Uppsala University

Introduktion till informationsteknologi

A Bug's Life

Authors:
Anonymous Student Team
Aletta Nylén

29 september 2023

1 Inledning

Swarm intelligence är ett koncept som innebär att en grupp av organismer, helt utan en ledare i centrum och endast med begränsad individuell intelligens, i grupp kan lösa komplexa problem. Till exempel består ett försvar av soldater. Varje soldat kan inte göra så mycket själv, men som grupp är soldaterna mycket starkare.

BugsLife är en simulering där myror kan hitta utplacerad mat och sedan ta tillbaka den till sitt bo. Genom att lämna feromoner på vägen kan myrorna sedan hitta tillbaka till maten genom att följa sina egna spår. Systemet är konstruerat så att det är fullständigt concurrent och arbetar helt utan någon supervisor eller tidssynkronisering. Detta har åstadkommits genom att låta varje objekt i simuleringen vara en egen process där all kommunikation mellan processerna sker via meddelandeöverföring. Systemet använder sig utav actor-modellen och är fritt från deadlocks.

Genom att simulera enklare

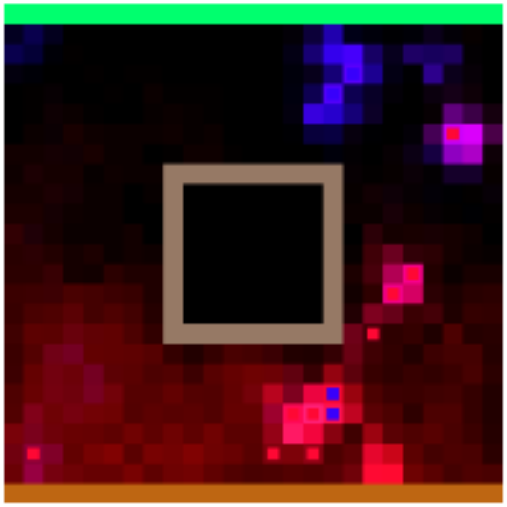
2 Översikt över systemet

Systemet ger användaren möjligheten att observera simuleringen av en virtuell myrkoloni i realtid.

Användaren startar simuleringen utan input-parametrar och systemet körs sedan automatiskt och kräver ingen interaktion från användaren. Vid första anblick kan myrkolonin uppfattas som simpel, men komplexiteten i simuleringen ligger i systemarkitekturen och de algoritmer myrorna använder sig av för att effektivisera hämtningen av mat. En skärmdump av simuleringen kan ses i Figur 1.

2.1 Concurrency

På grund av den höga nivån av autonomitet kombinerat med separationen mellan de olika processerna används så kallad eventdriven programmering, vilket är när ett program styrs av externa händelser som inte går att kontrollera eller förutse. Ett exempel på det är vid programmering av användargränssnitt, då det inte går att förutspå när eller i vilken ordning användaren kommer att ge input till systemet.



