeller vi använt för concurrency.

5.1 Modeller

Vi använder message passing för kommunikation mellan erlangprocesser och mellan erlang och Java. Fördelen med detta är att vi slipper hantera delat minne, som hade varit ett alternativt sätt att kommunicera på. Tack vare det så kan vi utnyttja parallellism maximalt i erlangprocesserna.

¹³ karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang* s. 14-17

I Erlang använder vi flera processer och i Java använder vi istället trådar. Detta valde vi att göra eftersom detta är vad respektive programspråk har bäst stöd för i form av exempelvis färdigskrivna bibliotek. Trådarna i Java använder delat minne och därför måste vissa delar av programmet exekvera seriellt.

5.2 Samtidighet

En av tankarna bakom detta projekt var att öva oss på att programmera concurrent, det vill säga att använda fördelarna med samtidighet. Detta kunde vi utnyttja i erlangprocesserna samt SimulationMain, StudentSpawner, Communicator och Simulation i Java. Tack vare att Eprocesserna inte använder delat minne så kan de exekvera helt parallellt.

Trådar som skapas i Java använder gemensamma resurser (delat minne) och därför måste vissa delar av den koden synkroniseras för att bli threadsafe¹⁴, därför exekveras en del av programmet seriellt.

5.3 Synkronisering

Vi använder synkronisering för hantera mikrovågsugnarna. Mikrovågsugnarna representeras av en semafor vars värde är lika med antalet lediga mikrovågsugnar. Det är nödvändigt med denna synkronisering eftersom en student representeras som en tråd i Java och när flera studenter skapas så har de tillgång till gemensamma resurser. (Lampka & Marklund, 2012)

5.4 För- och nackdelar

Fördelar med denna implementation är bland annat att GUI:t är skapas väldigt enkelt då alla bibliotek som används är färdigskrivna och inbyggda i Java. Smidig synkronisering av åtkomsten till mikrovågsugnarna. Erlangdelarna blir väldigt små och överskådliga då inga tunga beräkningar utförs utan allt sköts nästan uteslutande genom meddelandeskickning.

Nackdelarna är dels den lilla prestandaförlust vi får då många meddelanden måste skickas mellan Java och Erlang samt att tester med Eunit¹⁵ har varit svåra att skriva i den implementation vi valt eftersom många av funktionerna inte returnerar något värde.

5.5 Alternativ

Ett alternativ till denna implementation hade varit att skriva all kod i ett språk. Då hade problem som kan uppstå då man jobbar mellan två olika språk kunna undvikas. Dessutom hade systemarkitekturen blivit mindre komplex.

5.6 Deadlocks

Som tidigare nämnts så använder vi oss av semaforer för att sköta åtkomsten till en mikro. En deadlock skulle kunna uppstå om någon Communicatortråd oväntat skulle dö efter att mikrosemaforen plockats. Då skulle en mikro alltid fortsätta vara upptagen även om ingen värmer sin mat i den.

Ett annat tillfälle då körningen skulle låsas är om InputBoxtråden skulle dö innan den har släppt sin setup_done-semafor som SimulationMain väntar på.

¹⁴ http://en.wikipedia.org/wiki/Thread_safety

¹⁵ <http://www.erlang.org/doc/apps/eunit/chapter.html>

6 Algoritmer och datastrukturer

6.1 SimulationMain

SimulationMain är det första som körs i Java när programmet startar och det är här den grafiska delen av simuleringen körs. Till att börja med så startas en ny InputBox-tråd där användaren får mata in simuleringsparameterar. Efter att Inmatningen av datat är färdigt så fortsätter exekveringen och en initiering av GUI:t startas.

```
211 void draw()
212 {
213     Graphics g = getGraphics();
214     Graphics bbg = backBuffer.getGraphics();
215
216     bbg.setColor(Color.WHITE);
217     bbg.fillRect(0, 0, windowWidth, windowHeight);
218
219     for(int i = 0; i < microwaves.length; i++) {
220         microwaves[i].draw(bbg);
221     }
222
223     queueRep.draw(bbg);
224
225     g.drawImage(backBuffer, insets.left, insets.top, this);
226 }
```

Figur 6, Ritningsloopen i SimulationMain

Efter det hamnar SimulationMain i sin ritningsloop (Fig. 5). Loopen kommer köras så länge som simuleringen är igång och kontakt mellan Java och Erlang existerar. I loopen körs drawmetoden med en frame rate på ungefär 30FPS. Drawmetoden ritar ut alla objekt som skall synas i fönstret.

