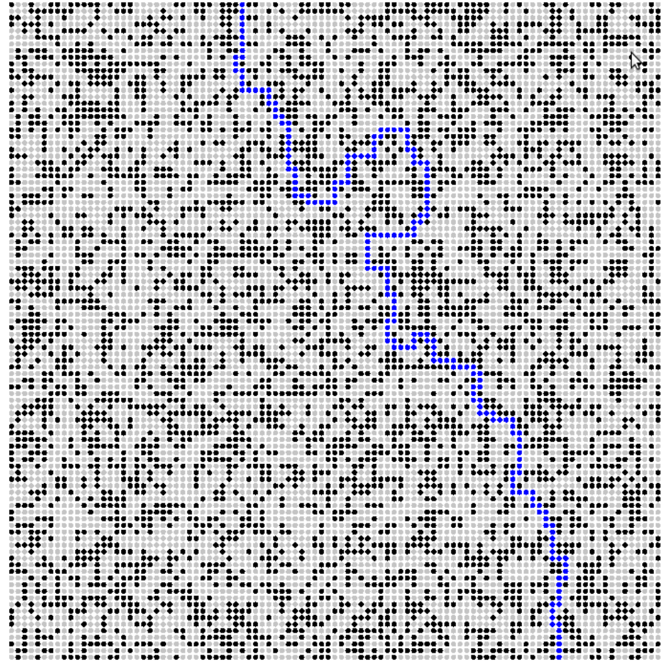
Kungl. Tekniska Högskolan
NADA



Händelsestyrd vs. tidsstyrd simulering

En studie i simulationsparadigmer

Inlämningsdatum:
2012-04-12

Författare:
Max Roth
070-698 74 21
maxroth@kth.se

Fredrik Hillnertz
070-094 64 16
frhi@kth.se

Handledare:
Henrik Eriksson

Abstract

This essay is a qualitative study of two simulation paradigms: *event-based* and *time-based* simulation. The purpose is to find positive and negative characteristics in implementations where one of the simulation paradigms has been applied. Two models were implemented, a queue model and a percolation model. It appeared to be a bit harder to understand event-based simulation while the time-based simulation was easy to implement. Time-wise it took about the same amount of time to implement the queue model with both simulation paradigms while the percolation model required more time to implement with event-based simulation. This was mostly due to the fact that it was harder to interpret the percolation model in terms of events.

By carefully observing the execution time while at the same time changing the parameters it was possible to distinguish characteristics of the different implementations. Event-based simulation had the best performance in most cases, which was influenced a lot by what time-step was chosen for the time-based simulation. It was also found that both implementations had very different execution times depending on which parameters were variated. The overall conclusion is that event-based simulation is the most time-efficient solution but also the most difficult one. One should have the approach to attempt implementation of an event-based simulation and use time-based simulation in case it is not possible. If you have the time you could also try to implement both variants to get the best of both.

Sammanfattning

Den här uppsatsen är en kvalitativ studie av två simulationsparadigmer: *händelsestyrning* och *tidsstyrning*. Syftet är att ta reda på olika positiva och negativa egenskaper hos implementationer där man tillämpat en av simulationsparadigmerna. Två modeller implementerades, en kömodell och en perkolationsmodell. Det visade sig vara aningen svårare att förstå sig på händelsestyrning medan tidsstyrningen var enkel att implementera. Tidsmässigt tog det ungefär lika lång tid att implementera kömodellen med båda simulationsparadigmerna medan perkolationsmodellen tog längre tid att implementera med händelsestyrning. Detta berodde till stor del på att det var svårare att tolka perkolationsmodellen i termer av händelser.

Genom att noggrant granska exekveringstiden samtidigt som parametrarna varierades gick det att utröna egenskaper hos de olika implementationerna. Händelsestyrning presterade bäst i de flesta fallen, vilket berodde mycket på vad för tidssteg man valde till den tidsstyrda simulationen. Man fann också att båda implementationerna hade väldigt olika exekveringstid beroende på vilka parametrar som varierades. Den sammanfattade slutsatsen är att händelsestyrd simulering är den tidseffektivaste lösningen men även den svåraste. Man bör ha inställningen att försöka implementera simulationen med händelsestyrningen och byta till tidstyrning om det inte går. Utifall man har tid kan man även försöka använda båda simulationsparadigmerna för att få ut det bästa från båda.

Förord

Denna rapport har utförts i samband med kursen DD143X – *Examensarbete inom Datalogi, grundnivå 15hp.* Kursen lästes på civilingenjörsutbildningen i datateknik på KTH. Rapporten har skrivits av Max Roth och Fredrik Hillnertz. I överlag har båda två samarbetat på alla delar av projektet, rapportskrivande såsom programmering. Naturligtvis så har vissa delar arbetats på mer av den ena eller andra personen. Specifikt har Max lagt mer tid på formatering, språk och innehåll av rapporten medan Fredrik har ägnat mer tid åt programmering, testning och felsökning.

Innehållsförteckning

1 Introduktion....1
1.1 Syfte....1
1.2 Problemställning....2
2 Bakgrund....3
2.1 Simulering....3
2.2 Diskret-händelse simulering....3
2.3 Kontinuerlig simulering....3
2.4 Perkolation....4
3 Genomförande....5
3.1 Kösimulation....5
3.1.1 Modell....5
3.1.2 Implementation....6
3.1.3 Körexempel....11
3.2 Perkolationssimulation....11
3.2.1 Modell....11
3.2.2 Implementation....13
3.2.3 Körexempel....19
4 Evaluering av implementationerna....20
4.1 Kösimulation....20
4.1.1 Analys av exekveringstid....20
4.1.2 Statistisk analys....25
4.2 Perkolationssimulation....26
4.2.1 Analys av exekveringstid....26
4.2.2 Statistisk analys....31
4.3 Implementationssvårighet....32
5 Slutsats....34
6 Referenser....36

1 Introduktion

Med ökande processorkraft och teknologi finns det goda möjligheter att lösa problem som inte går att lösa för hand. Dock finns det nästan alltid något problem som är av en utredande natur där det inte är självklart vad man vill få fram. Det kan vara frågan om hur ett leveranssystem mellan olika lager fungerar under drift eller hur något fysikaliskt fenomen beter sig. Ett tillvägagångssätt till att förstå dessa problem och kanske även lösa dem är att simulera. Man modellerar problemsituationen och implementerar sedan modellen som ett simuleringsprogram. Nu när simulering har funnits i ett par år har diverse olika metoder att simulera på uppkommit. Vilken metod bör man välja? Det är vad denna uppsats ska försöka ge insikt i.

1.1 Syfte

Målet med denna uppsats är att göra en kvalitativ jämförelse mellan de två simuleringsparadigmerna, *tidsstyrd* och *händelsestyrd* simulation (se avsnitt 2.1-2.3 för dess definitioner). Deras för- och nackdelar ska utrönas utifrån implementationerna. Författarna vill först och främst ta reda på vilken av de två tillvägagångssätten som är effektivast, alltså vilken av dem som kan exekveras snabbast. Men aspekter såsom hur komplicerat det är och hur lång tid det tar att programmera modellerna på de två olika sätten ska också undersökas.

1.2 Problemställning

När uppsatsen planerades sattes följande kriterium upp:

- *Formulera två modeller som kan implementeras både som tidsstyrd simulation och händelsestyrd simulation. Modellerna skall innehålla slumpmässiga element samt variationer och en av modellerna skall även illustrera något som inte är enkelt beräkningsbart.*
- *Analysera implementationerna utifrån följande aspekter:*
    - *Exekveringstid.*
    - *Implementationstid.*
    - *Implementationssvårighet.*
    - *Simuleringsresultat.*
- *Dra slutsatser utifrån resultaten, bland annat om vilken av simuleringssätten som passar bäst för modellerna som specificerats.*

De två modellerna som formulerades blev till slut en enkel kömodell och en något mer avancerad perkolationsmodell. Perkolationsmodellen är baserad på perkolationsteorin som beskrivs i avsnitt 2.4.

Sammanfattningsvis så är alltså frågeställningen som ska besvaras följande:

" Vilka för- och nackdelar finns det med tidsstyrning och händelsestyrning? "

och

" Är tidsstyrning eller händelsestyrning bättre än den andre i någon aspekt? "

2 Bakgrund

I detta avsnitt introduceras de tre termerna som är mest nödvändiga för att förstå innehållet i uppsatsen. Dessa termer är diskret-händelse simulering, kontinuerlig simulering och perkolationsteori. Perkolationsteorin används i perkolationsmodellen, som presenteras senare (avsnitt 3.2.1).

2.1 Simulering

Diskret-händelse simulering, kontinuerlig simulering och Monte-Carlo simulering utgör tillsammans en uppdelning av simulationsteori [1]. Monte-Carlo simulering kommer inte att behandlas vidare, utan det är diskret-händelse och kontinuerlig simulering som är det centrala. Dessa refereras till som *simulationsparadigmer* i denna uppsats. Den största skillnaden mellan dessa två simuleringsparadigmer är hur man hanterar tid och på så sätt styr simulationens gång. Beroende på vilken av dessa två simuleringsparadigmer man använder sig av så blir programmets struktur väldigt annorlunda.

2.2 Diskret-händelse simulering

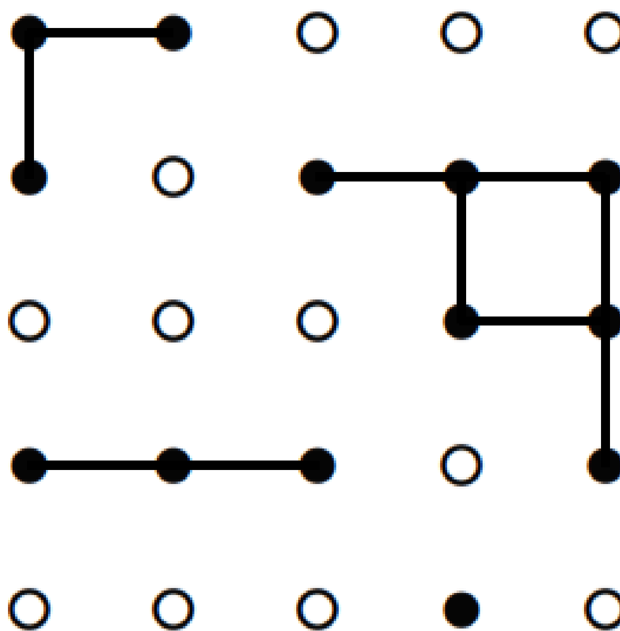
Diskret-händelse simulering innebär att man hanterar simulationens tillstånd genom att definiera olika *händelser*. En *händelse* inträffar ögonblickligen vid en viss tidpunkt [1]. Det som karaktäriserar diskret-händelse simulering är att tiden (i simulationen) endast rör sig från en händelse till en annan. Detta kan leda till stora hopp i tiden, då ett visst tidsintervall inte innehåller någon händelse. Man använder händelser för att uppdatera simulationens tillstånd. Denna grundprincip är det som kommer refereras till som *händelsestyrning*.

2.3 Kontinuerlig simulering

I kontinuerlig simulering så brukar modellen bestå av matematiska formler som bestämmer vad som ska hända. Istället för att stipulera händelser vid diskreta tidspunkter så observerar man istället systemet kontinuerligt [1]. Detta gör man genom att föra fram tiden med ett valt tidssteg. För varje tidssteg undersöker man om systemets tillstånd ska ändras. En klassisk modell lämpad för kontinuerlig simulation är “predator-prey” modellen [2]. I den modellen har man två populationer, rovdjur och byten, som beror direkt av varandra. För att projicera storleken av populationerna efter en viss tid så kan man beräkna populationen delvis, dvs. för varje tidssteg, fram tills den önskade tiden. Ju mindre tidssteg, desto noggrannare resultat. Detta sätt att kontrollera simulationen är det som kan kallas *tidsstyrning*.

2.4 Perkolation

Enligt Svenska Akademins elektroniska Ordbok (SAO) [3] så är perkolation definierat som processen där vätska passerar igenom en perkolator. En perkolator har i sin tur definierats som bland annat en typ av kaffekokare i samma upplaga. Bilden av vatten som passerar ett filter är en bra bild på vad *perkolation* är. Uppfattningen i denna uppsats är att perkolation innebär att vätska passerar igenom ett poröst material. Nuförtiden använder man även termen för att beteckna *perkolationsteori*, vilket är det som perkolationsmodellen är baserad på.



Figur 2.4.1: Bilden är hämtad från [4] och illustrerar site-perkolation. De hörn (sites) som är "öppna" är svarta medan kanterna mellan dessa hörn är dess förbindelser (där 'vätska' kan ta sig igenom).

Inom perkolationsteori studerar man beteendet som slumpgenererade klustergrafer uppvisar. Ett kluster innebär en mängd hörn. Det finns olika modeller för att beskriva dessa klustergrafer. En av de mer välkända modellerna är "site-percolation" (eng: site, sv: plats) [4]. En instans av en "site-percolation" modellen visas i figur 2.4.1. I figuren så finns det svarta hörn, vita hörn och streck mellan svarta hörn. Alla svarta hörnen är "öppna", dvs. de kan släppa igenom "vätska". Dessa svarta hörn utgör tillsammans grafens kluster. De vita hörnen är "stängda". En svart kant mellan två hörn representerar att "vätska" kan transportera sig mellan hörnen, vilket sker om båda hörnen är "öppna" och är närliggande (inte diagonalt). Varje hörn har med sannolikhet p att vara "öppen" och sannolikhet 1-p att vara "stängd". Hörnen är också oberoende av varandra. Problemet man beskriver med modellen är följande: om man genererar en graf som i figur 2.4.1 slumpmässigt med ett givet p, hur stor sannolikhet är det att det finns en stig från toppen av grafen till botten?

3 Genomförande

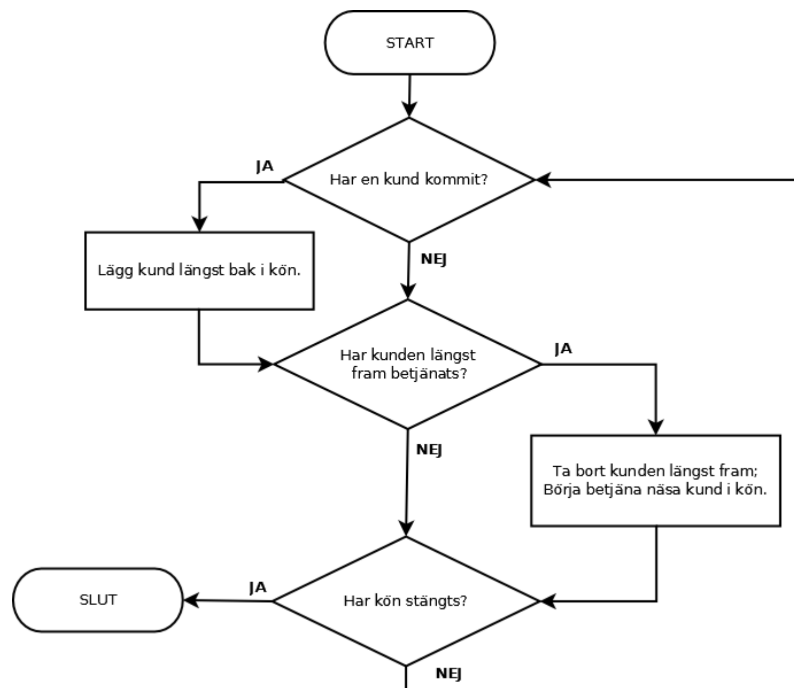
Denna sektion går in på hur modellerna både specificerats och implementerats. Alla implementationer har gjorts i det objektorienterade språket Java [5]. Själva programmeringen samt testningen av implementationerna har gjorts i Eclipse [6] . Båda modellerna har illustrerats med flödesdiagram. Varje implementations struktur kan överskådas i deras UML-klassdiagram, som följer version 2.0 av UML-standarden [7]. För att förklara implementationerna beskrivs varje klass dels med ord men också med pseudokod. Förutom att förklara implementationerna så visas även exempel på körningar.