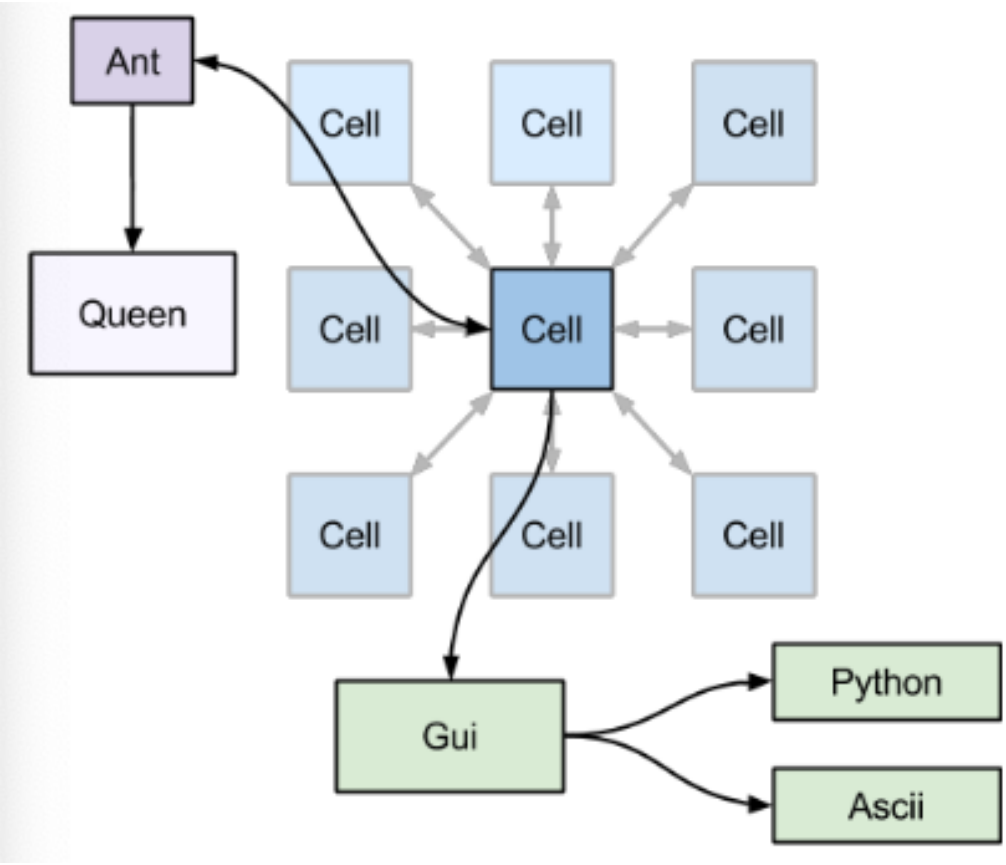
Figur 1: En skärmdump utav simuleringen. De brun/orangea rutorna är myrbo, de gröna är mat, de bruna/gråa är väggar. De små röda rutorna är myror som letar efter mat och de små blåa rutorna är myror som är på väg hem med mat. I bakgrunden syns feromonerna.

Systemet använder sig utav actor-modellen då den ger bra möjligheter att på ett konceptuellt och effektivt sätt implementera eventdriven programmering. Actor-modellen ger en möjlighet att fokusera på en modul i taget utan att behöva ta hänsyn till hur de andra modulerna är implementerade. Det enda som behövs är ett meddelandeprotokoll som alla moduler följer.

Det är viktigt att påpeka att systemet använder sig utav asynkron eventdriven programmering. Detta ger upphov till många problem då det inte går att garantera att meddelandena kommer i rätt ordning. Problemen löses med hjälp av meddelandefiltrering, som innebär att en process väntar på ett specifikt meddelande innan den går vidare och buffrar de meddelanden som kvarstår. Varje process behöver då ha ett speciellt tillstånd när den accepterar alla nya förfrågningar.

2.2 Systemdesign

I figur 2 syns en schematisk överblick av systemarkitekturen. Alla moduler arbetar helt asynkront och all kommunikation mellan modulerna sker via meddelanden.



Figur 2: En schematisk överblick över systemarkitekturen

2.2.1 Ant

Ant-modulen är den modul som implementerar myran. Ant-modulen innehåller de funktioner som krävs för att myran skall kunna fatta beslut över hur den ska agera baserat på hur dess omgivning ser ut. Myran skickar endast förfrågningar till den cell som den befinner sig på. Under normal drift så tar myran aldrig emot några förfrågningar från någon annan process, den tar endast emot svar från den cell som den befinner sig i.

Myran kommer på eget bevåg att skicka förfrågningar till den cellen den står på och autonomt att fatta beslut om vilken handling som är lämpligast, baserat på den informationen myran har tagit del av.

Ant-modulen är utrustad med funktionalitet så att myran kan analysera

den information som ges av grannskapet till cellen den står i. Informationen gör det möjligt för myran att fatta ett beslut, baserat på vad myrans omgivning består av. Myror i det verkliga livet kommunicerar med varandra genom olika typer av feromoner, detta för att meddela var maten finns. Det är viktigt att myrorna kan inspektera sin omgivning så att den kan fatta ett beslut baserat på vad andra myror har upptäckt i omgivningen. Finns det feromoner som indikerar till myran att det finns föda i närheten kan myran ta ett beslut att röra sig i den riktningen och leta rätt på födan. Myrorna använder sig även av feromoner för att hitta hem till sitt näste.