När simuleringen är färdig så bryts loopen och simuleringsdatat från SimulationData visas på skärmen i resultatrutan (Fig. 3). Om loopen bryts av att kontakten med Erlang avbrutits visas istället en ruta med meddelandet "Connection Lost". Efter att rutan stängts ner så avslutas programmet.

6.2 Simulation

I Simulations konstruktor sätts parametrarna som angetts av användaren. Simulation skapar även en ny instans av QueueManager samt startar en StudentSpawner-tråd. När SimulationMain sedan startar Simulationtråden påbörjas simuleringen.

Simulation kommer då kalla på QueueManager's readyCheck-metod så länge som det finns någon student i kön och det finns en ledig mikro. Detta kommer genomföras tills det att antalet studenter som har värmt klart sin mat är lika med antalet studenter som användaren angivit.

När alla studenter värmt färdigt, skickar Simulation ett terminatemeddelande till Erlang och bryter ritningsloopen i SimulationMain.

6.3 Communicator (Java)

Instanser av Communicator är de trådar som sköter det mesta av kommunikationen med Erlang. Communicator kan skapas på två sätt. Antingen då en ny studentprocess ska spawnas i Erlang eller

då en mikro blivit ledig och en students tillstånd söks. En mailbox med ett unikt id för varje ny instans av Communicator skapas så att Erlang vet var den ska skicka svaret.

Då en ny student ska spawnas så skickas meddelandet `{mailbox-id, 'new_client', "Switch rate"}` till Erlang. Variabeln **mailbox-id** är det unika id som denna Communicator-tråd fått tilldelat, **'new_client'** är meddelandet som gör att Erlang vet att en ny tillståndprocess ska skapas.

Communicator väntar sedan på svaret från Erlang som innehåller den nyskapade tillståndprocessens PID. StudentSpawner:s räknare över antalet skapade studenter ökas sedan och den nya studenten läggs till i kön i QueueManager.

Om en befintlig students tillstånd ska kontrolleras så skickas meddelandet `{mailbox-id, 'ready', PID}` Variabeln **mailbox-id** är åter igen det unika id som denna Communicator-tråd fått tilldelat, **'ready'** är meddelandet som gör att Erlang vet att en tillståndprocess tillstånd eftersöks. **PID** är PID:t till studentens tillståndprocess.

Communicator väntar på svar från Erlang så lång tid som användaren matat in (*Heating Time*). Om svaret anländer innan tidsgränsen nåtts så sätts timern i ett Microwaveobjekt till studentens värmtid och startas. Communicator sover sedan under värmtiden och ber sedan Erlang att ta bort studentens tillståndprocess.

Om svaret inte kommit inom tidsgränsen så läggs Studentobjektet tillbaka i kön och mikrosemaforen släpps så att en ny student kan ta mikron.

7 Förslag på förbättringar

Något som vi skulle vilja göra om vi hade haft mer tid är att visa tillstånden (till exempel ready eller away) för de studenter vars matlådor ligger bland synliga matlådorna i GUI:t. Det hade då ytterligare förtydligat vad som händer under simuleringens gång.

Detta hade kunnat implementeras genom att varje studentprocess i Erlang skickat ett meddelande till en "lyssnartråd" i Java varje gång den bytte tillstånd. Javatråden skulle sedan uppdatera ett objekt som representerar en students tillstånd, vilket sedan hade ritats ut av GUI:t.

Fortsättningsvis är många variablerstatiskt implementerade, ofta med anledningen att det var den enklaste lösningen vid tillfället. Det är något vi önskat ändra på, givet att vi haft mer tid på oss. T ex hade man kunnat använda sig av get- respektive set-funktioner istället, och på så sätt undvika direkt åtkomst till variabeln.

Många av funktionerna i Erlang var svåra att testa. Till exempel communicator-funktionen, implementerad som en enda loop som skickar och tar emot meddelanden till och från Java. Det är inte lätt att undersöka om den fungerar som den ska i och med att den inte returnerar något värde av intresse. Hade vi haft tid hade vi kanske kunnat lösa det på något sätt, men vi valde inte att prioritera det.

Generellt hade vi velat snygga till koden lite mer och fixa kompatibiliteten på den automatiserade körningen till fler operativsystem än Windows, för just nu startas simuleringen från ett batch-script. Det finns även brister i dokumentation och tester.

