HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Analyzing Different Scheduling Policies in Natjam using Timed Rebeca

Helgi Leifsson

Thesis of 30 ETCS credits
Master of Science in Computer Science

January 2015

Analyzing Different Scheduling Policies in Natjam using Timed Rebeca

Helgi Leifsson

Thesis of 30 ECTS credits submitted to the School of Science and Engineering
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science in **Computer Science**

January 2015

Supervisor:

Dr. Marjan Sirjani, Supervisor
Professor, Reykjavik University, Iceland

Examiners:

Dr. Anna Ingólfsdóttir, Examiner
Professor, Reykjavik University, Iceland

Dr. Prasad Saripalli, Examiner
CTO & VP Engineering, Secure Fabric

Analyzing Different Scheduling Policies in Natjam using Timed Rebeca

Helgi Leifsson

30 ECTS thesis submitted to the School of Science and Engineering
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science.

January 2015

Student:

Helgi Leifsson

Supervisor:

Dr. Marjan Sirjani

Examiners:

Dr. Anna Ingólfsdóttir

Dr. Prasad Saripalli

Date

Helgi Leifsson
Master of Science

Analyzing Different Scheduling Policies in Natjam using Timed Rebeca

Helgi Leifsson

January 2015

Abstract

As computer systems become larger and more complex, such as with the advent of clouds, scientists and engineers can use software to correctly set up and evaluate their performance. Many such software tools are available today but have not dealt with deadline based scheduling and preemption of jobs running concurrently. If such software was available it could lead to more efficient use of current and future systems.

With the concurrency and distribution of computation come problems such as nondeterminism and race conditions which can be difficult to simulate and reproduce. In this project we present the ReGen software that uses Timed Rebeca to run Monte Carlo simulations of Natjam which is built into the Hadoop YARN MapReduce cluster software. It can be used to measure the efficiency of different job dispatch and job eviction policies in the presence of preemption. Many variables are under our control such as size of cluster, workload, deadline computation and more.

We present results showing the performance of EDF, FIFO, MDF and Priority Queue dispatch policies, and MDF and MLF policies for job eviction. The results suggest MDF is preferable for both dispatching and job eviction. We conclude that software to solve the above problem can be made and like to argue that our results can transfer to other systems that use the policies tested in this project.

Greining á Verkefnadreifingarstefnum í Natjam með Timed Rebeca

Helgi Leifsson

Janúar 2015

Útdráttur

Þegar tölvukerfi verða stærri og flóknari, eins og með tilkomu tölvuskýja, geta vísindamenn og verkfræðingar stuðst við hugbúnaðartól til að setja þau rétt upp og meta afköst þeirra. Mörg slík eru fáanleg en hafa hingað til ekki átt við dreifingu á verkefnum með tímafrest. Ef slíkur hugbúnaður væri til gæti það leitt til betri nýtingar á núverandi og framtíðar kerfum.

Með samhliða og dreifðum útreikningum koma vandamál eins og óregluleiki og kapp aðstæður sem erfitt getur verið að herma og endurskapa. Í þessu verkefni kynnum við ReGen hugbúnaðinn sem notar Timed Rebeca til að keyra Monte Carlo hermun af Natjam sem byggt er inn í Hadoop YARN MapReduce hugbúnaðarpakkann. Það getur verið notað til að mæla afköst mismunandi verkefnadreifingar- og verkefnabrottrekstrarstefna við aðstæður þar sem verkefni geta verið tafin til að keyra önnur verkefni. Hægt er að stjórna mörgum breytum eins og stærð tölvukerfisins, verkefnaálagi, tímafrestsútreikningum og fleiru.

Við kynnum niðurstöður sem sýna afköst EDF, FIFO, MDF og Priority Queue verkefnadreifingarstefna og MDF og MLF verkefnabrottrekstrarstefna. Þær niðurstöður gefa til kynna að MDF sé betri valkostur bæði fyrir dreifingu og brottrekstur verkefna.

Við drögum þá ályktun að þróun hugbúnaðar til að leysa ofantöld vandamál sé möguleg og viljum halda því fram að niðurstöðurnar sé hægt að yfirfæra á önnur kerfi sem nota þær stefnur sem prófaðar voru í þessu verkefni.

Dedicated to my parents and family.

Acknowledgements

Dr. Marjan Sirjani for her supervision, guidance, experience, teachings, expertise and everything. This work would not have been possible without her.

My examiners Dr. Anna Ingólfsdóttir and Dr. Prasad Saripalli for their invaluable input.

Dr. Indranil Gupta and Muntasir Raihan Rahman for all their helpful input.

Fellow students Ehsan Khamespanah and Ali Jafari for all their help and teamwork.

My family and friends for all their support throughout the years.

The work on this thesis has been partially supported by the project Timed Asynchronous Reactive Objects in Distributed Systems: TARO (nr. 110020021) of the Icelandic Research Fund.