2.2.2 Cell

Cell-modulen representerar cellerna, varje cell är en ruta i det rutnät som utgör världen där myrorna lever. En cells primära uppgift är att ta emot och besvara förfrågningar från myran som står på cellen, samt de kringliggande cellerna. En cell kommer under normal drift aldrig att skicka en begäran till en annan cell, förutsatt att det är inte en begäran för att hantera en förfrågan som cellen har tagit emot. En cell kan endast skicka förfrågningar till de celler som är i dess direkta grannskap.

Cellen är det objekt som innehåller den relevanta informationen för systemet. En cell innehåller information om vad det är för typ av cell, om cellen exempelvis representerar ett myrbo: huruvida det finns mat på cellen, vad för intensitet de olika feromonerna har på cellen och om det står en myra på cellen.

Cellerna är de enda objekten som kommunicerar med GUI-modulen. När en cell får en förfrågan som förändrar cellens tillstånd skickar cellen ett meddelande till GUI-modulen där den talar om sin position och sitt nya tillstånd.

2.2.3 Message_Buffer

Message Buffer är den modul som hanterar kommunikationen av meddelanden mellan processerna. Modulen exporterar receiver-funktionen som är central för att använda den typ av eventdriven programmering som används. Receiver-funktionen används för att hantera deadlocks och filtrering av meddelandekön.

2.2.4 GUI

GUI-modulen är implementerad i Python och Erlang med hjälp av ErlPort. Modulen tar emot och behandlar meddelanden innehållandes cellernas tillstånd från cellerna. Informationen omvandlas till Pythons syntax via Erlport och skickas till en Python-instans som hanterar den grafiska representationen.

2.2.5 Grid_Init

Modulen används för att bygga upp världen samt placera alla celler och länka ihop dessa. Grid Init modulen tilldelar även alla celler dess initiala attribut samt skapar, placerar och startar alla myrorna.

I denna modul initieras även en Queen-process, denna process binds alla myror till. Myrorna skickar statistik till Queen-processen, modulen är till för debugging och prestandamätning.

3 Implementation

3.1 Programmeringsspråk

Som programmeringsspråk används Erlang då det är ett väletablerat och stabilt språk som är byggt kring actor-modellen och meddelandeöverföring. Utöver Erlang används Python för den grafiska delen och ErlPort[1] för kommunikationen mellan Erlang och Python. Implementationen av ErlPort har fungerat mycket bra och har gett oss ett smidigt interface mellan Erlang och Python.

3.2 GUI

GUI-modulen initieras med att skapa en lista motsvarande hela griden, den fylls med tomma atomer som Python-modulen inte ritar ut. GUI-modulen körs i en main loop som tar emot meddelanden från cell-modulerna, bearbetar dessa och fyller på den initierade listan.

En algoritm används för att fördela listan på rätt X och Y koordinater, koordinaterna är angivna i meddelandet från cellen. Listan skickas via ErlPort till Python där den ritas upp grafiskt.

[1] http://erlport.org/

3.2.1 Erlport

Erlport är ett bibliotek som gör det möjligt för Erlang att kommunicera med andra programmeringsspråk. I dagsläget har Erlport stöd för Python och Ruby.

För att Erlport ska kunna kommunicera med Python så skapas först en Python-instans. En Python-instans är i princip en OS-process som representeras i Erlang av en Erlang process. Man kan skicka och ta emot medelanden mellan Erlang och Python på samma sätt som man skickar meddelanden mellan vanliga processer i Erlang.

3.2.2 Pygame

För den grafiska renderingen i projektet används det community-utvecklade Pythonbiblioteket Pygame. Pygame är ett bibliotek av färdigutvecklade moduler till Python designade för att enkelt kunna skapa spel med en enkel grafisk implementation. Pygames design gör att det är enkelt att använda på alla plattformar som har stöd för Python.