3.1 Kösimulation

3.1.1 Modell

Modellen avbildar ett enkelt kösystem med en betjäningsstation. Kunder kommer in i systemet, ställer sig i kön för att bli betjänad och kommer ut ur systemet efter att ha blivit betjänade. Kundernas ankomsttider är Poissonfördelade, vilket innebär att kunderna ankommer slumpmässigt och oberoende av varandra med en viss frekvens, till exempel: en kund per minut. Det betyder emellertid att tiden mellan två kunders ankomster är exponentiellt fördelad. Detta faktum används i implementation. Kön är en vanlig “först in först ut”-kö (FIFO). För enkelhetens skull är tiden det tar för en kund att betjänas konstant (dvs. lika för alla kunder). Kunder som är kvar i systemet efter att tidsintervallet har löpt ut får ingen betjäning.

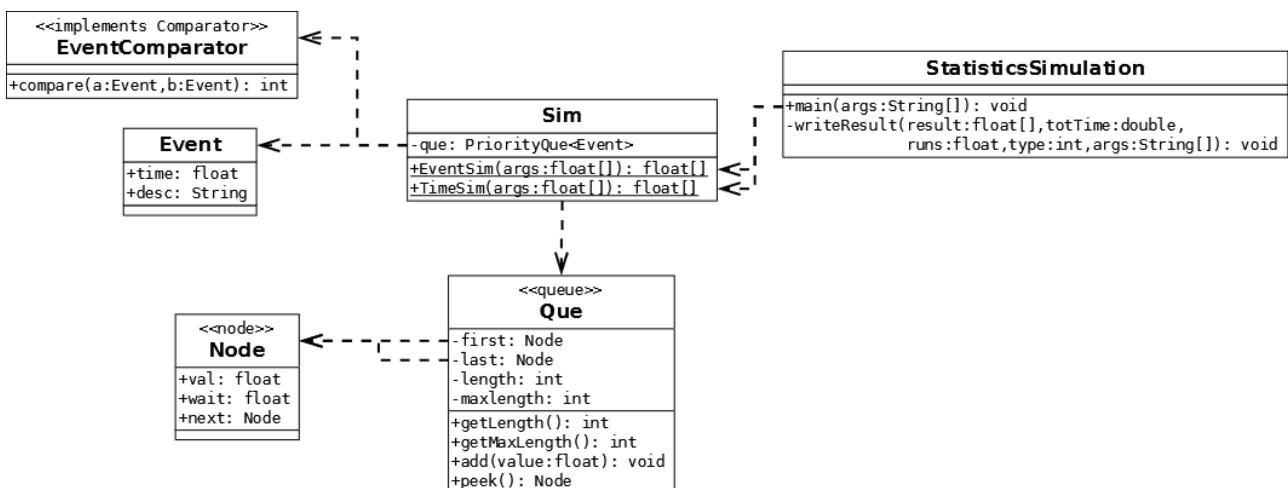
Figur 3.1.1. nedan illustrerar kömodellen som beskrivits. För den tidsstyrda simuleringen kan man i princip översätta flödesdiagrammet direkt till implementationen. Man behöver enbart gå fram ett tidssteg varje gång man kommer till if-blocket “*Har en kund kommit?*”. För händelsestyrningen däremot kan man inte använda samma metodik. Anledningen är att man i händelsestyrning inte nödvändigttvis behandlar if-blocken “*Har en kund kommit?*” och “*Har kunden längst fram betjänats*” i samma ordning som i flödesdiagrammet. Istället för att fråga sig om det har hänt för varje tidssteg så bearbetar man händelserna. De händelserna som formulerades för händelsestyrningen var "*arrival*" och "*depature*". Den tolkning som gjorts av modellens flöde för händelsestyrd simulering är att först generera alla ankomster för hela tidsspannet och sedan uppdatera simulationen genom att bearbeta dessa ankomster.



Figur 3.1.1: Flödesdiagram av kömodellen.

3.1.2 Implementation

I figur 3.1.2 kan vi se den övergripande klasstrukturen i implementationen. *Sim*-klassen är den centrala klassen i implementationen och binder samman alla klasser för att utföra själva simulationen. *StatisticsSimulation* anropar metoderna i *Sim*, samlar statistik från en eller flera simulationer och bearbetar simulationsresultatet för utskrift.



Figur 3.1.2: UML-klassdiagram som visar strukturen i implementationen av kösimulationen. Bara de viktigaste variablerna och metoderna är med för att diagrammet ska vara enkelt att sätta sig in i.

3.1.2.1 Event

Denna klass beskriver en händelse med en beskrivning (en String) och en tidsstämpel (en float). Tidsstämpeln är när händelsen har skett och används i *EventComparator* för att jämföra två stycken *Event*-objekt. *Event*-klassen används bara i den händelsestyrda simulationen (*EventSim*).

3.1.2.2 Que

Denna klass motsvarar betjäningskön i simulationen. Varje element i *Que*-listan är en *Node*. En *Que* instans håller bara reda på första och sista elementet, precis som en vanlig *Queue*. Det som skiljer *Que*-klassen från en vanlig *Queue* är metoderna *getMaxLength* och *getLength*. De returnerar, som namnet implicerar, längden på kön och den längsta kö som funnits i en *Queue* instans. Dessa metoder används för att generera statistik efter en simulationskörning.

3.1.2.3 EventComparator

EventComparator implementerar *Comparator* gränssnittet (eng: interface) och dess metod, *compare.* Anledningen till att *EventComparator* finns är för att *PriorityQueue*-klassen, som finns i Java API:t [5], behöver den för att kunna jämföra och sortera *Event*-instanser. Jämförelsen mellan två *Event*-instanser görs på deras tidsstämplar. *EventComparator* är programmerad så att den händelse som plockas ut ur prioritetskön alltid är den händelse som inträffat tidigast. Genom att använda *PriorityQueue* så kan vi enkelt gå igenom händelser enligt deras tidsstämplar, (och därmed gå från händelse till händelse) med en bra tidskomplexitet. Insättning tar O(log n), där n är antalet element som finns i prioritets kön) och utplockning tar O(1).

3.1.2.4 Sim

Sim klassen samlar själva simulationstyperna och det är via *Sim* som en simulation av kömodellen startas ifrån. Det är metoderna *EventSim* och *TimeSim* som motsvarar händelsestyrd respektive tidsstyrd simulering. Båda metoderna tar, förutom tidssteget, samma indata. Indatan till metoderna är: betjäningstiden (tiden det tar för en kund i "kassan"), tidsspannet *stopTime* (hur länge kön ska simuleras), ankomstfrekvensen *arrivalRate* (t.ex 60 skulle innebära att det är $\frac{1}{60}$ sannolikhet att en kund kommer per tidsenhet) och tidssteget Δt. En viktig detalj att komma ihåg är att tidssteget, tidsspannet, betjäningstiden och ankomstfrekvensen alla måste ha samma tidsenhet. Det är nämligen en förutsättning som gjorts för alla beräkningar i programmet.

<u>*TimeSim*</u>
Denna metod simulerar en kö med tidsstyrd-simulering efter de givna inparametrarna. Pseudokod 3.1.1 förklarar hur *TimeSim* fungerar. Simulering som *TimeSim* genomför startar vid tiden *t=0* och körs tills t överskrider körtiden, alltså när *t >= stopTime*. Vid varje tidssteg avgörs det om en kund ankommer och det undersöks om den kund under betjäning är klar. Om kunden är klar kan nästa kund i kön påbörja sin betjäning. När dessa satser är klara uppdateras tiden med Δt och samma procedur upprepas igen.

```
while( t <= stopTime ) {
    if( customerArrival() ) {
        Add new customer to que
        Customers++
    }
    if( Que not empty ) {
        if( firstInLIne().firstInLineTime >= serviceTime ) {
            que.removeFirst()
            satisfiedCustomers++
        }
        else {
            firstInLine().firstInLineTime += Δt
        }
    }
    t += Δt
}
```

Pseudokod 3.1.1: Huvud-loopen i TimeSim metoden.

En av de viktiga aspekterna i simulationen är hur kundankomsterna beräknas då tidssimulering och händelsestyrd-simulering skiljer sig väldigt mycket på den punkten. Denna beräkning görs i *customerArrival* i pseudokod 3.1.1 och förklaras i pseudokod 3.1.2 nedan:

```
boolean customerArrival() {
    upperRoof = Round( 1/ ( Δt *  1/arrivalRate )) )
    if( RandomInteger( [0, upperRoof) ) == 0 ) {
        return true
    }
    return false
}
```

Pseudokod 3.1.2: Förklarande kod för hur en ankomst beräknas.

Låt oss ta ett exempel. Låt Δt = 0.5, *arrivalRate* = 60 (dvs. $\frac{1}{60}$ kund per tidsenhet) och låt tidsenheten vara i sekunder. För varje halv sekund ska alltså chansen vara $\frac{1}{120}$ att en ny kund ankommer (varav väntevärdet blir: $\frac{60}{0.5} \cdot \frac{1}{120} = 1$, dvs. 1 kund i minuten). Variabeln *upperRoof* kommer då att bli 120. Sedan tas ett godtyckligt heltal mellan 0 och 120 (0 inkluderat, 120 exkluderat). Om det heltalet blir 0 ankommer en ny kund och chansen för det är precis $\frac{1}{120}$.

EventSim
Denna metod simulerar en kö med händelsestyrd-simulering. Händelserna som simuleringen kommer att styras av lagras i en prioritetskö som sorteras enligt händelsernas tidsstämpel (se avsnitt 3.1.2.3). I princip så kommer *EventSim* att genomföra kösimulationen genom att bearbeta händelserna i prioritetskön. Det första som sker, innan man börjar bearbeta händelser, är att kundankomsterna genereras. I pseudokod 3.1.3 visas det hur detta görs. För varje iteration som generer *exp* en tid mellan förra ankomsten och framåt. Denna tid är när nästa ankomst inträffar. Ankomsten i dess händelseformat läggs sedan in i prioritetskön. Om den genererade tiden överstiger tidsspannet så ska inte fler händelser genereras, vilket villkoret i while-satsen bekräftar.

```
t = 0
while( t <= stopTime) {
    t += exp( 1 / arrivalRate )
    customers++
    Add Event = ( “arrival”, t ) to priorityQue
}
```

Pseudokod 3.1.3: Visar hur ankomster skapas i EventSim metoden.

En avgörande faktor i skapandet av ankomster är funktionen *exp*. Funktionen beräknar hur lång tid det tar tills nästa kund anländer beroende på den angivna ankomstfrekvensen. Den kan uttryckas som i pseudokod 3.1.4 nedan.

```
double exp( λ) {
    double real = RandomReal( (0, 1] )
    return -ln( real ) / λ
}
```

Pseudokod 3.1.4: Sannolikhetsberäkningen för en kundankomst.

Först antas variabeln *real* till ett godtyckligt reellt tal mellan 0 och 1 (0 exkluderat, 1 inkluderat). För att förtydliga; det innebär att det kan bli ett väldigt litet tal. Det minsta talet som en double kan anta är 2^{-1074} enligt Java dokumentationen [5]. När real fått ett värde utförs beräkningen som ger en tid. Beräkningen är fördelningsfunktionen för exponentialfördelning där man löst ut x, givet en sannolikhet. Fördelningsfunktionen för exponentialfördelning är $1-e^{-\lambda x}$.
Ekvationslösningen kan granskas i (3.1):

$$\begin{aligned} p &= 1-e^{-\lambda x} \\ 1-p &= e^{-\lambda x} \\ \ln(1-p) &= \ln(e^{-\lambda x}) \\ \ln(1-p) &= -\lambda \cdot x \cdot \ln e \\ x &= -\frac{\ln(1-p)}{\lambda} \end{aligned} \tag{3.1}$$

För att förenkla beräkningen låter vi *real* motsvara *1-p*. Det sista ledet i ekvationen motsvarar då beräkningen som görs på return-satsen i pseudokod 3.1.4.

När ankomsterna har skapats körs sedan simulationen genom att ta händelser från prioritetskön. Simuleringen slutar antingen om inga fler händelser finns i kön eller om tidsspannet överskridits av en händelses tidsstämpel. I denna modell finns två sorts händelser: "*arrival*" och "*departure*". Beroende av vilket sorts händelse som plockas ut så görs olika saker. Pseudokod 3.1.5 nedan visar hur detta görs i *EventSim*.

```
while( priorityQue not empty && priorityQue.peek().time < stopTime) {
    event =  priorityQue.pop()
    t = event.time
    if ( event = “arrival” ) {
        que.add( t )
        if( first in line ) {
            Add Event = ( “departure”, t + serviceTime ) to priorityQue
        }
    }
    if (event = “departure”) {
        que.pop()
        satisfiedCustomers++
        if( someone next in line ) {
            Add Event = ( “departure”, t + serviceTime ) to priorityQue
        }
    }
}
```

Pseudokod 3.1.5: Denna kod visar hur händelser bearbetas från prioritetskön.

För varje ”arrival”-händelse så läggs en tidsstämpel in i kön som motsvarar kunden. Samtidigt undersöks det om kunden är först i kön, eftersom kundens betjäning då kan påbörjas. Om en ”depature”-händelse plockas ur kön så vet vi att nästa person i kön kan påbörja sin betjäning samt att totalt en till kund har betjänats.

3.1.2.5 StatisticsSimulation

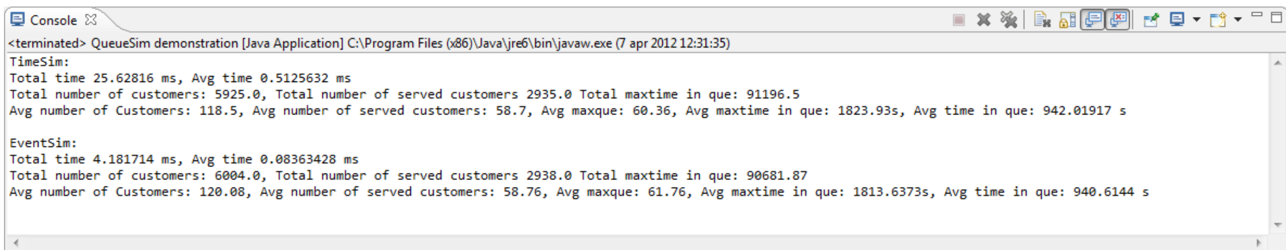
StatistiscSimulation utför både den händelsestyrda och tidsstyrda simulationen med de inparametrar som angivits samt genererar statistik. Förutom simulationsresultaten så innehåller även statistiken tidsmätningar för att kunna jämföra effektiviteten i båda implementationerna. För tidsmätningen har Javas inbyggda metod *System.nanoTime* använts. De specifika parametrarna som *StatistiscsSimulation* tar in är tidssteg, betjäningstid, tidslängd, ankomstfrekvens och antal körningar.

Tidsstegsparamtern används bara i den kontinuerliga simulationen medan betjäningstiden, tidslängden och ankomstfrekvensen används i båda simulationerna. Tidslängden är hur länge en simulation ska pågå. Varje körning motsvarar både en tidsstyrd simulation och en händelsestyrd simulation. För att underlätta analysen av statistiken så skrevs en metod *writeResult*. Denna metod använder ett bibliotek vid namn JExcelAPI [9] för att skriva utdatan som visas i terminalen till en

excelfil. Även inparametrarna för körningen lagras i denna excelfil.

3.1.3 Körexempel

Utdatan från en körning av kösimulationen består av följande värden: exekveringstiden, antalet kunder, antalet betjänade kunder, kötid för alla kunder samt den högsta kötiden för en enstaka kund. Varje värde har skrivits ut dels som det summerade värdet för alla körningar men även som medelvärde över alla körningar. I figur 3.1.3 visas en körning av kösimulationen.



Figur 3.1.3: Utdata från en körning av kösimulationen i Eclipse. Inparametrarna för denna körning var: "0.5 60 3600 30 50".