Contents

List of Figures **xii**

List of Tables **xv**

1 Introduction **1**

1.1 Contribution 2

1.2 Overview of the Thesis 2

2 Background **3**

3 Methods **7**

3.1 Example Hadoop MapReduce and ReGen Scenario 7

3.2 ReGen Architecture, Diagrams and User Interface 8

3.2.1 ReGen Classes 10

3.3 Dispatch Policies 10

3.3.1 Dispatch Model Overview 10

3.4 Natjam-R 16

3.4.1 Natjam-R Model Overview 16

3.4.2 Avoiding Concurrency Problems 21

4 Experiments **25**

4.1 Test Setup 25

4.2 Dispatch Policy Results 25

4.2.1 Common parameters to the dispatch policies 26

4.2.2 Earliest Deadline First (EDF) 27

4.2.3 First-In-First-Out (FIFO) 32

4.2.4 Maximum Deadline First (MDF) 33

4.2.5 Priority Queue 36

4.2.6 Summary of the Dispatch Policy Results 40

4.3 Natjam-R Job Eviction Policy Results 44

4.3.1 Common parameters between the Natjam policies (Table 4.15) . . 45
4.3.2 Maximum Deadline First (MDF) 45
4.3.3 Maximum Laxity First (MLF) 46
4.3.4 Summary of the Natjam-R Eviction Policy Results 48

5 Conclusions 53
5.1 Conclusion and Future Work 53
5.1.1 Conclusion 53
5.1.2 Future work 54
5.2 Discussion 54

Bibliography 57

A ReGen User Manual 61
A.1 Policy Options 61
A.1.1 Dispatch 61
A.1.2 Natjam-R eviction 61
A.1.3 Dispatch policies 61
A.1.4 Natjam-R policies 62
A.1.5 Job arrival patterns 62
A.1.6 Job length patterns 62
A.2 Common parameters 63
A.2.1 Max AppMasters 63
A.2.2 Queue size 63
A.2.3 Simulation traces 63
A.2.4 Simulation timeunits 63
A.2.5 Epsilon 63
A.3 Job arrival and length parameters 64
A.3.1 Burst interval 64
A.3.2 Burst size 64
A.3.3 Nondeterministic job arrival/job length 64
A.3.4 Uniform value for job arrival/length 64
A.3.5 Wave job arrival/job length 64
A.3.6 Ascending job arrival/job length 65
A.3.7 Descending job arrival/job length 65
A.3.8 Exponential multiplier 66
A.4 High priority job options 66
A.4.1 Probability % 66

A.4.2 Length 66

A.5 Natjam-R options 66

A.5.1 Checkpoint overhead 66

A.6 Input/Output Options 67

A.6.1 Output path 67

A.6.2 Prefix 67

A.6.3 Compiler path 67

A.6.4 Traces path 67

A.7 Results window 67

A.8 Run button 68

B ReGen Generated Code Sample 69

List of Figures

2.1 Abstract syntax of Timed Rebeca 4
2.2 A Timed Rebeca model of a ticket service system 4
2.3 Overview of Hadoop’s YARN 5

3.1 Bursty job arrival pattern 11
3.2 Exponential job length pattern 11
3.3 Nondeterministic job arrival and length pattern 12
3.4 Uniform job arrival and length pattern 12
3.5 Wave job arrival and length pattern 13
3.6 Ascending job arrival and length pattern 13
3.7 Descending job arrival and length pattern 14
3.8 ReGen user interface 15
3.9 ReGen Sequence diagram 16
3.10 ReGen Class diagram 17
3.11 ReGen Component diagram 18
3.12 Dispatch policy model message passing overview 18
3.13 Dispatch policy code template 19
3.14 Natjam-R model message passing overview 19
3.15 Natjam-R code template 23

4.1 Mean number of deadline misses for a scenario in favor of EDF 29
4.2 Job request success rate for a scenario in favor of EDF 29
4.3 AM job success rate for a scenario in favor of EDF 30
4.4 Mean number of deadline misses for a scenario unfavorable to EDF . . . 31
4.5 Job request success rate for a scenario unfavorable to EDF 31
4.6 AM job success rate for a scenario unfavorable to EDF 32
4.7 Mean number of deadline misses for a scenario unfavorable to FIFO . . . 34
4.8 Job request success rate for a scenario unfavorable to FIFO 34
4.9 AM job success rate for a scenario unfavorable to FIFO 35

4.10 Mean number of deadline misses for a scenario unfavorable to MDF . . . 37
4.11 Job request success rate for a scenario unfavorable to MDF 37
4.12 AM job success rate for a scenario unfavorable to MDF 38
4.13 Mean number of deadline misses for a scenario in favor of Priority Queue 40
4.14 Job request success rate for a scenario in favor of Priority Queue 41
4.15 AM job success rate for a scenario in favor of Priority Queue 41
4.16 Mean number of deadline misses for a scenario unfavorable to Priority Queue 42
4.17 Job request success rate for a scenario unfavorable to Priority Queue . . . 43
4.18 AM job success rate for a scenario unfavorable to Priority Queue 43
4.19 Mean number of deadline misses for a scenario in favor of Natjam-R MDF 47
4.20 Job request success rate for a scenario in favor of Natjam-R MDF 47
4.21 AM job success rate for a scenario in favor of Natjam-R MDF 48
4.22 Mean number of checkpoints for a scenario in favor of Natjam-R MDF . . 49
4.23 Mean number of deadline misses for a scenario close to being in favor of Natjam-R MLF 50
4.24 Job request success rate for a scenario close to being in favor of Natjam-R MLF 51
4.25 AM job success rate for a scenario close to being in favor of Natjam-R MLF 51
4.26 Mean number of checkpoints for a scenario close to being in favor of Natjam-R MLF 52