3.3 Ants and Cells

Ant-modulen och Cell-modulen är väldigt lika varandra i sin implementation. Båda modulerna består utav en initialiseringsfunktion som används för att starta processer. Funktionen inväntar de meddelanden som är nödvändiga för att starta processerna. Detta kan till exempel vara ett meddelande om att myran har placerats korrekt och att cellerna har fått sitt grannskap definierat.

Cellerna och myrorna går efter ett meddelande till sin mainfunktion där de inväntar nya förfrågningar. Om det inte finns några obehandlade meddelanden kommer myrorna att agera spontant. När cellerna och myrorna får ett inkommande meddelande kommer de att anropa en funktion specifikt för den typen av meddelande. Dessa funktioner kan skicka och ta emot meddelanden själva med hjälp av meddelandefiltreringen. När en förfrågan eller annat meddelande har behandlats kommer processen återgå till sin mainfunktion. Om ett felaktigt eller otillåtet meddelande inkommer under någon del av exekveringen så kommer hela systemet att krascha.

Cellerna kommer vid varje inkommen förfrågan, beroende av cellens tillstånd, automatiskt genomföra en uppdatering av dess feromonnivåer. Detta sker som en funktion av den faktiska tiden (wall-time) som har gått sedan

cellen senast uppdaterades.

Om en cell har attributet block så kan en myra inte gå dit och alla `place_ant` förfrågningar kommer att misslyckas.

3.3.1 Myrans algoritm

Myran beslut baseras på en väldigt enkel algoritm. Myran kan vara i två olika tillstånd `searching_for_food` och `returning_with_food`.

När myran letar efter mat så kommer den att undersöka cellen den står i, om det finns mat i cellen så kommer myran att försöka plocka upp maten. Om myran lyckas plocka upp mat kommer den att byta tillstånd till `returning_with_food`, om myran misslyckas med att plocka upp mat så kommer den att fortsätta leta efter mat i andra celler.

När myran letar efter mat så kommer den att be cellen den står i att skicka tillbaka information om alla celler i dess grannskap. Myran kommer sedan att studera sitt grannskap och sortera cellerna efter hur mycket `food_feremone` varje cell innehåller och sedan genom *rank-selection* att välja den riktingen den skall gå i. Rank-selection innebär att den med en förutbestämd sannolikhet p kommer att gå till den cellen med det högsta antalet feromoner. Om den inte väljer den riktningen så kommer den att gå till den cellen med näst högst riktning med samma sannolikhet p.

Myran kommer efter varje genomförd förflyttning att släppa feromoner på den cellen där den tidigare var. Då myran letar efter mat kommer den att släppa `base_feremone` och då den går tillbaka med maten så kommer den att släppa `food_feremone`.

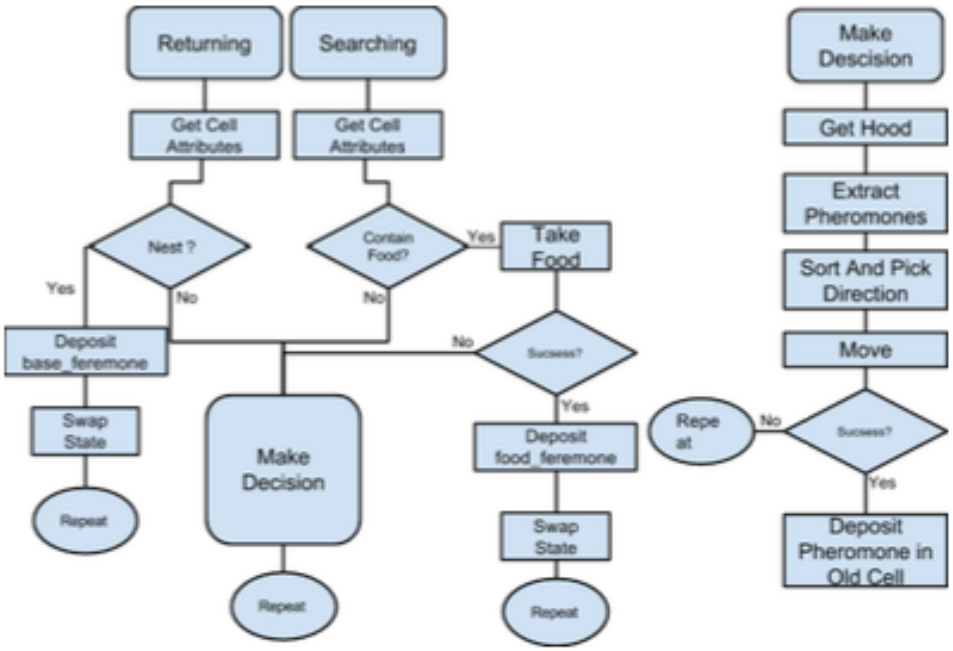
Då myran går tillbaka med mat så kommer den att följa en snarlik algoritm men istället leta efter den högsta koncentrationen utav `base_feremone`.