3.2 Perkolationssimulation

3.2.1 Modell

Modellen är baserad på "site-percolation" modellen (se avsnitt 2.3). Jämförelsevis så är den enda skillnaden att varje hörn representerar en atom. När ett hörn är öppet eller stängt så motsvarar det att atomen antingen har sönderfallit eller fortfarande är intakt. Vätska kan rinna igenom där det finns "tomrum" bland atomerna. En konsekvens av atomtolkningen är att hörnens öppenhet inte bestäms likadant. I "site-percolation" modellen genererar man en graf där varje hörns tillstånd beror på en sannolikhet *p*. I modellen som formulerats, däremot, så ska atomerna observeras över en viss tid där de kan sönderfalla. Hörn öppnas då kontinuerligt enligt atomens halveringstid.

Mängden av atomer efter en period av radioaktivt sönderfall är exponentialfördelad. En atoms halveringstid kan översättas till koefficienten τ i (3.2), som är ekvationen för exponentiellt sönderfall [10 s55]:

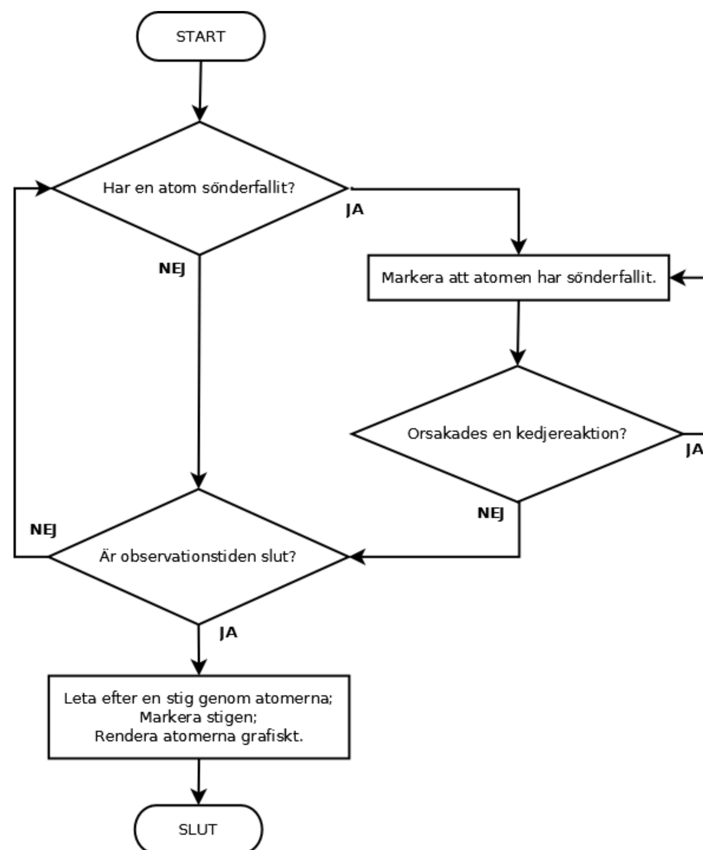
$$N(t) = N(t_0) \cdot e^{-\frac{t}{\tau}} \tag{3.2}$$

Relationen mellan τ och halveringstiden repeteras i (3.3).

$$\begin{aligned} T_{\frac{1}{2}} &= \ln 2 \cdot \tau \\ \tau &= \frac{T_{\frac{1}{2}}}{\ln 2} \end{aligned} \tag{3.3}$$

Ekvationerna (3.2) och (3.3) har använts i implementationen. Istället för att räkna på en mängd som i (3.2) så räknar vi istället på sannolikhet. Då får vi en beräkning för en atoms sönderfall som liknar det sätt som ankomster beräknas i kösimulationen.

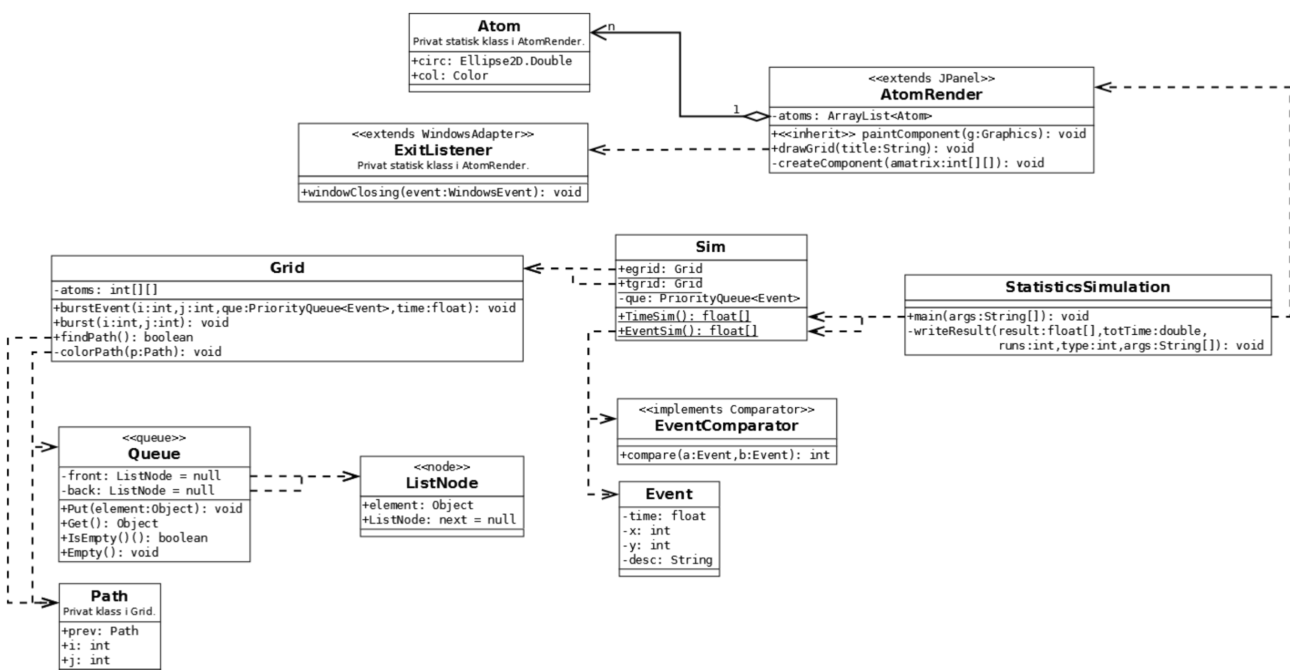
Ett av målen med denna modell var att simulera hur radioaktivt sönderfall även kan orsaka kedjereaktioner i form av att andra atomer också sönderfaller pga. kollisioner med det radioaktiva sönderfallet. För att representera detta så kan även atomer runt (inte diagonalt) den sönderfallande atomen också sönderfalla. Modellen har uttryckts så generellt som möjligt i figur 3.2.1 nedan. Precis som i kösimulationen så skiljer sig flödet i tidsstyrningen och händelsestyrningen markant. I den tidsstyrda implementationen följer man modellen rätt exakt med tillägget att man ökar tidssteget efter varje iteration (från "*har en atom sönderfallit*" till "*är observationstiden slut*"-blocken). I den händelsestyrda simulationen däremot så inför man direkt efter start-blocket alla sönderfall som "burst"-händelser. De kedjereaktioner som kan orsakas av "burst"-händelser tolkas som "chain"-händelser.



Figur 3.2.1: Flödesdiagram som illustrerar perkolationsmodellen.

3.2.2 Implementation

Datastrukturen som håller reda på atomerna är en heltalsmatris. Varje heltal motsvarar en atom och värdet på heltalet avgör om atomen är öppen, stängd eller del av stigen som hittas efter simulationen. Denna *atommatris* har både den tidsstyrda simuleringen och den händelsestyrda simuleringen gemensamt. Implementationen av perkolationssimulationen har en liknande klass-struktur som kösimulationen. En överblick av strukturen i perkolationssimulationen ges i figur 3.2.2. I denna figur kan man också se likheten med kösimulationens implementation, som visas i figur 3.1.2. De klasser med samma namn (*Sim*, *StatisticsSimulation*, *Event* och *EventComparator*) som i kösimulationen uppfyller samma syfte. De nytillkomna klasserna (*Queue*, *ListNode*, *Path*, *AtomRender*, *Atom*, *ExitListener*) använts i ändamålet att grafiskt rendera atommatrisen (och en stig genom den, om det existerar).



Figur 3.2.2: UML-klassdiagram som visar strukturen i implementationen av perkolationssimulationen. Av samma anledningar som i kösimulationens UML-diagram (figur 3.1.2) har vissa metoder utelämnats.

3.2.2.1 Atom

Atom är en mindre klass som representerar en atom grafiskt. *Ellipse2D.Double* klassen används för att rendera atomen som en cirkel, medan *Color* variabeln *col* är färgen som representerar en atoms *öppenhet* (se avsnitt 2.4). De tre färger som används är ljusgrå, blå och svart. Svart är stängd, ljusgrå är öppen och blå innebär att atomen är med i den funna stigen. *Atom*-klassen används i *AtomRender* som variabeln atoms, som är en *ArrayLis*t med *Atom* som element. Hela denna lista måste gås igenom när bilden ska renderas i metoden *paintComponent* i *AtomRender.*

3.2.2.2 AtomRender

Uppgiften som denna klass har är att grafiskt rendera atommatrisen. För att kunna göra detta användes Java Swing [8] biblioteket. De centrala metoderna i denna klass är *drawGrid*, *paintComponent* och *createComponent*. Metoden *drawGrid* är det som anropas för att rita upp en angiven atommatris och metoden anropar *createComponent*, som konverterar atommatrisen från int[][] datatypen till *ArrayList<Atom>*. Det *drawGrid* också gör är att den sätter upp alla nödvändiga komponenter som behövs för att rendera ett fönster. Till sist så anropas *paintComponent* varje gång fönstret måste uppdateras för att rendera *ArrayList<Atom>*.

3.2.2.3 ExitListener

ExitListener är en subklass till JavaSwing klassen *WindowListener*. När man stänger något av fönstren (dvs. de som visar grafiken) så körs koden i *ExitListener*. Det koden gör är att stänga ner programmet genom att göra anropet *System.exit*.

3.2.2.4 Queue

Queue är en FIFO-kö anpassad för att kunna hantera generella objekt. Den används i *Path* klassen för BFS:en, som hittar en stig från botten till toppen av atommatrisen.

3.2.2.5 ListNode

Varje element i *Queue* är en *ListNode*. Syftet med *ListNode* är dels att innehålla det objekt som är innehållet för noden samt en pekare för nästa element i *Queue*-instansen.

3.2.2.6 Path

Path utnyttjas för att hålla reda på en stig genom atommatrisen när breddenförstsökningen letar efter en stig igenom materialet. Varje *Path*-instans har en pekare till föregående *Path*, dvs. man går baklänges genom stigen. En *Path*-instans innehåller även koordinaterna för motsvarande atom i atommatrisen.

3.2.2.7 Event

Event är i princip likadan som i kösimulationen (se avsnitt 3.1.2.1) med tillägget att koordinaterna för atomen i atommatrisen också sparas i en *Event*-instans.

3.2.2.8 EventComparator

Används i *PriorityQueue<Event>* variabeln i *Sim*-klassen och fungerar precis som i kösimulationen (se avsnitt 3.1.2.3).

3.2.2.9 Sim

Såsom i kösimulationen befinner sig implementationerna av händelsestyrd simulering och tidsstyrd simulering i denna klass. För att köra simulationerna anropar man metoderna *EventSim* samt *TimeSim*. Inparametrarna är samma till båda, bortsett från tidssteget. Dessa parametrar är: tidssteget Δt, spridningsfrekvensen *burstRate* (t. ex 60 innebär att det är 1/60 sannolikhet att en atom orsakar en annan atom att sönderfalla), tidsspannet *stopTime*, halveringstiden *halfLife*, *xSize* (antal atomer i bredd, antal kolumner) och *ySize*(antal atomer i höjd, antal rader). Precis som i kösimulationen måste vissa parametrar använda samma tidsenhet, dvs. tidssteget, tidsspannet och halveringstiden.

TimeSim

Pseudokod 3.2.1 beskriver hur *TimeSim* fungerar. Simuleringen som *TimeSim* genomför startar vid tiden *t*=0 och körs tills *t* överskrider tidsspannet, alltså när *t >= stopTime*. Vid varje tidssteg undersöker man varje atom. Baserat på halveringstiden så avgörs det om atomen ska sönderfalla eller inte. Om den sönderfaller så behandlas de möjliga kedjereaktionerna i de omkringliggande atomerna. Kedjereaktionerna hanteras i *burst*-metoden (se avsnitt 3.2.2.8). När programmet gått igenom alla atomer uppdateras tiden med Δt och samma procedur upprepas igen.

```
while ( t <= stopTime ) {
    for each atom in Grid {
        if( timeToBurst( atom ) for current atom ) {
            burst( current atom )
        }
    }
    t +=  Δt
}
```

Pseudokod 3.2.1: Huvud-loopen för den tidsstyrda perkolationssimulationen.

Funktionen *timeToBurst* fungerar ungefär som funktionen för att beräkna ankomsttiden i kösimulationen, som förklarades i pseudokod 3.1.2. Nedan i pseudokod 3.2.2 visas det hur *timeToBurst* kan programmeras.

```
boolean timeToBurst( atom ) {
    upperRoof = Round( 1/ ( Δt *   (ln 2 / halfLife) ))
    if( atom is intact and RandomInteger( [0, upperRoof) ) == 0 ) {
        return true
    }
    return false
}
```

Pseudokod 3.2.2: Sannolikhetsberäkningen som avgör om en atom ska sönderfalla eller inte.

Den enda skillnaden från pseudokod 3.1.2 är att atomen måste vara intakt för att den ska kunna sprängas, vilket innebär att antalet möjliga sönderfall i en simulation är begränsat.

EventSim
Den händelsestyrda simuleringen börjar med att bestämma när varje atom sönderfaller oberoende av kedjereaktioner. Alltså det naturliga sönderfallet som är baserat på halveringstiden angiven vid programanropet.

Till skillnad från kömodellen där endast en kund kunde komma in i taget så kan atomer i perkolationsmodellen sönderfalla vid samma tidpunkt. På grund av detta så blev införandet av så kallade "burst"-händelser annorlunda, som beskrivet i pseudokod 3.2.3:

```
for each atom (i, j) in Grid {
    t = exp(halfLife)
    if( t > stopTime ) continue;
    Add Event = ( "burst", t ,  atom.i, atom.j ) to priorityQue
}
```

Pseudokod 3.2.3: Denna kod visar hur sönderfall som beror på halveringstidsparametern införs i den händelsestyrda simuleringen.

Atomernas naturliga sönderfall (dvs. utan kedjereaktioner) är oberoende av varandra, till skillnad från kunderna som alla ankommer med en viss frekvens. Så istället för att beräkna nästa händelses tidpunkt i förhållande till föregående händelse så införs nu alla händelser över hela tidsintervallet, oberoende av varandra. Metoden *exp* är densamme som i kösimulationen (se avsnitt 3.1.2.4).

Händelserna bearbetas på samma sätt kösimulationen som beskrivet i avsnitt 3.1.2.4 När en händelse plockas ur prioritetskön så undersöker man vilken händelse det är och tillståndet för atomerna uppdateras med metoden *burstEvent.* Metoden *burstEvent* hanterar kedjereaktioner och lägger till "burst" samt "chain"-händelser till prioritetskön om sådana inträffar. Metoderna finns i *Grid*-klassen (se avsnitt 3.2.2.11).

```
while( priorityQue not empty && priorityQue.peek().time < stopTime ) {
    event =  priorityQue.pop()
    atom = event.atom
    ( i , j ) = atom.position
    t = event.time

    if ( event = "burst" and atom is intact ) {
        burstEvent( i , j , priorityQue , t );
    } else if ( event = "chain" and atom is intact ){
        chainreactions++
        burstEvent(i, j, priorityQue, t);
    }
}
```

Pseudokod 3.2.4: Huvud-loopen i den händelsestyrda simulationen.