B.1 Code sample of an EDF dispatch policy model generated by ReGen, part 1 69
B.2 Code sample of an EDF dispatch policy model generated by ReGen, part 2 70
B.3 Code sample of an EDF dispatch policy model generated by ReGen, part 3 71
B.4 Code sample of an EDF dispatch policy model generated by ReGen, part 4 72
B.5 Code sample of an EDF dispatch policy model generated by ReGen, part 5 73
B.6 Code sample of an EDF dispatch policy model generated by ReGen, part 6 74
B.7 Code sample of an EDF dispatch policy model generated by ReGen, part 7 74

List of Tables

4.1 Common parameters to the dispatch policies 27
4.2 Parameters for scenario in favor of EDF 28
4.3 Results for scenario in favor of EDF 28
4.4 Parameters for a scenario unfavorable to EDF 28
4.5 Results for a scenario unfavorable to EDF 30
4.6 Parameters for a scenario unfavorable to FIFO 33
4.7 Results for a scenario unfavorable to FIFO 33
4.8 Parameters for a scenario unfavorable to MDF 36
4.9 Results for a scenario unfavorable to MDF 36
4.10 Parameters for a scenario in favor of Priority Queue 39
4.11 Results for a scenario in favor of Priority Queue 39
4.12 Parameters for a scenario unfavorable to Priority Queue 40
4.13 Results for a scenario unfavorable to Priority Queue 42
4.14 Summary of results for the dispatch policies 44
4.15 Common parameters between the Natjam-R policies 45
4.16 Parameters for a scenario in favor of Natjam-R MDF 46
4.17 Results for a scenario in favor of Natjam-R MDF 46
4.18 Parameters for a scenario close to being in favor of Natjam-R MLF 46
4.19 Results for a scenario close to being in favor of Natjam-R MLF 48
4.20 Summary of results for the Natjam-R eviction policies 49

Chapter 1

Introduction

Clouds are becoming increasingly prevalent in our society. They are however, large, expensive and complex systems which are hard to benchmark and study. Re-configuring parameters for differing workload patterns on an entire cluster is both tedious and time consuming, and not easily repeatable. A preferred way is the use of simulations to tune systems and experiment with different setups before deployment. Current cloud simulators can calculate a number of things such as cost, energy usage, performance, utilization, response time, resource allocation, auction based mechanisms or simulate workflow and more but do not compute deadline misses nor support preemption (Calheiros, Ranjan, Beloglazov, Rose, & Buyya, 2010), (Wickremasinghe, Calheiros, & Buyya, 2010), (Kliazovich, Bouvry, & Khan, 2012), (Núñez et al., 2012), (Lim, Sharma, Nam, Kim, & Das, n.d.), (Garg & Buyya, 2011), (Casanova, Giersch, Legrand, Quinson, & Suter, 2014), (Buyya & Murshed, 2002), (Bell et al., 2002), (Dumitrescu & Foster, 2005), (Chen & Deelman, 2012), (Bux & Leser, 2013), (Frey & Hasselbring, 2011).

To address this problem we introduce the software ReGen (Rebeca Generator) which uses an example of a Big Data cluster to compute deadline misses, job request success rates, job success rates, dropped jobs, job completions, breakdown of where deadline misses occur and the priority of those jobs. For preemption we add computation for number of checkpoints, checkpoint overflows and deadlines remaining of completed jobs. It also compares the efficiency of different policies for dispatching jobs (EDF, MDF, FIFO, Priority Queue), and for evicting jobs (MDF, MLF).

For engineers and scientists studying the effects of different policies on different workloads, we support job arrival and length patterns of nondeterministic, uniform, wave, ascending and descending. Additionally, bursty job arrival, and exponential job length patterns are available.

Users can set parameters such as number of workers, queue sizes, number of simulations and their lengths, rate of high priority jobs and their lengths, a checkpoint overhead, as well as parameters for the workload. The software then runs a Monte Carlo simulation which outputs charts that show how the policies selected perform comparatively as the number of workers grows, along with the average for every size of the cluster.

1.1 Contribution

This thesis attempts to engage the problem of scheduling and deadlines in distributed systems in presence of eviction and proposes an effective and easy method of writing software for it.
These contributions are:

- Resource management software for measuring performance of scheduling policies under different workloads in a distributed system
- Experimental results demonstrating the effectiveness of the software

1.2 Overview of the Thesis

The thesis is structured as follows: Chapter 2 introduces the software and methods, such as the Actor Model, behind ReGen. Chapter 3 explains the ReGen architecture, usage, and methods to avoid concurrency problems used in the project. Chapter 4 contains the results of experiments run by ReGen, their setups and parameters, both for dispatch and eviction policies. The results are ordered into categories of scenarios in favor of, and unfavorable to each policy, both dispatch and eviction. In Chapter 5 we draw conclusions from the results and discuss them further.