I figur 3 finns en flow-chart om hur algoritmen ser ut.

3.3.2 Grid_Init

Grid init är den modul som bygger upp världen. Det första som den modulen gör är att den startar alla cell-processer. Det krävs att GUI-modulen är startad, dess PID kommer att skickas med till alla celler. Den kommer sedan att skicka ut `set_cell_attribute` medelanden till alla dessa celler med cellernas attribut. Grid_init kommer sedan att initiera en *Queen*[2] process och starta

[2]Queen processen samlar in statistik från myrorna och används endast för debugging och testning.



Figur 3: Flowchart över myrans algoritm

alla myror och försöka placera ut dem på de celler där de skall starta. När alla myror har blivit utplacerade så kommer Grid_init modulen att skicka `start_ant` medelanden till myrorna vilket kommer att starta simuleringen.

3.4 Concurrency

Då concurrencyn i systemet uteslutande bygger på actor-modellen och message passing så har systemet tre definierade klasser utav meddelanden.

- Enkelrikatade meddelanden på formen
  `{Pid,{Type,Payload}}` eller `{Pid,Type}`
  Enkelriktade meddelanden är meddelanden som inte kommer att resultera i att något svar inkommer.

- Förfrågningar på formen
  `{Pid,Reference,Payload}`
  Alla förfrågningar resulterar i att processen blockerar och inväntar svarsmeddelanden.

- Svarsmeddelanden på formen
  `{Pid,Reference,Request_Reference,{Type,Reply_Payload}}`
  Svar är de meddelanden som skickas som svar på förfrågningar.

För alla meddelanden gäller att `Pid` är process id:t hos den process som skickar meddelandet, `Payload` är vad meddelandet faktiskt innehåller och `Reference` är en referens som den skickande processen ger meddelandet. `Request_Reference` är en referens från en förfrågning till vilket detta är ett svar.

Då systemet faller under *asynkron eventdriven programmering* så kan vissa problem uppstå om vi inte är försiktiga med hur vi implementerar systemet. Det naiva tillvägagångssättet hade varit en så kallad *Fifo, run to completion* metod. Detta innebär att man accepterar varje meddelande, hanterar det och låter det köra tills det att det är klart innan man hanterar nästa meddelande. Fifo-metoden leder dock till problem med att man måste hålla explicit koll på vilket tillstånd processen är i och ha en plan för hur man ska hantera alla kombinationer av meddelande-tillstånd. Detta skulle ha lett till extremt komplicerad och svårhanterlig kod. [1]

Lösningen på detta är att använda vad vi kallar för *state-driven message handling*, vilket är att vi låter en process tillstånd diktera vilka meddelanden

som den kommer att hantera. Det vi gör är att vi implementerar ett meddelandefilter som bara returnerar rätt meddelande (baserat på meddelandets unika tag/referens) och alla andra meddelanden som inkommer läggs på en buffer så att de senare kan hanteras.

Detta leder till att koden blir kortare och lättare att underhålla då vi på ett lätt sätt kan hantera hur systemet beter sig. Det finns dock ett krav på att alla processer måste ha ett eller flera tillstånd då de accepterar nya meddelanden. I detta tillstånd kommer meddelanden som ligger på buffern att hanteras först och när buffern är tom kommer de nya meddelandena att hanteras.