Innan *burstEvent*-metoden anropas måste det undersökas om atomen redan har sönderfallit. Det är på grund av att en atom kan sönderfalla på två sätt. Antingen genom en kedjereaktion eller från "naturligt" sönderfall.

Om en "burst"-händelse för en atoms "naturliga" sönderfall redan är inlagt in prioritetskön och den sönderfaller innan, som resultatet av en kedjereaktion, så blir den senare "burst"-händelsen onödig. Den tar då upp onödig plats i prioritetskön, vilket påverkar prestandan (se avsnitt 4.1.2.3 för mer om detta).

3.2.2.10 StatisticsSimulation

StatistsicsSimulation skriver ut relevant statistik om simulationskörningarna och mäter dess exekveringstid, såsom i kösimulationen (se avsnitt 3.1.2.5). Inparamterarna för perkolationssimulationen är tidssteget, spridningsfrekvensen, tidsspannet, halveringstiden, antal atomer i bredd, antal atomer i höjd och antalet körningar. Det som skiljer denna *StatistiscsSimulation* från kösimulationens *StatisticsSimulation* mest är att den sista körningen i denna *StatistiscsSimulation* renderas grafiskt. Även i denna klass finns en metod *writeResult* som använder sig av JExcelAPI [9] för att skriva ut utdatan till en excelfil.

3.2.2.11 Grid

Grid används för att representera atomstrukturen och utföra operationer på den. Atomernas tillstånd lagras i heltalsmatrisen *atoms* där varje heltal motsvarar en atom. Om en atom är 1 är den intakt och 0 om den har sönderfallit. Anledningen till att det är en heltalsmatris och inte bara en boolsk matris är för att kunna markera själva stigen som motsvarar vätska som rinner igenom materialet. Detta görs senare med metoden *colorPath*. En atom som är med i stigen har värdet 7.

De tre metoder som utgör funktionaliteten i *Grid* är *burst*, *burstEvent* och *findPath*. Metoderna *burst* och *burstEvent* gör i princip samma sak, vilket är att de hanterar själva sönderfallet av en atom. När *burst* (eller *burstEvent*) anropas så sätts den nuvarande atomen till att ha sönderfallit och sedan undersöks det om detta sönderfall orsakade en kedjereaktion. Detta beror då på spridningssannolikheten *burstRate* som angavs när man startade *StatistiscSimulation*. I pseudokod 3.2.5 samt 3.2.6 kan man enkelt observera att det som skiljer *burst* och *burstEvent* metoderna är dels inparametrarna men även hur kedjereaktioner hanteras. För varje kedjereaktion så anropar *burst*-metoden sig rekursivt medan *burstEvent* lägger till en ny "chain"-händelse i prioritetskön för varje kedjereaktion. Om man har en begränsad stack samt en *burstRate* nära 1 kan *burst*-metoden krascha av att stacken överskrids pga. de rekursiva anropen.

```
public void burst( int i, int j ){
    set atom at position i, j in atoms to 0.

    if(atom above bursts){
            burst( i-1,  j );
    }
    if(atom below bursts ){
            burst( i+1,  j );
    }
    if(atom to the left bursts){
            burst( i,  j-1 );
    }
    if atom to the right bursts){
            burst( i,  j+1 );
    }
}
```

Pseudokod 3.2.5: Den tidsstyrda simuleringens hantering av ett sönderfall.

```
public void burstEvent( int i, int j, PriorityQueue priorityQue, float time  ){
    set atom at position i, j in atoms to 0.

   if(atom above bursts){
            Add Event = ( "chain", time ,  i-1, j ) to priorityQue
    }
    if(atom below bursts ){
            Add Event = ( "chain", time ,  i+1, j ) to priorityQue
    }
    if(atom to the left bursts){
            Add Event = ( "chain", time ,  i, j-1 ) to priorityQue
    }
    if (atom to the right bursts){
            Add Event = ( "chain", time ,  i, j+1 ) to priorityQue
    }
}
```

Pseudokod 3.2.6: I den händelsestyrda simuleringen hanteras ett sönderfall lite annorlunda än i tidsstyrning. Den främsta skillnaden är att istället för ett rekursivt anrop till sig självt så skapas istället en ny "chain"-händelse.

Det som *findPath* metoden gör är att den letar sig igenom atommatrisen för att hitta en stig som en "vätska" skulle kunna rinna igenom. För att hitta den kortaste stigen genom atomstrukturen används en enkel bredden-först-sökning (BFS). Om en stig hittas markeras den genom att sätta de positioner i heltalsmatrisen med i stigen till 7. Denna markering används i *AtomRender*-klassen.

3.2.3 Körexempel

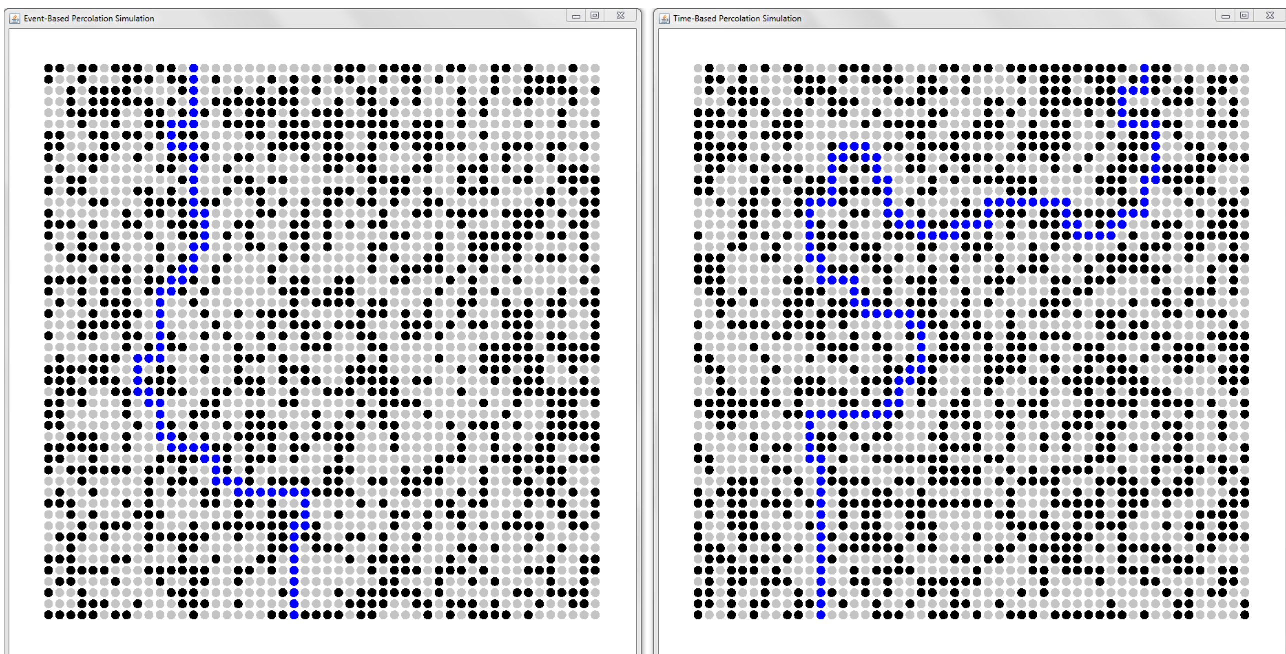
I figur 3.2.3 nedan visas utdatan från en körning av perkolationssimulationen. Utdatan består av två delar, *TimeSims* resultat och *EventSims* resultat. Värdena i utskriften beskriver dels körningstiden (”Total/Avg time”), antalet funna stigar i körningarna (”Paths through/runs”) och antalet sönderfallna atomer (”Total/Avg bursted atoms”, ”Total/Avg chain reactions”, ”Total/Avg decompositions”). Alla medelvärden är det totala värdet delat på antalet körningar.

```
Console
<terminated> PercSim demonstration [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (7 apr 2012 12:35:18)
Timesim:
Total time: 93.4167709350586 ms, Avg time: 9.34167709350586 ms, Paths through/runs: 4/10
Total bursted atoms: 14227.0, Total chain reactions: 3762.0, Total decompositions: 10465.0
Avg bursted atoms: 1422.7, Avg chain reactions: 376.2, Avg decompositions: 1046.5

Eventsim:
Total time: 11.447380065917969 ms, Avg time: 1.144738006591797 ms, Paths through/runs: 3/10
Total bursted atoms: 14241.0, Total chain reactions: 3827.0, Total decompositions: 10414.0
Avg bursted atoms: 1424.1, Avg chain reactions: 382.7, Avg decompositions: 1041.4
```

Figur 3.2.3: Utdata från en körning i Eclipse med inparametrarna “1 10 70 77.27 50 50 10”

Varje körning har också en grafisk representation som *AtomRender* beräknar. Den motsvarande grafen för simulationen i i figur 3.2.3 visas i figur 3.2.4 nedan.



Figur 3.2.4: Grafisk representation av körningen i figur 3.2.3. Enbart den tionde körningen renderades eftersom eftersom det var totalt tio körningar.

4 Evaluering av implementationerna

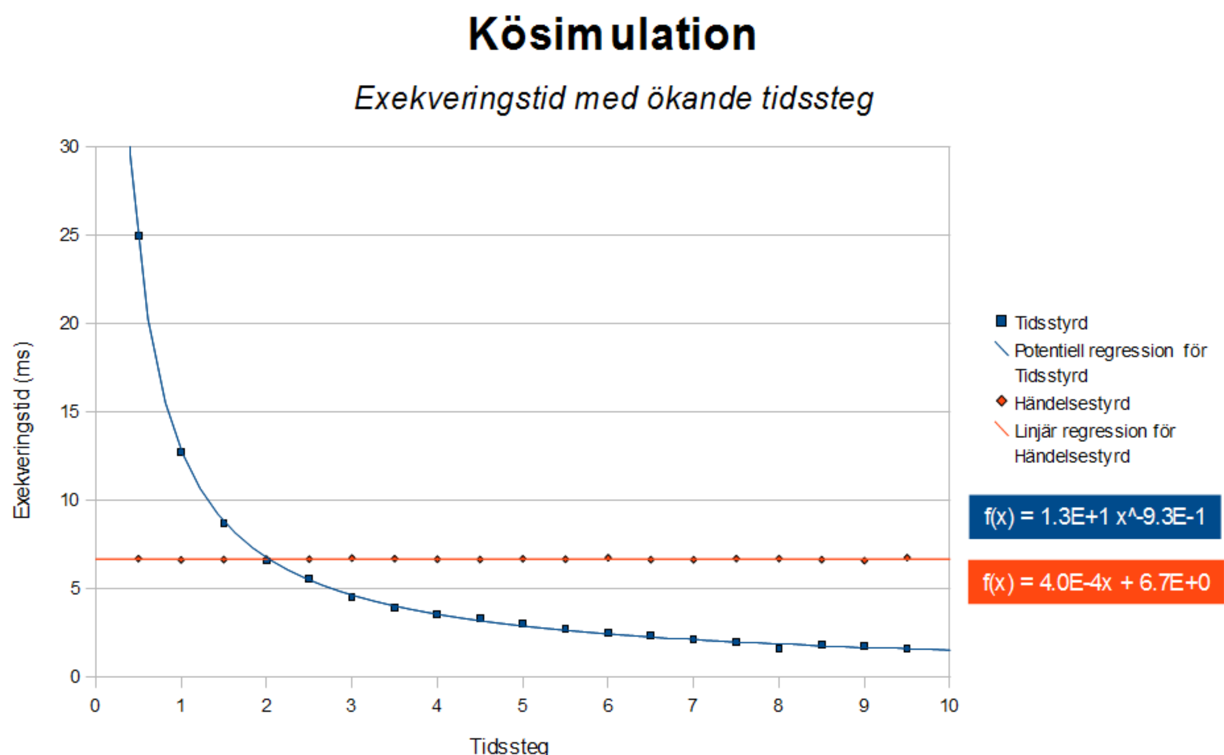
I detta kapitel så evalueras alla de fyra implementationerna som gjorts, dvs. tidsstyrd och händelsestyrd kösimulation samt perkolationssimulation. Analysen har utförts på de egenskaperna listade i problemformuleringen (se avsnitt 1.2). Kapitlet är uppdelat i tre avsnitt. De två första avsnitten behandlar exekveringstiden och det statistiska resultatet av kö- samt perkolationssimulationen. Det tredje avsnittet går in på implementationssvårigheten av båda simulationsparadigmerna. Vidare så har alla simuleringar körts på en och samma dator. Datorn hade operativsystemet Ubuntu och var utrustad med en Intel Q9552 processor.

4.1 Kösimulation

4.1.1 Analys av exekveringstid

För att utforska exekveringstiden av implementationerna har inparametrarna varierats på olika sätt. De följande sektionerna kommer att behandla enstaka inparametrar i taget.

4.1.1.1 Tidssteg

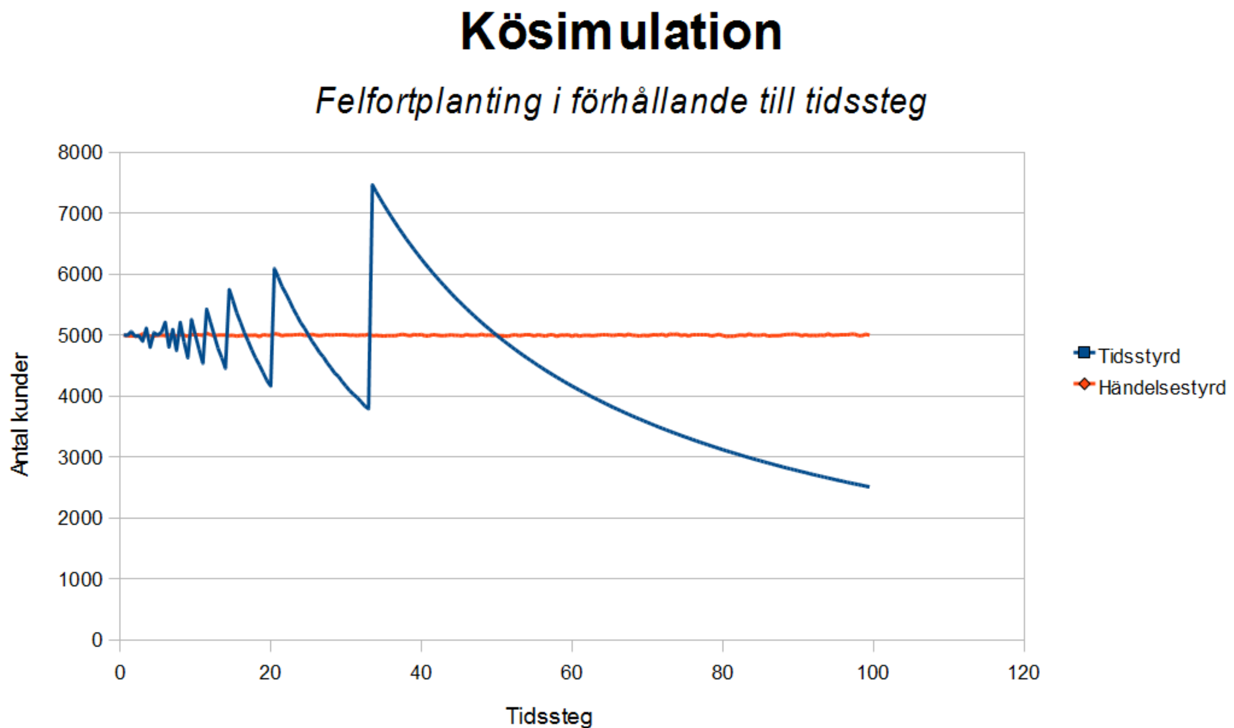


Figur 4.1.1: Varje punkt i grafen är 50 körningar med inparametrarna: betjäningstid: 10, tidsspann: 250000 och ankomstfrekvens: 50. Tidssteget börjar på 0.5 och ökar för varje punkt från vänster med 0.5.