Chapter 2

Background

Timed Rebeca is an extension of Rebeca (*Reactive Objects Language*) that includes timing (Aceto et al., 2014). Rebeca is an actor-based language that can be used for modeling distributed and asynchronous systems with timing constraints. The actor model (Hewitt, 1972) is a model of concurrent computation whose universal primitives are actors. An actor in Rebeca can make local decisions, send more messages, and determine how to respond to the next message it receives each time it receives a message. Messages to actors in Timed Rebeca are stored in a "bag" instead of in a separate message queue for each actor. The messages are selected to run from the bag nondeterministically for each time unit. Rebeca is an operational interpretation of actors with java-like syntax, model-checking software and formal semantics. Timed Rebeca adds extensions to Rebeca for computation time, message delivery time, message expiration, and for periods of occurrences of events which is convenient for analyzing the performance of scheduling policies over time. Figure 2.1 shows the syntax for Timed Rebeca and Figure 2.2 shows an example of a ticket service model (Khamespanah, Sabahi-Kaviani, Khosravi, Sirjani, & Izadi, 2012).

MapReduce is a programming model for generating and processing large data sets (Dean & Ghemawat, 2004). Its associated implementation allows users to specify a map function that processes a key/value pair into a set of intermediate key/value pairs. A user specified reduce function then merges all the intermediate values associated with the same intermediate key into a final result.

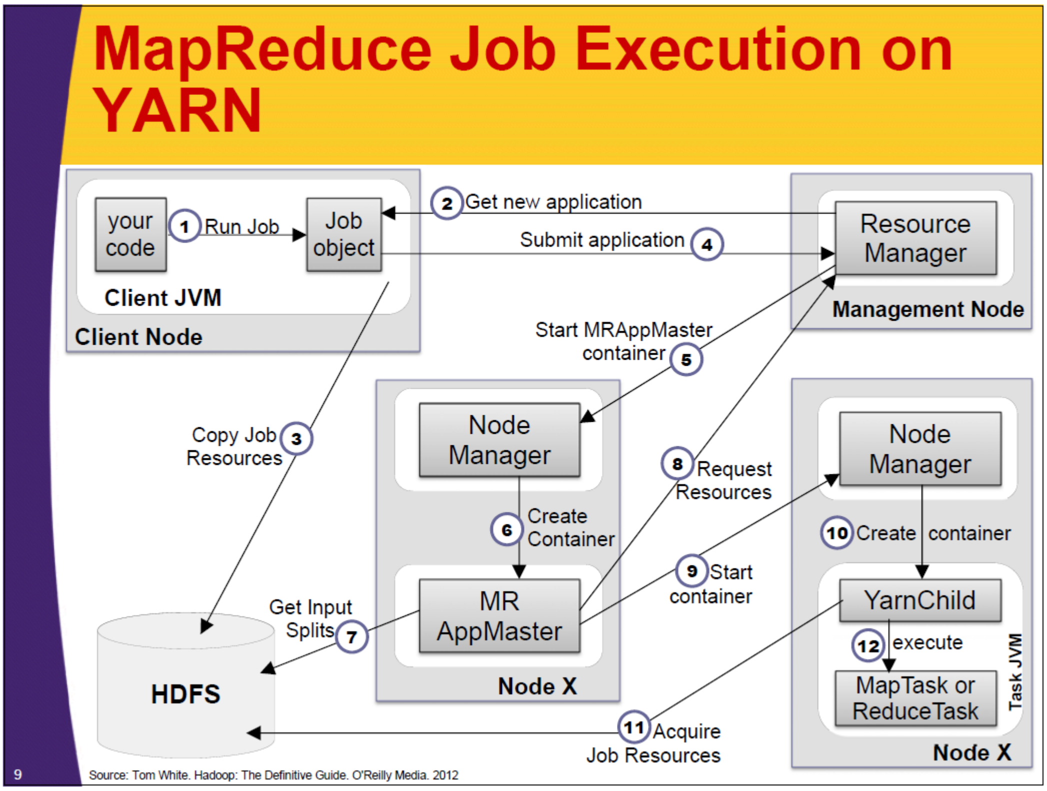
Hadoop (*http://hadoop.apache.org/*) is a framework for MapReduce, and YARN (Yet Another Resource Negotiator) is a part of Hadoop (White, 2012). YARN leaves the re-