3.4.1 Deadlocks

Simuleringen är full av deadlocks och de uppstår ofta. Därför behövs ett system för att upptäcka och lösa dessa.

För enklare deadlocks går detta att lösa effektivt och deterministiskt. Om en process väntar på svar på en förfrågan så kommer den att blockera. Om en process väntar på ett svar från en annan process men får en förfrågan från den processen innan den har fått svar så har ett deadlock uppstått. Processen som upptäcker att ett deadlock har uppstått kommer direkt att ge svar om misslyckande till den förfrågan som orsakade deadlocken. Det leder till att den andra processen kommer sluta blockera och förfrågan kan därefter hanteras. Detta förutsätter att det alltid är tillåtet för en förfrågan att misslyckas och att en process alltid kommer efter en finit tid kunna hantera nya förfrågningar efter det att en förfrågan har misslyckats.

Den här metoden kan bara endast lösa deadlocks som uppstår mellan två processer, men systemet drabbas ständigt av mer komplicerade deadlocks.

För att lösa mer komplicerade distribuerade deadlocks skapades en metod som är väldigt simpel men mycket effektiv. Metoden löser alla deadlocks helt automtiskt och kräver inte att några rollbacks måste genomföras eller analys av det globala tillståndet med en Wait-For-graf. [2] Metod kräver inte heller att några probe meddelanden skickas mellan processerana som i Chandy-Misra-Haas algoritmen. [3]

Dock så predikeras vår metod på en uppsättning krav på systemet.

- Alla förfrågningar kan misslyckas på ett väldefinierat sätt.
- Alla processer kommer att invänta ett svar efter en förfrågan har skickats.

- Från det att ett svar på en förfrågan har inkommit kommer processen alltid att inom en finit tid drivas till ett tillstånd där den accepterar inkommande förfrågningar.

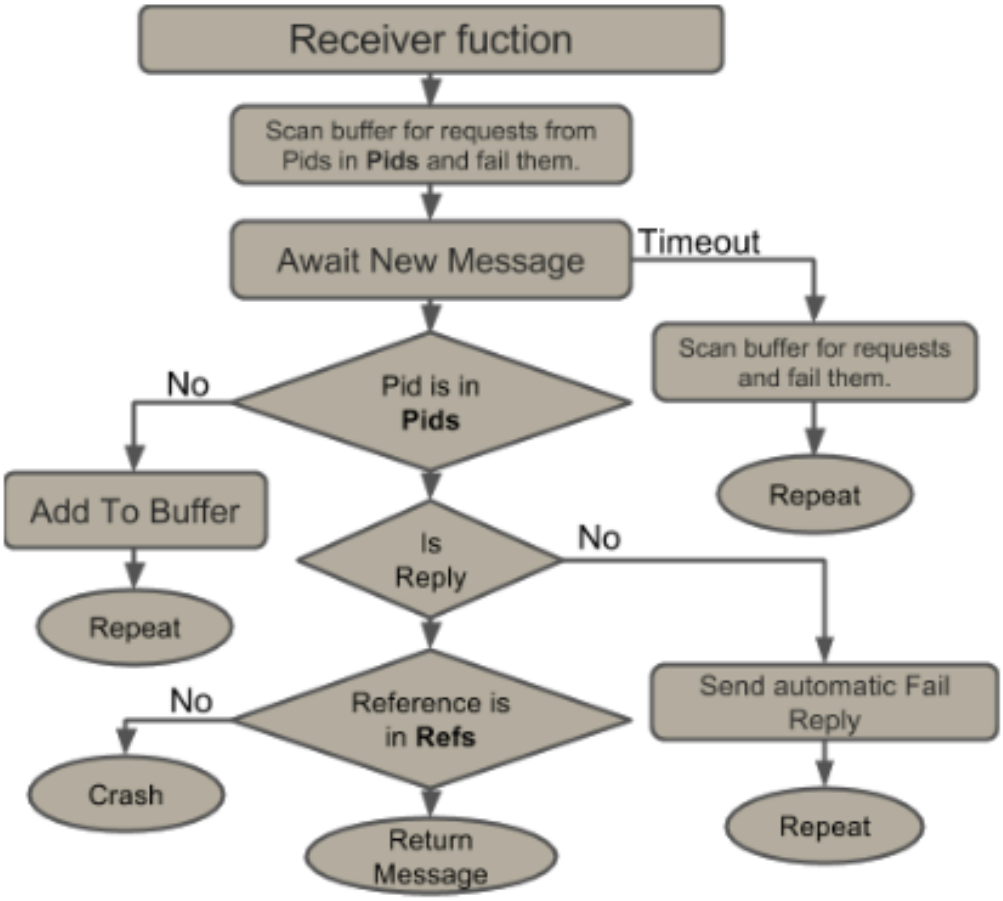
Dessa är en uppsättning preliminära predikat. Det kan finnas ytterligare begränsningar som har missats då de inte är applicerbara på vårt system. Aven då dessa predikat kan förefalla vara hårda så innebär inte det att funktionaliteten drabbas signifikant. De tillåter att en process som väntar på ett svar fortfarande kan hantera vissa meddelanden och skicka nya förfrågningar till processer. De tillåter även att man implementerar en prioritering utav olika typer av förfrågningar eller olika typer av processer.