8 Resultat

Tabell 1, Resultat av testkörning

	Körning 1	Körning 2	Körning 3
Waiting Time(s)	15	8	3
Idle time/Microwave(s)	10,28	9,13	0,94

Vi genomförde en enklare testkörning för att ta reda på hur dötiden beror av väntetiden. Vi utförde tre körningar med Spawn Rate: 90%, Switch Rate: 50%, 10 Studenter, en värtid mellan 15 och 20 sekunder samt fyra mikrovågsugnar. Resultatet (se Tabell 1) visar på att en minskning av väntetiden bidrar till en minskning av tiden en mikro står tom.

9 Reflektion

Ett oväntat och onödigt svårt problem har varit att sätta upp och använda ett repository i Github. Tanken var att använda Git-pluginen direkt i Eclipse för snabb distribution, men då Erlide fungerade så pass dåligt för oss tog vi beslutet att avstå från detta och istället använda bashversionen när vi väl fått allt att fungera. Detta har tagit onödigt lång tid och det visade sig att vi missförstått hur själva fildelningen används.

Vad gäller metodik för programutveckling har vi varken använt oss av *scrum* eller *extreme programming*, vilket vi kanske hade gjort annorlunda om vi fått börja om. Vi har dock inte haft något problem vid utvecklingen i och med att gruppen mestadels suttit fysiskt närvarande och fått direkt feedback och hjälp av varandra vid behov.

10 Installation och utveckling

Här nedan finner ni de uppgifter som behövs för att hämta och köra systemet.

10.1 Versioner

Vi har skrivit koden i Java i JDK1.7 och Erlang i Erlang 5.9.1

10.2 Ladda ner

Koden finns tillgänglig på <https://github.com/pilver/MicroQ>

10.3 Katalogstruktur

I mappen "doc/html" ligger dokumentationen av erlangkoden. I "ebin" mappen placeras de kompillerade .beam-filerna. I "src" ligger erlangkoden och i MicroQ-mappen ligger Javakoden som ett Eclipseprojekt. Javakoden återfinns i "MicroQ\src\javacom".

10.4 Kompilering

Erlangkoden kompileras med kommandot make. Javakoden har kompilerats i Eclipse.

10.5 Automatiserade tester

Eftersom vi inte hann ta reda på hur testning av message passing med EUnit fungerar så har vi ingen automatiserad testning i Erlang. I Java så kan ett automatiserat JUnit-test köras från Eclipse.

10.6 Automatiserad generering av dokumentation

Vi har använt EDoc och Javadoc som både fungerade bra och var smidiga att använda när vi väl lärt oss hur de fungerar.

10.7 Start av systemet

Systemet startas från filen startup.bat i projektmappen

11 Litteraturförteckning

Eclipse. (u.d.). *The Eclipse Foundation open source community website*. Hämtat från <http://www.eclipse.org/> den 23 05 2012

Ericsson AB. (u.d.). *erlang.org*. Hämtat från Java-Erlang Interface Library: <http://www.erlang.org/documentation/doc-5.0.1/lib/jinterface-1.2/doc/html/com/ericsson/otp/erlang/OtpNode.Mailboxes.html> den 31 05 2012

Ericsson AB. (den 01 04 2012). *erlang.org*. Hämtat från jinterface Reference Manual: <http://www.erlang.org/doc/apps/jinterface/index.html> den 24 05 2012

JUnit.org. (u.d.). Hämtat från JUnit API: <http://kentbeck.github.com/junit/javadoc/latest/index.html?overview-summary.html> den 23 05 2012

Lampka, K., & Marklund, K. (den 27 01 2012). Process Oriented Programming (1DT049), Chapter 4: Threads. Uppsala.

Lampka, K., & Marklund, K. (den 18 02 2012). Process-oriented Programming (1DT049), Synchronization --coordinating access to shared resources. Uppsala.

Larsson, P. (den 23 02 2009). *Svensk språkdoldis gör succé, Computer Sweden*. Hämtat från IDG: <http://www.idg.se/2.1085/1.213789/svensk-sprakdoldis-gor-succ%C3%A9> den 22 05 2012

Marklund, K. (den 23 03 2012). Process Oriented Programming (1DT049), Erlang: modules, TDD with EUnit, process supervision. Uppsala.

Marklund, K. (den 20 03 2012). Process Oriented Programming (1DT049), Generators, Coroutines, Actors and Erlang. Uppsala.

Marklund, K. (den 24 01 2012). Process Oriented Programming (1DT049), The Process Concept and Inter Process Communication. Uppsala.

Oracle.com. (u.d.). Hämtat från Javadoc Tool Home Page: <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html> den 23 05 2012

Oracle.com. (u.d.). Hämtat från The Java Tutorials, Synchronized Methods: <http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html> den 24 05 2012