$$
\begin{aligned}
\mathit{Model} &::= \mathit{Class}^* \ \mathit{Main}\\
\mathit{Main} &::= \textbf{main} \ \{ \ \mathit{InstanceDcl}^* \ \}\\
\mathit{InstanceDcl} &::= \mathit{className} \ \mathit{rebecName}(\langle \mathit{rebecName}\rangle^*) : (\langle \mathit{literal}\rangle^*);\\
\mathit{Class} &::= \textbf{reactiveclass} \ \mathit{className} \ \{ \ \mathit{KnownRebecs} \ \mathit{Vars} \ \mathit{MsgSrv}^* \ \}\\
\mathit{KnownRebecs} &::= \textbf{knownrebecs} \ \{ \ \mathit{VarDcl}^* \ \}\\
\mathit{Vars} &::= \textbf{statevars} \ \{ \ \mathit{VarDcl}^* \ \}\\
\mathit{VarDcl} &::= \mathit{type} \ \langle v\rangle^+;\\
\mathit{MsgSrv} &::= \textbf{msgsrv} \ \mathit{methodName}(\langle \mathit{type} \ v\rangle^*) \ \{ \ \mathit{Stmt}^* \ \}\\
\mathit{Stmt} &::= v = e; \mid \mathit{Call}; \mid \mathit{if} \ (e) \ \{ \ \mathit{Stmt}^* \ \} \ [\mathit{else} \ \{ \ \mathit{Stmt}^* \ \}] \mid \textbf{delay}(t);\\
\mathit{Call} &::= \mathit{rebecName}.\mathit{methodName}(\langle e\rangle^*) \ [\textbf{after}(t)] \ [\textbf{deadline}(t)]
\end{aligned}
$$

Figure 2.1: Abstract syntax of Timed Rebeca. Superscript * is for repetition zero or more times and superscript + for at least once. Angle brackets ⟨...⟩ are used as meta parenthesis and text within normal brackets [...] is optional. Identifier *e* denotes an (arithmetic, boolean, or nondeterministic choice) expression, *v* denotes a variable, and *t* stands for time.

```
reactiveclass TicketService {
  knownrebecs {
    Agent a;
  }
  statevars {
    int issueDelay;
  }
  msgsrv initial(int myDelay) {
    issueDelay = myDelay;
  }
  msgsrv requestTicket() {
    delay(issueDelay);
    a.ticketIssued(1);
  }
}

reactiveclass Agent {
  knownrebecs {
    TicketService ts;
    Customer c;
  }
  msgsrv requestTicket() {
    ts.requestTicket()
        deadline(5);
  }
  msgsrv ticketIssued(byte id) {
    c.ticketIssued(id);
  }
}

reactiveclass Customer {
  knownrebecs {
    Agent a;
  }
  msgsrv initial() {
    self.try();
  }
  msgsrv try() {
    a.requestTicket();
  }
  msgsrv ticketIssued(byte id) {
    self.try() after(30);
  }
}

main {
  Agent a(ts, c):();
  TicketService ts(a):(3);
  Customer c(a):();
}
```

Figure 2.2: A Timed Rebeca model of a ticket service system. There are three reactiveclasses (actors) each with their own methods (messageservers) to process messages. In the main function in this model there is a single instance (rebec) being created of each reactiveclass although multiple instances can be created in Rebeca.



MapReduce Job Execution on YARN

Source: Tom White. Hadoop: The Definitive Guide. O'Reilly Media. 2012

Figure 2.3: Overview of Hadoop's YARN. In ReGen everything but the AppMaster and the Resource Manager has been abstracted away. The Resource Manager in ReGen generates jobs instead of the Client and AppMasters run jobs without splitting them into tasks which would require another actor to run Task JVMs.

sponsibilities of job scheduling and task progress monitoring (doing task bookkeeping, keeping track of tasks, maintaining counter totals, and restarting failed or slow tasks) to a Resource Manager (RM). An Application Master (AM) negotiates with the RM for resources to manage the lifecycle of applications like MapReduce jobs running on the cluster. On a cluster there is a single RM, for every job there is a single AM, and jobs can be made up of many tasks. YARN can use different policies for dispatching jobs to AMs based on things like deadlines, priorities and arrival times. At this time, YARN does not support preemption. An overview diagram of YARN is shown in Figure 2.3.

A dual priority setting is common for jobs in MapReduce clusters: high priority (production) and low priority (research) jobs (Cho et al., 2013). A popular approach is using separate clusters for each priority which is both expensive and inefficient. Natjam attempts to remedy this by using the same cluster for both priorities and preempting jobs as needed. Natjam-R adds support for prioritized scheduling to YARN using hard deadlines.

ReGen (Rebeca Generator) is a Java Application that uses Timed Rebeca to generate Hadoop YARN models with different policy, job arrival pattern, and job length pattern parameters, runs and gathers results. It was written specifically for this project to compare the efficiency of different dispatch and job eviction policies in YARN clusters with Natjam-R. ReGen does this by creating models of YARN with its Resource Manager distributing jobs to AppMasters or preempting jobs already running. ReGen then runs a Monte Carlo simulation using the models independently of a cluster and gathers results from traces left on the hard drive. They can demonstrate how large a cluster is needed under small workloads, as well as which dispatch or eviction policies are preferred.