När en process väntar på svar från en annan process så kommer den att efter en viss förutbestämd tid (timeout), skicka automatiska fail-svar till alla inkomna förfrågningar som ligger på dess meddelande-buffer. Detta kommer att repeteras tills att ett svar har inkommit. Ni kan se en schematisk beskrivning av algoritmen i figur 4.

Vår metod har dock den nackdelen att den ofta kommer att upptäcka falska deadlocks, det vill säga att den kommer att tro att det finns ett deadlock när det inte gör det och i onödan avvisa vissa förfrågningar. Detta leder till att **Timeout** parametern måste finjusteras. En timeout som är för lång kommer innebära att deadlocks kommer att ligga kvar för länge innan de upptäcks och delar utav simuleringen kommer att lagga. En för kort timeout kommer att leda till att väldigt många förfrågningar kommer att avisas i onödan vilket också kan påverka systemets prestanda.

4 Slutsatser

Vi har lyckats göra en representation av en myrkoloni som samlar mat och kommunicerar med varandra med hjälp av feromoner. Myrorna kan även undvika olika hinder och objekt. Aven om vi inte riktigt lyckades implementera allt vi hade iåtanke när projektet började så är vi nöjda med slutresultatet. Det är svårt att få en verklig uppfattning av problemen och utmaningarna med ett arbete innan man satt igång med grunden på vilken andra implementationer ska vila. Vi bestämde oss för att se till att göra en grundläggande representation av en myrkoloni och se till att denna har god concurrency och en felfri körning, detta har vi uppnåt och även utvecklat en grafisk representation Det är detta som vi nämnde tidigare, som är vår grund. På



Figur 4: En schematisk överblick över hur meddelandefiltreringen och deadlockhanteringen fungerar. **Refs** är en lista med de referenser från de förfrågningar meddelanden som har skickats. **Pids** är en lista med de Pids som förfrågningarna har skickats till.

denna grund kan man enkelt bygga ut och genom detta implementera flera funktioner, som till exempel flera olika typer av mat, fientliga insekter och en konkurrerande myrkoloni. Aven om vår första tanke var att använda det nya och lite mer spännande språket Nim, så är vi nöjda med att vi valde att använda det mer utvecklade och etablerade språket Erlang. De verktyg som Erlang har tillgängliga har varit hjälpsamma och det är ett intressant språk att skriva i och actor-modellen är lätthanterlig för den här typen av concurrency.

Sammanfattningsvis så har systemet levt upp till de förväntningar som gruppen hade haft i projektets tidiga stadie.

Referenser

[1] U. Wiger, "Death by accidental complexity," 2010. Tillgänglig på http://www.infoq.com/presentations/Death-by-Accidental-Complexity.

[2] P. Krzyzanowski, "Distributed deadlock," 2012. Tillgänglig på https://www.cs.rutgers.edu/ pxk/417/notes/deadlock.html.

[3] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 2, pp. 144–156, 1983.