Utifrån grafen i figur 4.1.1 ser vi att den tidsstyrda simuleringen tar längre tid när tidssteget är litet men också att den är snabbare än den händelsestyrda simuleringen när tidssteget är tillräckligt stort. Det är inte särskilt överraskande eftersom antalet varv som tidssimuleringen ligger i

simuleringsloopen är tidsspannet delat på tidssteget. En fördubbling av tidssteget medför alltså ungefär en halvering av exekveringstiden, vilket syns tydligt i grafen. Den händelsestyrda simuleringens exekveringstid påverkas självklart inte eftersom den inte använder tidsteget.

Nu är frågan vilket tidssteg som bör väljas och om det finns några positiva eller negativa effekter av att variera tidssteget förutom att exekveringstiden påverkas. I figur 4.1.2 illustreras dess effekter tydligt.



Figur 4.1.2: Den här grafen har samma inparametrar och antalet körningar per punkt som i figur 4.1.1. Dock så används istället värdet för antalet ankomna kunder (medelvärde över 50 körningar) på y-axeln. Totalt utgörs grafen av 199 punkter.

Med de angivna parametrarna i figur 4.1.2 bör antalet kunder ligga runt 5000 ($\frac{1}{50}$ kund per tidsenhet ger $\frac{1}{50} \cdot 250000 = 5000$). Med händelsestyrd simulering som referens syns det tydligt att tidsstyrningen divergerar från det förväntade värdet. Även om det är frestande att använda ett stort tidssteg för att få en snabbare exekveringstid så verkar det inte vara en bra idé. Även en liten ökning av tidssteget gör att simulationsresultatet påverkas. I figuren kan vi se att felet ökar med högre tidssteg samtidigt som det pendlar mellan positivt och negativt tecken.

Anledningen till grafens beteende är att funktionen som beräknar tidsstyrningens kundankomster (se pseudokod 3.1.2) inte fungerar bra när tidssteget är för stort. Att antalet kunder sjunker när tidssteget ökas beror på att chansen för att en kund ankommer varje tidssteg inte förändras under intervallet. Men antalet tillfällen, antalet tidssteg, som en kund kan ankomma på minskar. Den plötsliga skiftningen från för få till för många kunder beror på att den övre gränsen på intervallet som ett godtyckligt värde tas ifrån går från att avrundas uppåt till att avrundas neråt. När tidssteget

är lågt och chansen för att en kund ankommer varje tidssteg går från t.ex. $\frac{1}{100}$ till $\frac{1}{99}$ märks ingen stor skillnad. Däremot när tidssteget är stort och chansen går från exempelvis ½ till 1 blir den resulterande skiftningen väldigt stor.

För att förtydliga problemet beskrivet ovan måste vi minnas hur pseudokod 3.1.2 slumpar ett tal inom ett intervall beroende på *arrivalRate* och tidsteget Δt. Om det här ska fungera måste den övre gränsen vara ett heltal. I den utförda implementationen motsvarar detta Javas egna *Math.Round*. När tidssteget är 21 kommer beräkningen för det övre intervallet bli följande:

$$Math.Round\left(\frac{1}{\Delta t\cdot\frac{1}{arrivalRate}}\right)=Math.Round\left(\frac{1}{21\cdot\frac{1}{50}}\right)\approx Math.Round\,(2.38095239)=2$$

Detta motsvarar att en kund anländer med sannolikheten $\frac{1}{2}$. Då simulationen tar $\frac{250000}{21}$ tidssteg så ankommer det alltså i snitt $\frac{250000}{21}\cdot\frac{1}{2}=5952$ kunder. Om tidssteget ökas till 22 kommer chansen att en kund ankommer inte förändras på grund av avrundningen som visas nedan:

$$Math.Round\left(\frac{1}{22\cdot\frac{1}{50}}\right)\approx Math.Round\,(2.27272727)=2$$

Dock så tas färre tidssteg eftersom tidssteget har ökats. Totalt simuleras $\frac{250000}{22}=11363$ steg och i snitt ankommer det $11363\cdot\frac{1}{2}=5681$ stycken kunder. På samma sätt fortsätter antalet kunder att minska när tidssteget ökas tills tidssteget är 34. Då kommer en kund alltid anlända, eftersom beräkningen för den övre gränsen resulterar i 1:

$$Math.Round\left(\frac{1}{34\cdot\frac{1}{50}}\right)\approx Math.Round\,(1.47058824)=1$$

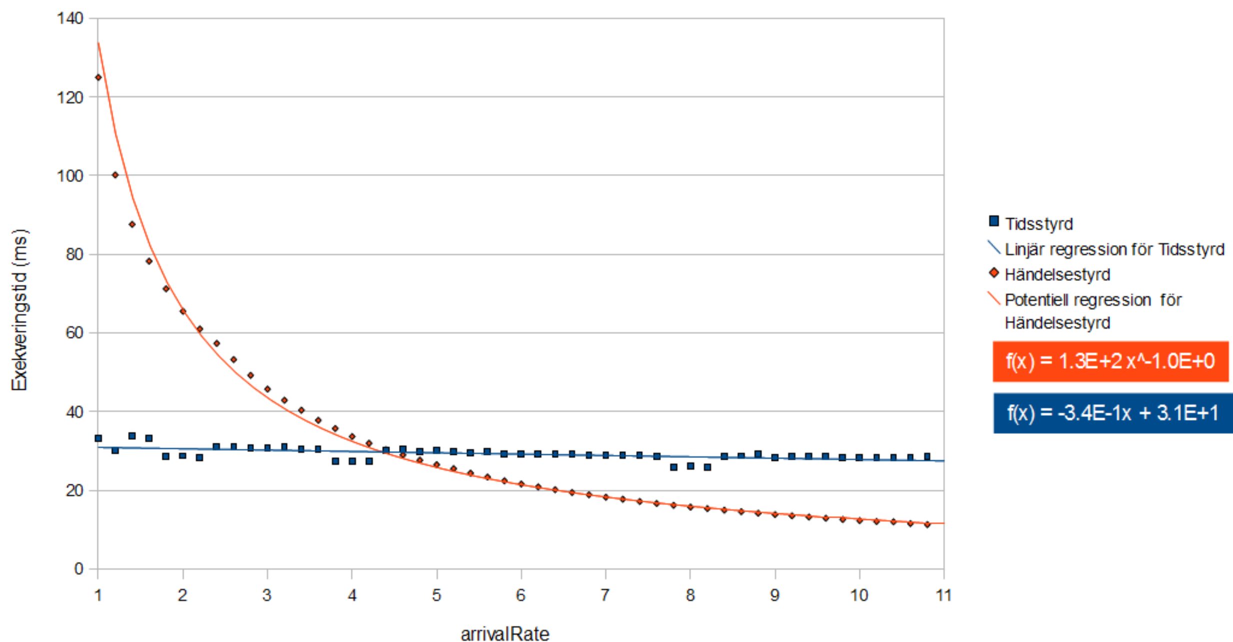
Därför ankommer det nu i snitt $\frac{250000}{34}=7352$ stycken kunder. Detta förklarar varför antalet kunder går från att vara färre till fler än det förväntade medelvärdet.

Ett ytterliggare skäl till att det kan bli fel när ett större tidssteg används är att en kunds betjäningstid kan överskridas. Betrakta följande exempel: en kund ankommer vid tiden t=0. Med en betjäningstid på 11 kommer kunden, om allt är som det ska, ut ur systemet vid tid t=11. Fast om tidssteget är 2 undersöks det vid t=2,4,6,8,10 och så vidare om kunden först i kön har blivit serverad tillräckligt länge. Vid t<=10 har kunden inte stått först i kön tillräckligt länge för att plockas ut. Dock så har hon gjort det när t=12, och plockas därefter ut. Men kundens totala kötid är större än vad den egentligen borde vara, 12 istället för 11. Dessutom kommer kunden härnäst i kön att ställa sig först i kön en tidsenhet senare än vad som skulle vara fallet med ett mindre tidssteg.

Under en längre simulation kan dessa små avvikelser adderas ihop till ett större fel. Det som påverkas mest av ett för stort tidssteg är antalet kunder, som vid vissa fall kan påverkas drastiskt. Kundantalet påverkar i sin tur hela simulationsresultatet vilket gör att simulationsresultatet inte går att lita på om ett stort tidssteg har använts.

4.1.1.2 Parametern arrivalRate

Det som undersöks här är hur exekveringstiden för simulationerna påverkas av att variera kundernas ankomstfrekvens. Det som visas är hur de två simulationsteknikerna presterar när det ankommer väldigt många respektive väldigt få kunder.

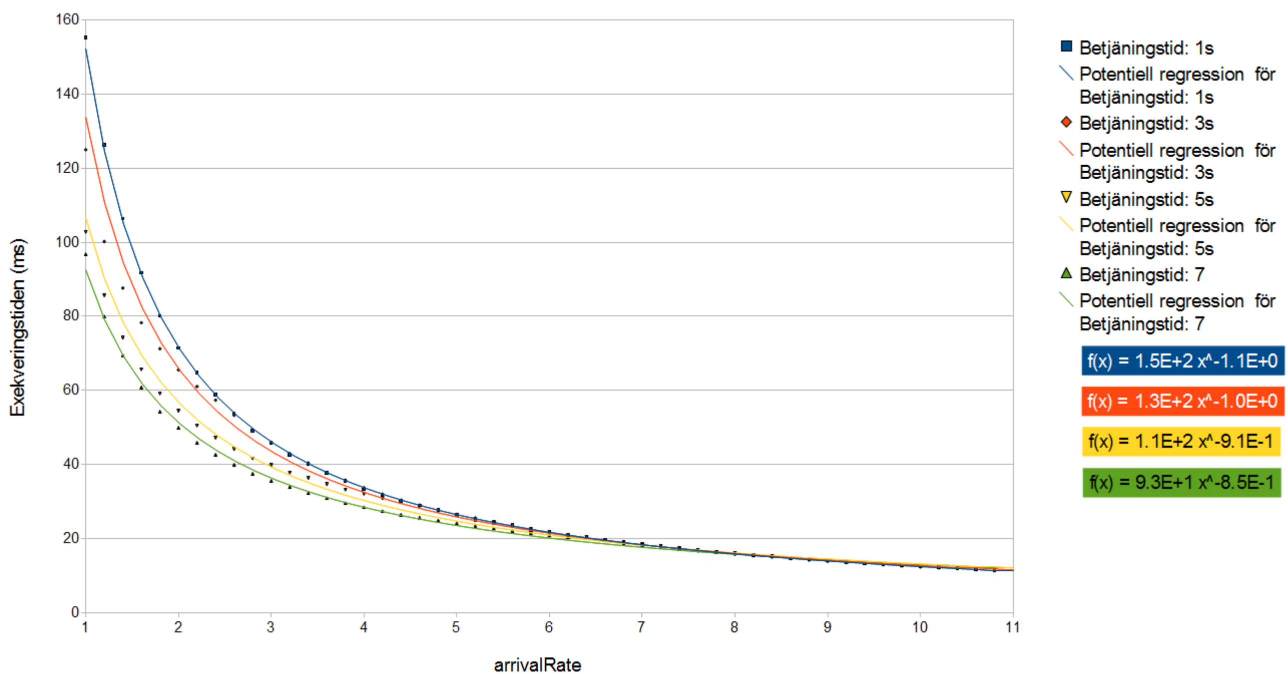


Figur 4.1.3: Graf som avbildar resultatet av 50 kösimulationener. Inparamterarna i varje körning var tidssteg: 0.5, betjäningstid: 2.5, tidsspann: 250000 och ankomstfrekvens: 1/x kunder per sekund (x = 1 + 0.5i, för varje punkt i från vänster till höger). Antal körningar var 50 per punkt.

I figur 4.1.3 kan vi se att den händelsestyrda simulationens exekveringstid beror stark på kundernas ankomstfrekvens medan den tidsstyrda knappt påverkas. När ankomstfrekvensen är väldigt hög (från 1 till $\frac{1}{2.5}$ kunder per tidsenhet) så tar den händelsestyrda simuleringen mycket längre tid jämfört med den tidsstyrda. Sedan, allt eftersom ankomstfrekvensen minskar, så tar den händelsestyrda simulationen över som den snabbaste simulationsparadigmen. Dock så planar den händelsestyrda kurvan ut efter x=10 och man tjänar inte särskilt mycket i exekveringstid på att sänka ankomstfrekvensen ytterligare efter det. Detta fenomen kan förklaras med att den händelsestyrda simulationens exekveringstid i största hand beror på hur många händelser som sker under simulationens gång (då operationer på prioritetskön tar längre tid ju fler händelser som finns i den). Ju högre ankomstfrekvensen är, desto fler händelser inträffar. Den tidsstyrda

simulationen däremot simulerar varje tidssteg vad som händer, och skillnaden i exekveringstid om det händer eller inte händer något under ett tidssteg verkar vara försumbart, eftersom trendlinjen för tidssimulationen är så gott som horisontell.

Om man även minskar betjäningstiden så belastas händelsekön ytterligare eftersom det skapas fler “depature”-händelser, vilket går att se i figur 4.1.4. Det grafen verkar antyda är att exekveringstiden, fast med olika betjäningstider, sammanfaller vid högre *arrivalRate* (dvs. mindre ankomstfrekvens). Naturligtvis måste det vara så att betjäningstidens effekt är proportionell mot *arrivalRate.* Vid låga värde på *arrivalRate* blir det många ankomster och för varje ankomst så skapas en "depature"-händelse. Vid högre värden på *arrivalRate* så blir det färre ankomster och därmed också färre "depature"-händelser. Effekten som betjäningstiden har på exekveringstiden verkar i princip försvinna vid tillräckligt stora värden på *arrivalRate.*

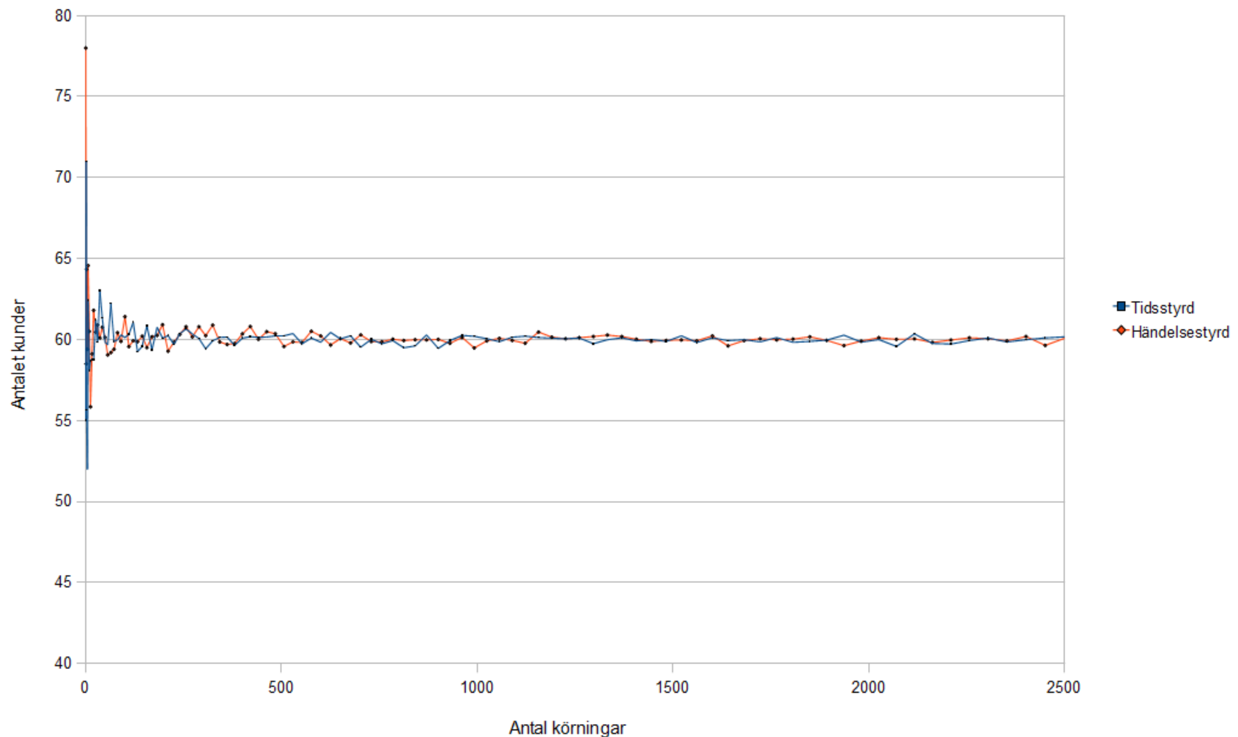


Figur 4.1.4: Samma indata som i figur 4.1.3. Alla fyra linjer behandlar den händelsestyrda kösimulationen.