Chapter 3

Methods

3.1 Example Hadoop MapReduce and ReGen Scenario

A MapReduce job runs in two phases. First, a map function maps the data into key/value pairs. An example could be weather data and a job would be to find the highest temperature of each year. In this case the map function would parse the data and organize year and temperature into pairs where years are the keys and temperatures the values. In the second phase the reduce function seeks out the highest temperature for each year and returns new key/value pairs where the keys are again the years and the value now the highest temperature.

The way this works in Hadoop's YARN (Yet Another Resource Negotiator) is that a Client JVM running on a Client Node sends a MapReduce job request to the Resource Manager (RM) that is running on a Management Node, of which there is only one in a cluster. To fully use the cluster, either the clients or the RM can be set to split jobs into tasks depending on their sizes and the size of the containers running on the third type of nodes, the Node Manager nodes. They run containers whose size is fixed by administrators and is usually about 1GB of memory. In these containers AppMasters (AMs) that manage MapReduce jobs and tasks, or Task JVMs that run the tasks can be started. If jobs are small enough, the overhead of allocating and running them in other containers can outweigh running them in parallel. The AMs can instead run them sequentially in their own containers, in which case they are referred to as "uber tasks". Administrators can set the size of jobs to be "uberized". ReGen currently assumes all jobs to be uber tasks so tasks are not being modeled, only jobs. The jobs are also assumed to be any kind of job so the only difference between them is their length which does not change from the time the job enters the incoming queue.

So, in our weather data example, the RM would find an empty container in the cluster to start an AM to run the job. The AM would send the RM a request for containers to run tasks if the job is not an uber task and the RM would reply with that information. The AM would then message the Node Managers where the free containers are and they start the Task JVMs which run the tasks. In ReGen, negotiations like requests for resources, entities like Client nodes and Node Managers, containers like Task JVMs have been abstracted away, leaving only two actors, the RM and AM. Figure 3.12 shows the simplified model of YARN (Figure 2.3) used by ReGen.

In ReGen when a user has selected the policies and configured the workload of a simulation through the user interface, there is a model created for each number of AMs in the cluster being simulated, starting with one AM and up to a user-defined maximum. A user-defined number of simulations is then run for each of the models and all of them with the same workload. The results from the simulations will demonstrate how a cluster of each size handles the workload with each policy. The way the models work is by the RM automatically generating jobs into the incoming queue instead of a Client actor sending jobs like in YARN. The RM explores the incoming queue every time unit for the next job to run using a dispatch policy, finds an AM to run the job and sends it to him. The AM then runs the job and sends the RM its result when it ends either by completion or by missing the deadline. As all jobs are uber tasks in ReGen, jobs are completed by simply decrementing an integer that represents the jobs length, and the same goes for the deadline.

3.2 ReGen Architecture, Diagrams and User Interface

ReGen is a Model-View-Controller Java application that generates Timed Rebeca code based on user-defined parameters and is about 4000 lines of code. Users have a choice between two different policy types for the Resource Manager (RM), dispatch and eviction. The dispatch policies are for prioritizing which job to run next from the RM's incoming job queue and the eviction policies are for selecting which job running at the AppMasters (AMs) to evict when performing preemption. Currently ReGen supports Earliest Deadline First (EDF), First-In-First-Out (FIFO), Maximum Deadline First (MDF) and Priority Queue dispatch policies, and Maximum Deadline First (MDF) and Maximum Laxity First (MLF) eviction policies. For any policy there are different job arrival (bursty, nondet, wave, uniform, ascending, descending) and job length patterns (exponential, nondet, wave, uniform, ascending, descending). Figures 3.1 to 3.7 show diagrams of the currently implemented job arrival and length patterns. The workload of a cluster is defined as a

combination of a job arrival pattern and a job length pattern.
The following steps are performed by ReGen using the parameters the user selects, visible in Figure 3.8:

1. Directories for artifacts are created.

2. The code for each model is generated into a template. For each combination of policy, job arrival and length pattern there is one model for each number of AMs and each model is one file.

3. Batch files to run each combination of policy, job arrival and length are created.

4. All batch files are run. A batch file compiles the Rebeca code previously generated into C++ code using the Rebeca Model Checker, compiles the C++ code with a C++ compiler (G++ was used), and runs a Monte Carlo simulation with the executable generated from the C++ code. The traces of the simulations are saved on the hard drive.

5. The traces are parsed, results are aggregated and written out in charts and text files.

This sequence is demonstrated in Figure 3.9, the classes used in Figure 3.10, and all relevant components in Figure 3.11.
For a sample of code generated by ReGen, refer to Appendix B.

3.2.1 ReGen Classes

The following are the classes used as shown in the class diagram in Figure 3.10:

- DeadlineModelView
- DeadlineModelParameters
- AbstractModelCharter
- AbstractModelGenerator
- DeadlineModelController
- DispatchModelGenerator
- DispatchModelCharter
- NatjamModelGenerator
- NatjamModelCharter
- DeadlineTracesParser
- NatjamTracesParser

The view can create multiple controllers simultaneously so users can run concurrent simulations. The parameters are passed from the view to the controller which creates the model generator. The generator generates the models and batches which the controller then runs. There is one batch for each policy and the batches are run concurrently. After running them, the controller creates a charter which parses the traces using a parser and writes charts from the results.
The abstract classes contain functionality shared between the different charters and generators.

3.3 Dispatch Policies

3.3.1 Dispatch Model Overview

A dispatch policy (Earliest Deadline First, First-In-First-Out...) controls how queued jobs are dispatched by for processing. ReGen generates a model (Figure 3.12) for this that has two actors, ResourceManager (RM) that uses the policy and AppMaster (AM) that receives the jobs. There is only one RM but there can be many AMs. There is no job

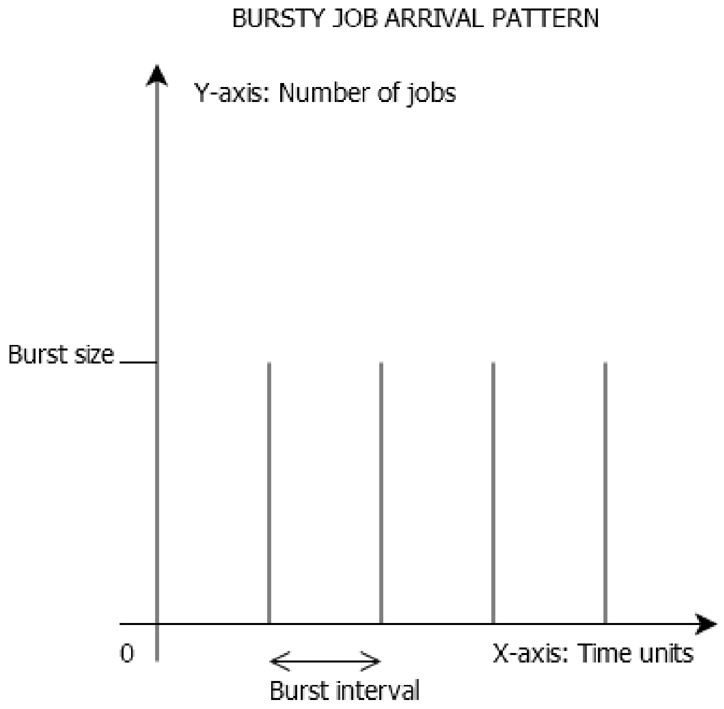


Figure 3.1: The bursty job arrival pattern selectable in ReGen. Starting at time 0, jobs arrive continuously in fixed bursts with a fixed interval. The size and interval can be set by the user.

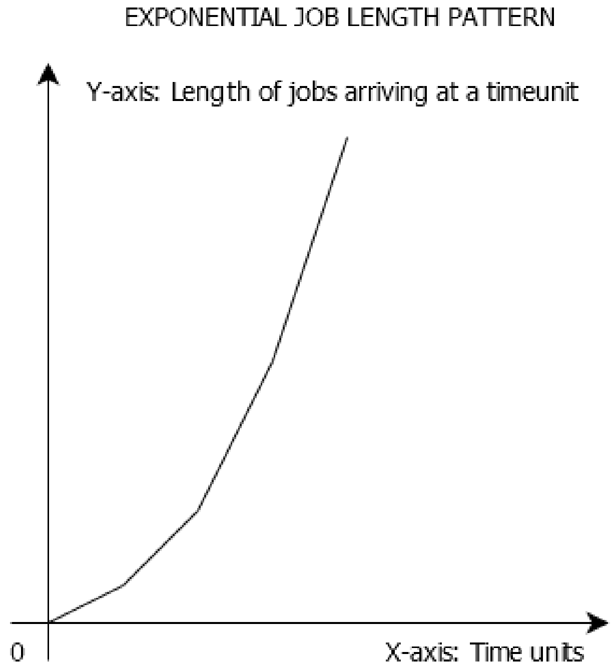


Figure 3.2: The exponential job length pattern selectable in ReGen where job length grows exponentially and is a function of time. The user can set the multiplier used to multiply the jobs arrival time unit such that $length = now * multiplier$.