I en situationen där man vill simulera ett kösystem med hög ankomstfrekvens bör man välja den tidsstyrda implementationen, eftersom den inte berördes av *arrivalRate.*

4.1.2 Statistisk analys

Om ankomster av kunder har implementerats korrekt så bör antalet kunder som kommer in till systemet sammanfalla mot ett medelvärde motsvarande den *arrivalRate* given.



Figur 4.1.5: Antalet körningar per punkt ökar med 1 + i/2 för varje punkt i (från vänster). De andra inparametrarna för varje punkt är: tidssteg: 0.1, betjäningstid: 10, tidsspann: 3600, ankomstfrekvens: 60.

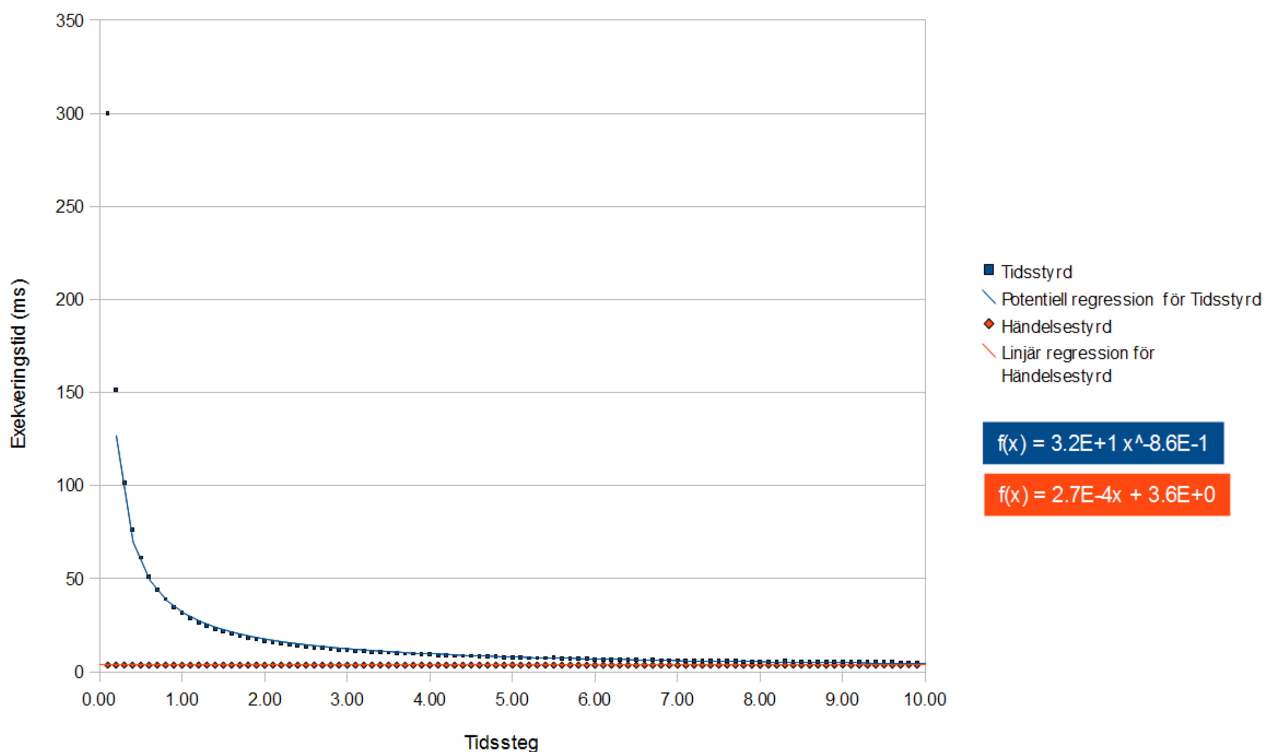
I figur 4.1.5 så tolkas tidsenheten som sekunder. Parametern *arrivalRate* är angiven som 60, vilket motsvarar 1 kund per minut. Om tidsspannet är 3600 (dvs. 60 minuter) så innebär det att antalet kunder rent statistiskt befinna sig runt 60. Detta syns väldigt tydligt i grafen, då medelvärdet av över 500 körningar inte avviker nämnvärt från 60 kunder/körning.

4.2 Perkolationssimulation

4.2.1 Analys av exekveringstid

På samma sätt som i avsnitt 4.1.1 kommer varje inparameter för perkolationssimulationen att iakttas och varieras för att undersöka dess påverkan på de olika implementationerna.

4.2.1.1 Tidssteg

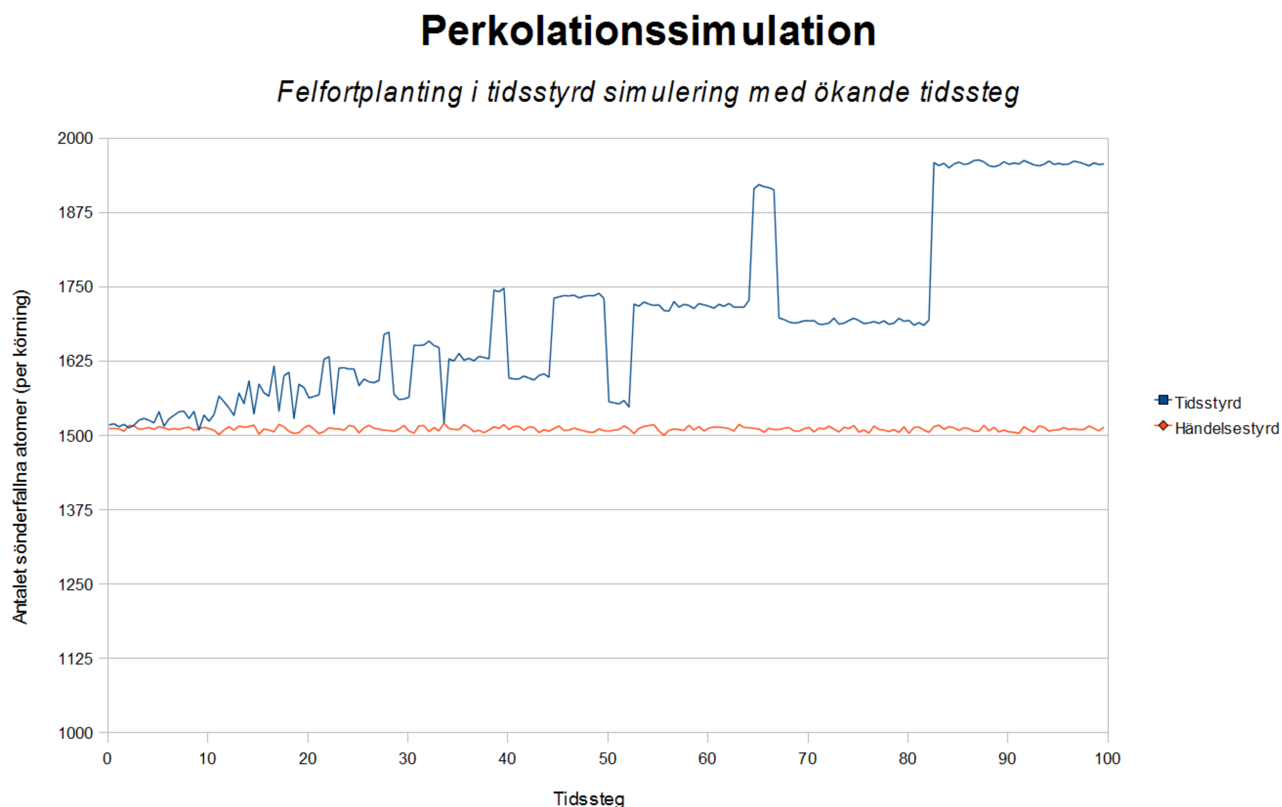


Figur 4.2.1: Varje markerad punkt är ett resultat från 50 körningar vara det är totalt 100 punkter. För varje punkt ökades tidssteget med 0.1, varav tidssteget börjar på 0.1. Den övriga indatan bestod av följande: burstRate: 10, antal atomer i bredd och höjdledd: 100.

Från figur 4.2.1 ser vi att händelsesimulering är snabbare än tidsimulering, för det mesta. Den tidsstyrda simuleringens exekveringstid närmar sig den händelsestyrda simuleringens exekveringstid när större och större tidssteg används.

För att förstå varför detta händer måste vi påminnas om hur tidssimuleringen utförs. För varje tidssteg så går programmet igenom alla atomer och undersöker om de sprängs vid det tidssteget. Om tidssteget fördubblas kommer alltså antalet beräkningar som avgör om atomerna sprängs halveras. Från dessa iakttagelser verkar det som att det är fördelaktigt att använda ett stort tidssteg vid tidssimulering. Men vad som inte syns i grafen är att simuleringens precision påverkas.

Detta kan man se i figur 4.2.2 nedan.



Figur 4.2.2: Graf som visar felfortplantningen för tidsstyrd simulation. Varje punkt i grafen är en av 200 körningar varav varje körning hade inparametrarna: burstRate: 10, tidsspann: 200, halveringstid: 200, antal atomer i höjd och bredd: 50 samt 50 körningar gemensamt. Tidssteget börjar på 0.1 och ökar med 0.5 för varje punkt (mot höger).

Från figur 4.2.2 är det tydligt att man inte bör använda ett stort tidssteg vid tidssimulering. Utifrån inparametrarna så bör ungefär 1500 atomer ha sönderfallit totalt. Av dessa så ska 1250 atomer ha sönderfallit från "naturligt" sönderfall medan en tiondel, dvs. 250, ha sönderfallit från kedjereaktioner. Genom att använda händelsestyrning som referens (då den ligger runt 1500) kan man se när tidsstyrning börjar avvika från det förväntade värdet. Det är endast precis i början, när tidssteget är mindre än 2, som resultatet av tidssimulationen blir någorlunda korrekt. Vid ett större tidssteg sönderfaller mer atomer än vad som är förväntat vilket gör hela simulationen inkorrekt.

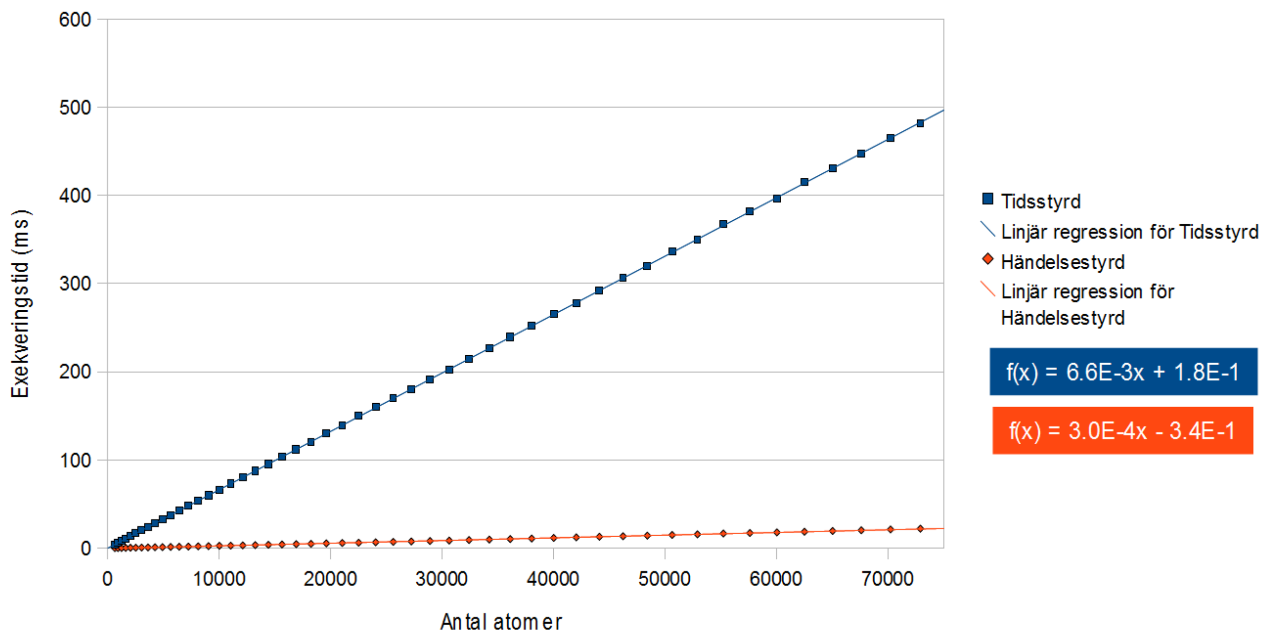
Att grafen ser ut som den gör beror till stor del på hur vi implementerat funktionen som beräknar om en atom sönderfallit. Det är alltså snarlikt det problem beskrivet i avsnitt 4.1.1.1, men inte riktigt. Graferna ser helt olika ut. Exakt vad som är annorlunda med den tidsstyrda perkolationssimulationen gick inte att hitta och förklara. Vad som dock går att konstatera är att med ökande tidssteg så ökas felet i den tidsstyrda simuleringen.

Enligt modellen kan en atom sönderfalla när som helst. Detta går att representera väldigt exakt med händelsesimulering men med tidsstyrd simulering så måste en kompromiss göras. Detta då ett väldigt litet tidssteg inte är praktiskt att använda på grund av den ökade exekveringstiden. Men, som vi kan se från figur 4.2.2, så är skillnaderna försumbara vid små tidssteg. Det är först när

tidssteget börjar bli stort som skillnaderna visar sig.

Säg att tidssteget är ett år. Då finns det bara ett tillfälle per år att atomen sönderfaller, jämfört med de 365 tillfällena när tidssteget är en dag. När ett stort tidssteg används finns det risk för att uppdateringar för en viss tid missas och samtidigt så blir antalet tillfällen som saker kan hända på färre. Det ger mindre precision och gör att simulationen överensstämmer sämre med modellen.

4.2.1.2 Antal atomer



Figur 4.2.3: Varje punkt är en exekvering av perkolationssimulationen med ökande antal atomer. Alla andra parametrar är konstanta (tidssteg: 0.5, tidsintervall: 72.27, burstRate: 0, halveringstid: 72.27, körningar: 50). För varje punkt mot höger ökar antalet atomer i bredd och höjd med 5, dvs. $(25+5x)^2=225^2+225x+25x^2$ *för punkten x.*

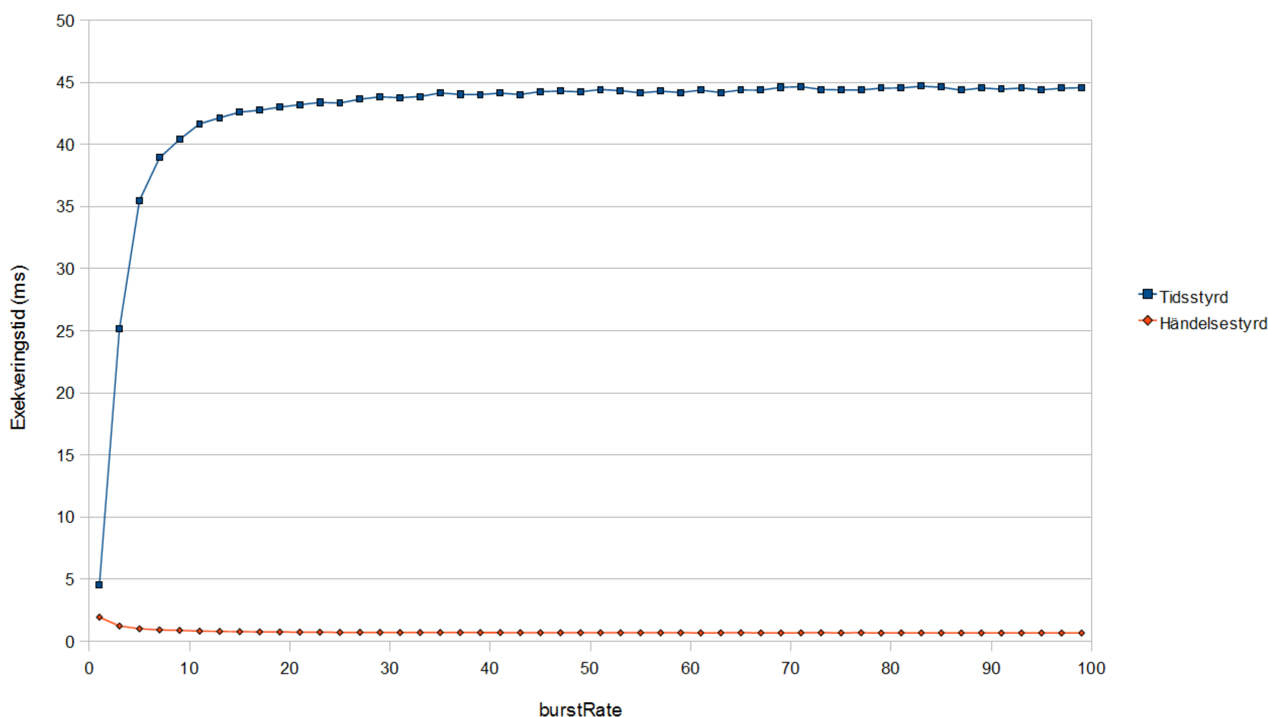
I figur 4.2.3 verkar det som att båda simulationsparadigmernas resulterande exekveringstid verkar skala linjärt mot antalet atomer i modellen. Dock så märks det tydligt att den händelsestyrda simuleringens exekveringstid ökar väsentligt mycket mindre än den tidsstyrda med antalet atomer. En snabb jämförelse mellan trendlinjernas koefficienter visar att den tidsstyrda simuleringens exekveringstid ökar *20* gånger snabbare än den händelsestyrda simuleringen.

Det är ingen överraskning att tidssimuleringen skalar så dåligt med antalet atomer jämfört med den händelsestyrda simuleringen. Om tio atomer läggs till i modellen gör händelsesimuleringen tio extra beräkningar, en beräkning för varje atom för att räkna ut vid vilken tid den sönderfaller. Dessutom tar varje efterföljande operation på prioritetskön lite längre tid. Tidssimuleringen gör däremot $\frac{stopTime}{tidssteget}$ extra beräkningar. I det här fallet blir det $\frac{72.27}{0.5}\approx 145$ stycken extra beräkningar för varje atom. Totalt 1450 stycken fler beräkningar om antalet atomer ökas med tio. Även om

atomerna sönderfaller tidigt så måste tidssimulationen vid varje tidssteg undersöka om atomerna redan har sprängts, vilket i och för sig inte är lika kostsamt som att anropa slumpfunktionen men som ändå tar tid.

4.2.1.3 Parametern burstRate

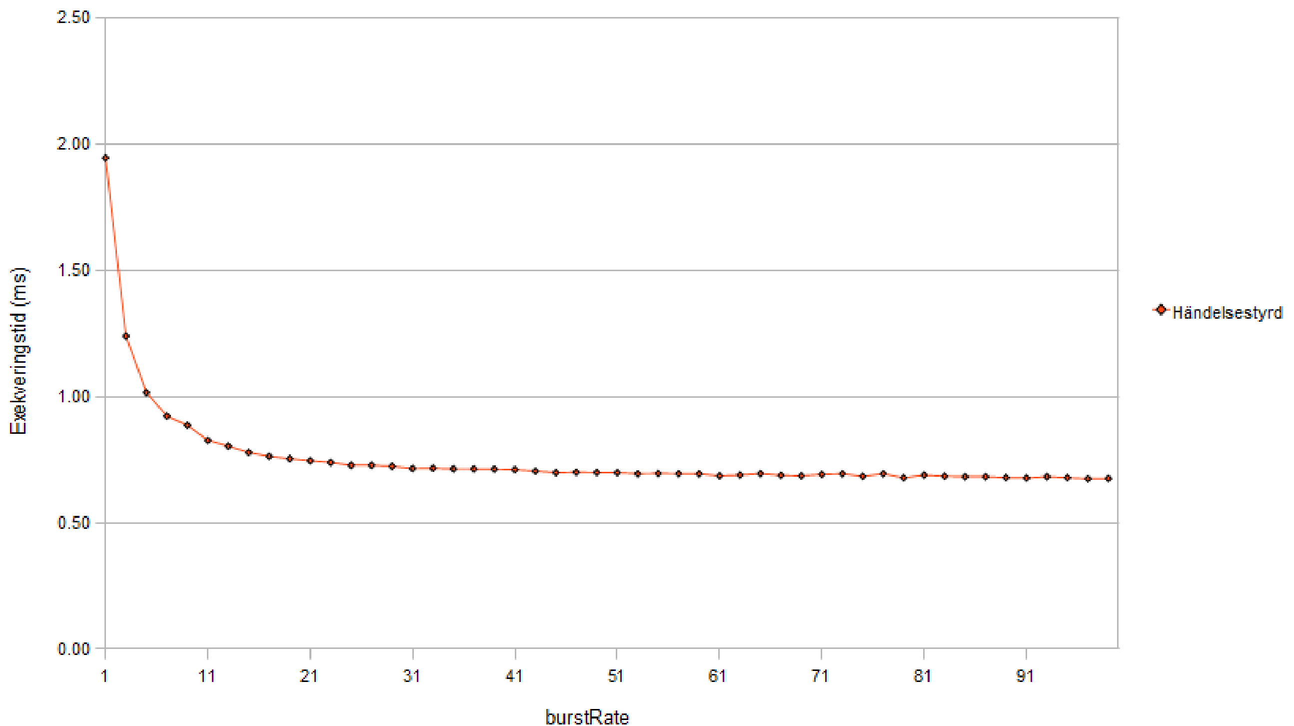
Det som förväntades av att variera *burstRate* var att den händelsestyrda simuleringen skulle belastas likt kösimulationen när man ökade ankomstfrekvensen samt minskade betjäningstiden. Vad som faktiskt hände vara att bara tidsstyrningen påverkades märkbart, som grafen i figur 4.2.4 visar.



Figur 4.2.4: Varje körning för varje punkt i grafen har inparametrarna: tidssteg: 0.5, tidsspann: 200, halveringstid: 200, atombredd och atomhöjd: 50 samt 50 körningar. Parametern för kedjereaktionshalten, burstRate, börjar på 1 och ökar med 2 för varje punkt.

Anledningen till att tidsstyrningen har lägre exekveringstid för lågt värde på *burstRate* är för att många atomer elimineras tidigt från atommatrisen via kedjereaktioner. Detta innebär att man inte behöver beräkna om en atom sönderfaller lika många gånger för varje tidssteg, vilket minskar körtiden. Sedan så verkar det som att exekveringstiden inte ökar mer än till 45 ms, vilket antyder att *burstRate* inte har någon påverkan på exekveringstiden efter värdet 35. Värsta fallet av körningen är att man blir tvungen att iterera genom alla atomer för varje tidssteg, vilket verkar vara fallet efter värdet 35 på *burstRate*. Vid sådana värden för *burstRate* verkar som att chansen för en kedjereaktion är så liten att antalet kedjereaktioner inte påverkar exekveringstiden nämnvärt. Då det bara finns ett begränsat antal atomer så kommer antalet händelser också vara begränsat. Även om man har en låg *burstRate* som resulterar i många sönderfall så är dess påverkan minimal.

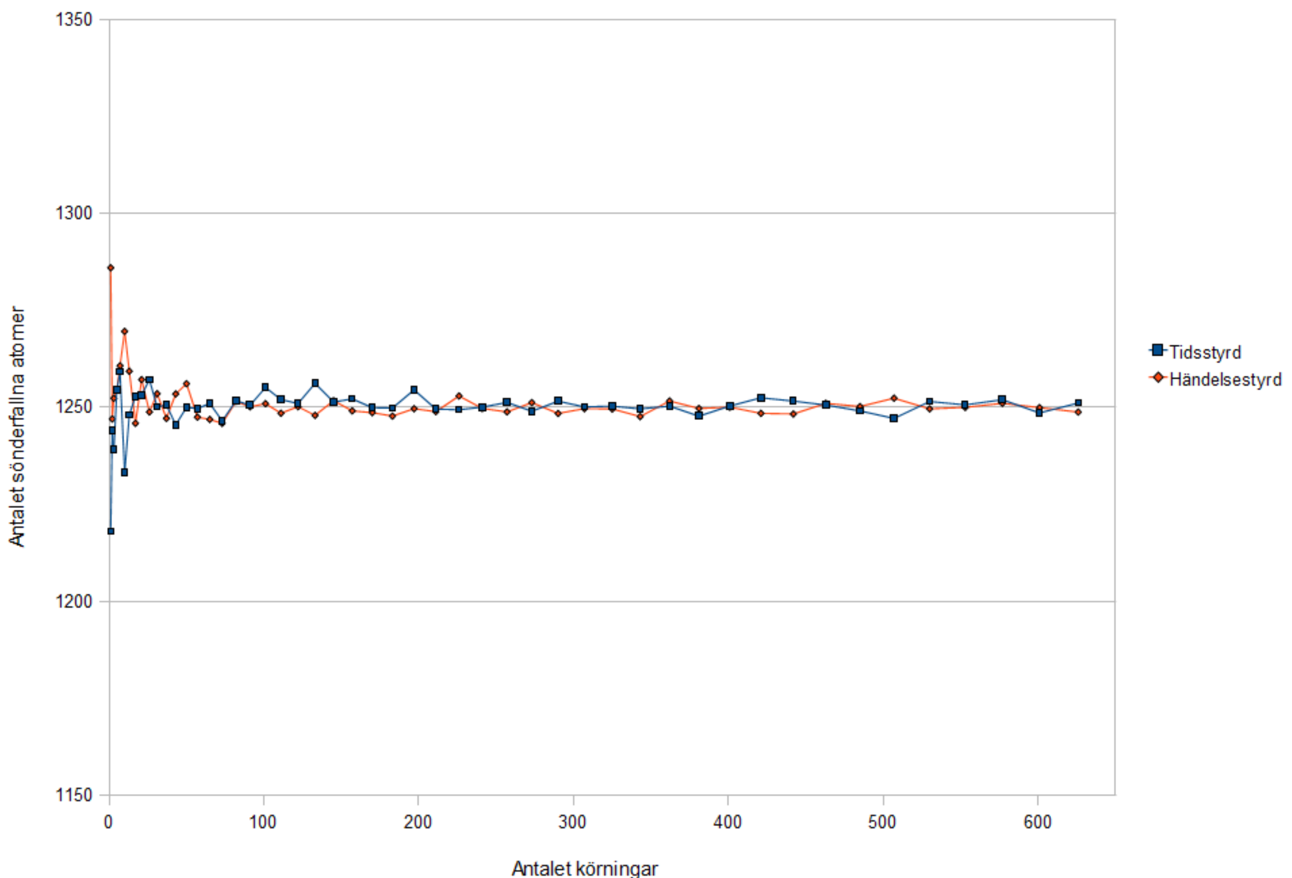
Den händelsestyrda simulationen är så snabb jämfört med den tidsstyrda att det inte syns så tydligt i figur 4.2.4 att den händelsestyrda simulationen varierar sig en aning i början av grafen. Detta blir då mycket tydligare i figur 4.2.5. Om man tittar på denna figur kan vi se att den händelsestyrda simuleringen faktiskt tar mer än dubbelt så lång tid när *burstRate* är 1 i förhållande med exekveringstiden för högre värden på *burstRate*. Kort sagt beror det på att en kedjereaktion gör den kommande ”burst”-händelsen för samma atom onödig. Om samma parametrar används som finns listade under figur 4.2.4 med *burstRate* på 1 så kommer alla atomer att direkt falla offer för en massiv kedjereaktion när det första sönderfallet inträffar. Det gör alla normala sönderfallshändelser, i snitt 1250 stycken, som kommer efter i prioritetskön helt överflödiga och irrelevanta, eftersom de redan sönderfallit. Dessutom sjunker prioritetsköns prestanda genom att ha dessa ”fantomhändelser” som element. Med stor sannolikhet är detta anledningen till den ökade exekveringstiden för lägre värden på *burstRate*.



Figur 4.2.5: Samma graf som i figur 4.2.4 men utan tidsstyrningen.

4.2.2 Statistisk analys

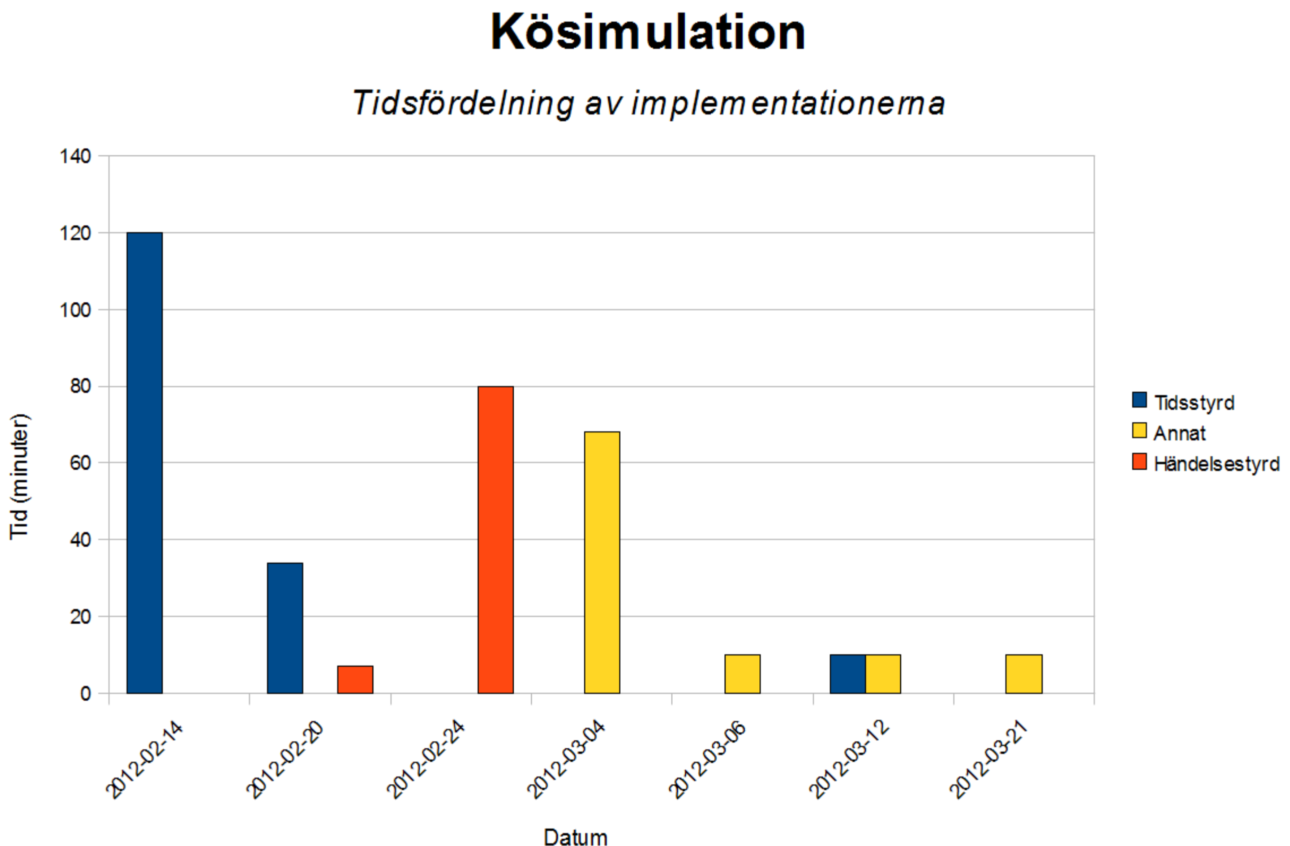
Nu undersöker vi om ett värde blir det vi förväntar, såsom i avsnitt 4.1.2. För perkolationssimulationen observerar vi antalet sönderfallna atomer, utan inverkan av kedjereaktioner. Vi kan enkelt halvera antalet atomer efter att ha kört ett lika långt tidsspann som halveringstiden. I figur 4.2.6 ser vi att det totala antalet sönderfallna atomer närmar sig det förväntade genomsnittet. I körningarna så är det lika långt tidsspann som halveringstid, vilket innebär att antalet återstående atomer bör vara hälften om allting har gjorts rätt. Antalet atomer från början är 2500 och man kan tydligt se att både den tidsstyrda och händelsestyrda simulationen ligger runt hälften, dvs. 1250.



Figur 4.2.6: Grafen utgörs av 50 punkter. Antalet körningar börjar på 1 och för varje punkt ökar det 1 + i/2, varav i är numret på punkten (räknat från vänster). De övriga inparametrarna var: tidssteg: 0.1, burstRate: 0, tidsspann: 77.27, halveringstid: 77.27, antalet atomer i bredd och höjdledd: 50.

4.3 Implementationssvårighet

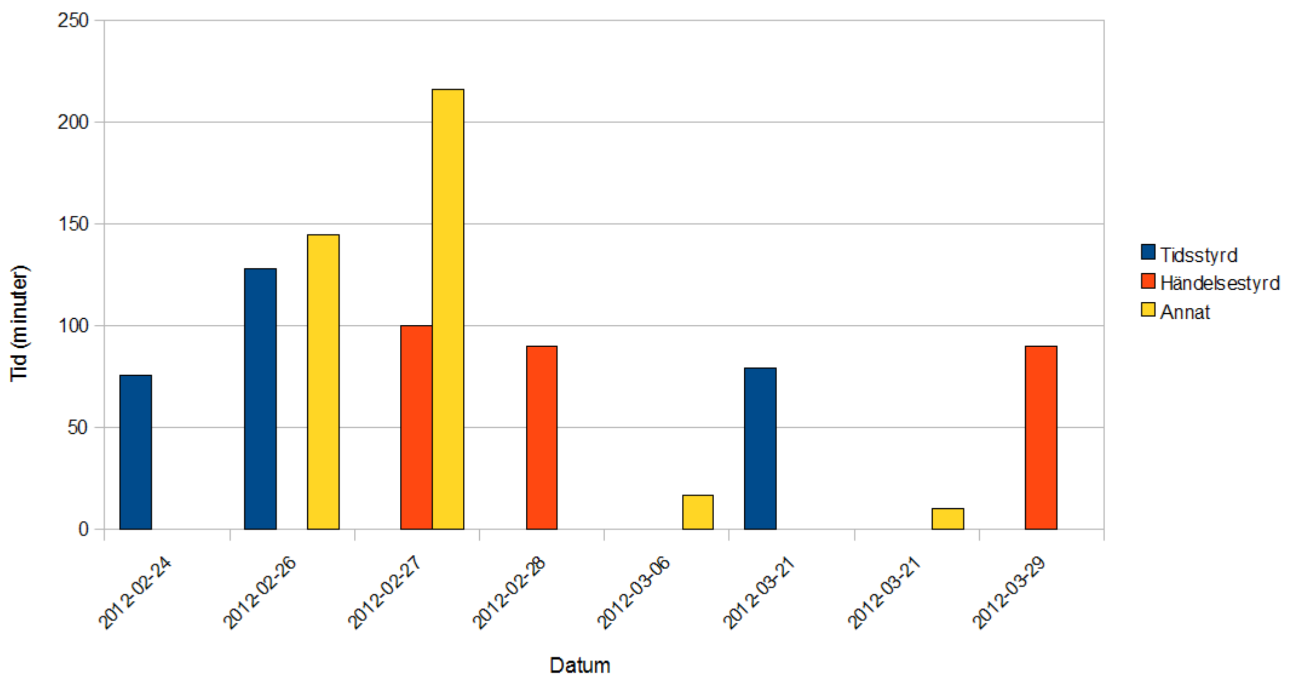
I figur 4.3.1 nedan illustreras den tid som spenderats på kösimulationen med ett stapeldiagram över de datum då programmeringen utfördes. Stapeldiagrammet har delats upp i tre olika kategorier, händelsestyrning, tidsstyrning och annat. Stapeln annat innebär att ändringar orelaterade till själva simulationsparadigmen utfördes, såsom *AtomRender* och *StatistiscsSimulation* (se avsnitt 3.2.3).



Figur 4.3.1: Stapeldiagram över implementationstiden för kösimulationen.

I stapeldiagrammet kan det verka att det tog mer tid för tidssimuleringen, men detta är inte fallet. När den tidsstyrda kösimulationen programmerades lades grunderna för de kommande implementationerna. Mycket av den kod som skrevs till tidssimulationen kunde återanvändas med små justeringar i den händelsestyrda varianten. Detta kan ses tydligt i UML-diagrammen figur 3.1.2 och 3.2.2. När händelsestyrningen sedan skulle göras var det enkelt och tydligt vad som skulle läggas till, vilket förklarar varför så lite tid behövdes på det i förhållande till tidsstyrningen. Både den tidsstyrda samt den händelsestyrda simulationen tog ungefär lika lång tid att implementera.

Rent konceptuellt var det mycket enklare att förstå hur man skulle koda tidsstyrningen jämfört med händelsestyrningen. Till exempel så undersöker tidsstyrning om en kund anländer vid en viss tidpunkt, vilket enkelt översattes till en beräkning för varje tidssteg. I händelsestyrningen däremot så genereras alla ankomster över hela tidsspannet redan innan man börjar bearbeta dem. Hur detta sker var till en början inte helt självklart. Konsekvensen blev att det blev svårt att följa tidsflödet i den händelsestyrda simulationen. Om någon annan skulle försöka sätta sig in i koden skulle nog det vara aningen mer krävande att förstå händelsestyrningen.



Figur 4.3.2: Stapeldiagram över implementationstiden för perkolationssimulationen.

Precis som kösimulationen så är delvis av den uppmätta tiden som visas i figur 4.3.2 för tidsstyrning belastad av att lägga grunderna för perkolationssimulationen. Mycket var redan färdigt när programmeringen av den händelsestyrda simulationen påbörjades. Förhållandevis så tog det alltså mer tid att göra händelsestyrningen. Det som tog mest tid under implementationen av händelsestyrningen var att uttrycka allt i händelser. En av anledningarna till denna svårighet var den samtidighet som beskrevs i modellen, dvs. att flera atomer skulle kunna sönderfalla samtidigt. I tidsstyrningen var det självklart hur det skulle gå till. Det upptäcktes också fler underliga buggar i den händelsestyrda simuleringen vilket försvårade arbetet långt efter programmeringen var ansedd klar.

5 Slutsats

I de flesta graferna framstår det tydligt att den händelsestyrda simulationen exekverar mycket snabbare än den motsvarande tidsstyrningen. Det visade sig dock också att man kan justera inparametrarna, t ex tidssteget, för att ge tidsstyrningen en snabbare exekveringstid. Även om det var möjligt kostade det alldeles för mycket på korrektheten för att det ska vara värt det.

Testerna visade att de händelsestyrda simuleringarna i båda modellerna var mycket mer exakta i den mening att något kan inträffa så gott som när som helst. Tidsstyrd simulering kan i teorin få lika hög precision som händelsestyrd simulering om ett mycket litet tidssteg används. Men det kommer till kostnaden av en ofattbart hög exekveringstid. Valet av tidssteg är en av de stora problemen med tidsstyrd simulering. Hur ska tidssteget väljas? För litet och exekveringstiden blir för stor och för stort tidssteg leder till större fel. En lösning att föredra är att vikta felmarginalen som skapas och den vunna tiden för att hitta en balans. Om tidssteget väljs på detta sätt minskar man på skillnaden mellan simulationparadigmernas resulterande exekveringstid.

Att implementera modellerna var enklast att göra i den tidsstyrda simulationen. Man kunde i princip översätta modellens flödesdiagram så var det klart. I den händelsestyrda simulationen däremot blev man tvungen att tolka om modellerna helt för att kunna förpassa allt som händelser i modellen. Flödet som syns i figur 3.1.1 och 3.2.1 i blev svårt att se i händelsestyrning jämfört med den direkta översättningen i tidsstyrning. Av samma skäl blev det svårt att åtgärda fel i koden då man inte riktigt visste var felet befann sig.

Olika parametrar för simulationerna påverkade implementationerna olika. I kösimulationen t.ex. så påverkades händelsestyrningen markant av ankomstfrekvensen samt betjäningstiden medan den tidsstyrda simulationen knappt påverkades alls. I perkolationsmodellen däremot så var det istället den tidsstyrda lösningens exekveringstid som försämrades rejält vid större *burstRate* samt antalet atomer. Slutsatsen av detta är att simulationsparadigmerna lämpar sig olika bra beroende på vad man har för indata till simulationen.

Den tidsstyrda versionen av både kösimulationen och perkolationssimulationen ledde till fler beräkningar än den händelsestyrda versionen. I perkolationssimulationen blev detta väldigt tydligt då skillnaden mellan implementationerna blev extrem när antalet atomer ökade. Den främsta anledningen till detta är att tidsstyrning låser antalet beräkningar för varje tidssteg. I den händelsestyrda lösningen däremot så utförs bara beräkningar när en händelse är aktuell. Man behöver inte undersöka mer än en gång om en händelse ska inträffa, vilket man i tidsstyrning gör för varje tidssteg. I fallet med perkolationsmodellen blev detta väldigt dyrsamt eftersom det resulterade i att varje atom måste undersökas för varje tidssteg. Självklart kan även det ske situationer där den händelsestyrda simulering blir mer belastad, såsom situationer då många händelser genereras, t.ex en hög *arrivalRate* i kösimulationen.

Tidsstyrd simulering fungerar i de flesta fall bra där den händelsestyrda fallerade och tvärt om. När t ex ankomstfrekvensen i kösimulationen var hög påverkades knappt den tidsstyrda simuleringens exekveringstid jämfört med den händelsestyrda, som gick mycket långsammare. Däremot så kräver tidsstyrningen ett väldigt litet tidssteg för ge korrekta resultat, vilket kombinerat med ett längre tidsspann resulterar i att händelsestyrningen exekverar mycket snabbare.

Så var den ena simulationsparadigmen bättre än den andre? Det går egentligen inte att säga helt på förhand vilken simulationsparadigm som fungerar bäst. Beroende på modellen kan det blir svårt om inte omöjligt att tolka saker i händelser. Redan i dessa två relativt enkla modeller var det inte helt självklart hur händelser skulle integreras. Tidsstyrning är flexibel på det sättet då modellen blev väldigt lätt att översätta till kod. Den resulterande koden reflekterade även den ursprungliga modellen väl. Problemet med tidsstyrning är dock att den leder till tunga beräkningar vid låga tidssteg, vilket tidssteget måste vara för att hålla korrekthetsfelen nere. Igen så beror även detta på modellen som ska simuleras. Alla modeller behöver inte nödvändigtvis involvera statistik som i de modeller som implementerades i denna studie där numeriska värden betyder allt. Den kulminerade slutsatsen är att om man kan så bör man implementera händelsestyrning eftersom det i överlag gav bäst exekveringstid.

Det finns dock ett alternativ. En av de saker som visades i testningen var att simulationsparadigmernas lösningar presterade olika beroende på parametrarna. Om man kan implementera modellen med båda simulationsparadigmerna kan man skapa en slags hybridlösning. I denna lösning skulle man analysera inparametrarna och välja den implementation som exekverar snabbast. I UML-klassdiagrammen 3.1.2 och 3.2.2 syns det tydligt att klasstrukturen i princip är densamme för både tidsstyrningen och händelsestyrningen. I detta fall skulle hybridlösningen enkelt kunna integreras genom att undersöka parametrarna till *StatisticsSimulation.* Underlaget för parameteranalysen skulle också enkelt kunna härledas utifrån tidigare testkörningar som de i avsnitt 4. Förstås skulle detta kräva mer tid att programmera och det finns ingen garanti att det är en vettig lösning i det allmänna fallet.

Avslutningsvis så presenteras de härledda egenskaperna i tabell 5.1 nedan.

Händelsestyrning	Tidsstyrning
+ Allmänt låg exekveringstid. + Lämplig för många simuleringar. + Hög precision oberoende av inparametrar.	+ Lätt att tolka modellerna. + Resulterande kod lätt att följa. + Går att anpassa inparametrar för att nå kortare exekveringstid. + Enkel att implementera
- Specialanpassningar av modellerna. - Resulterande kod svår att följa. - Kräver inlärningsperiod för att förstå konceptet.	- Hög exekveringstid för korrekta resultat - Allmänt hög exekveringstid - Justering av tidssteget kan leda till märkligheter i simulationen.

Tabell 5.1: Tabell som sammanställer de egenskaper som framtagits genom analysen av dels implementationens utförande och analysen av implementationen i sig.

6 Referenser

Referenserna har ordnats i den ordningen som de förekommer i uppsatsen.

[1]. Albrecht MC. *Introduction to DES* [hemsida på Internet]. 2010 [citerades 2012 Feb 23]. Tillgänglig från: http://www.albrechts.com/mike/DES/Introduction%20to%20DES.pdf

[2]. Moler C. *Predator-Prey Model* [hemsida på Internet] 2011 [citerades 2012 Mars 4] Tillgänglig från: http://www.mathworks.com/moler/exm/chapters/predprey.pdf

[3]. Svenska Akademins Ordbok [hemsida på Internet]. 2011 [citerades 2012 Mars 4] Tillgänglig från: http://g3.spraakdata.gu.se/saob/

[4]. Kozdron MJ. *An Introduction to Percolation* [hemsida på Internet]. 2007 [citerades 2012 Mars 4] Tillgänglig från: http://stat.math.uregina.ca/~kozdron/Research/Talks/regina_percolation.pdf

[5]. Oracle. *Java Platform, Standard Edition 7 API Specification* [hemsida på Internet]. 2011 [citerades 2012 Mars 06] Tillgänglig från: http://docs.oracle.com/javase/7/docs/api/

[6]. The Eclipse Foundation. *Eclipse* [hemsida på Internet]. 2012 [citerades 2012 Mars 06] Tillgänglig från: http://www.eclipse.org/

[7]. OMG. *Unified Model Language* [hemsida på Internet]. 2012 [citerades 2012 Mars 06] Tillgänglig från: http://www.omg.org/spec/UML/index.htm

[8]. Oracle. *What is Swing?* [hemsida på Internet]. 2012 [citerades 2012 Mars 06] Tillgänglig från: http://docs.oracle.com/javase/tutorial/ui/overview/intro.html

[9]. JExcelAPI. *Java Excel API - A Java API to read, write, and modify Excel spreadsheets* [hemsida på Internet]. 2009 [citerades 2012 Mars 25] Tillgänglig från: http://jexcelapi.sourceforge.net/

[10]. Björk LE., Brolin H., Pilström H., Alphonce R. *Formler och Tabeller från Natur och Kultur* Eskilstuna: Bokförlaget Natur och Kultur; 2006