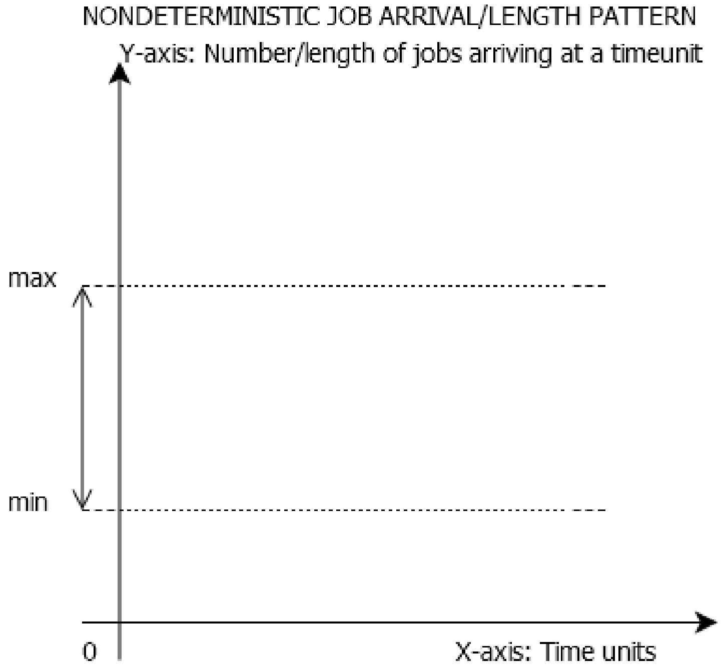

NONDETERMINISTIC JOB ARRIVAL/LENGTH PATTERN
Y-axis: Number/length of jobs arriving at a timeunit
max
min
0
X-axis: Time units

Figure 3.3: The nondeterministic pattern selectable in ReGen where the number or length of jobs is a nondeterministic value between a minimum and a maximum every time unit. The user sets the minimum and maximum values.

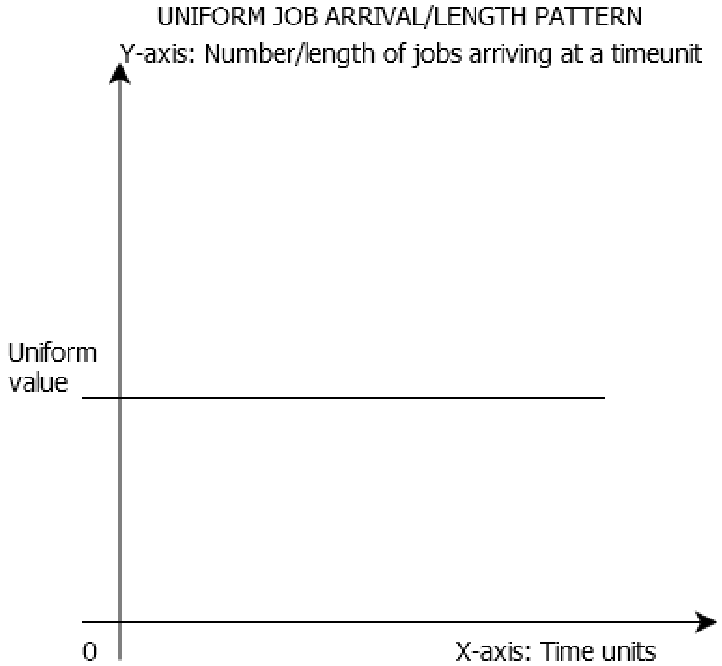

UNIFORM JOB ARRIVAL/LENGTH PATTERN
Y-axis: Number/length of jobs arriving at a timeunit
Uniform value
0
X-axis: Time units

Figure 3.4: The uniform pattern selectable in ReGen where users can set a uniform value for both job arrivals and lengths.

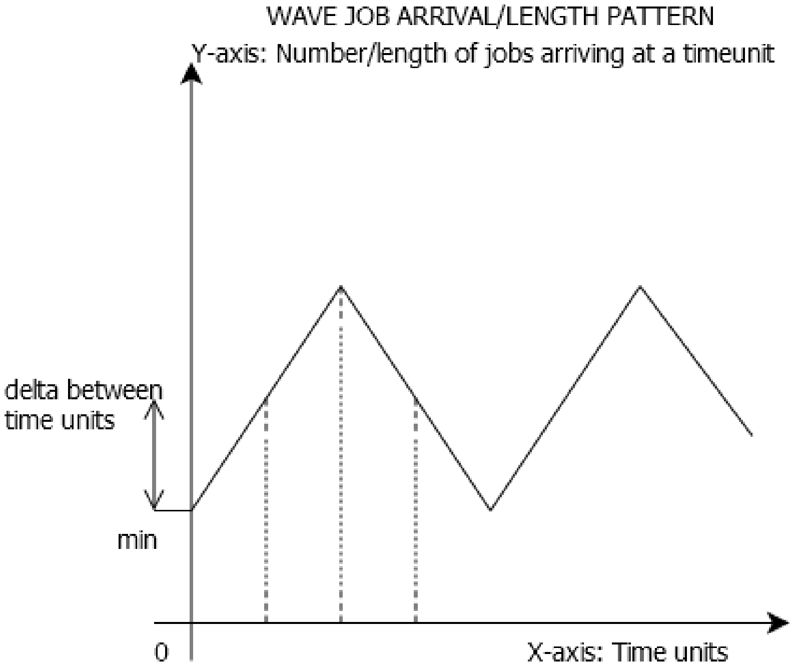

WAVE JOB ARRIVAL/LENGTH PATTERN
Y-axis: Number/length of jobs arriving at a timeunit
delta between time units
min
0
X-axis: Time units

Figure 3.5: The pattern selectable in ReGen where the number of jobs arriving or their lengths follow a wave. Users select a minimum for the wave, the difference on the wave between each time unit, and how many time units are on a single part of the wave which is then repeated. The figure shows a wave with four points, starting at time 0.

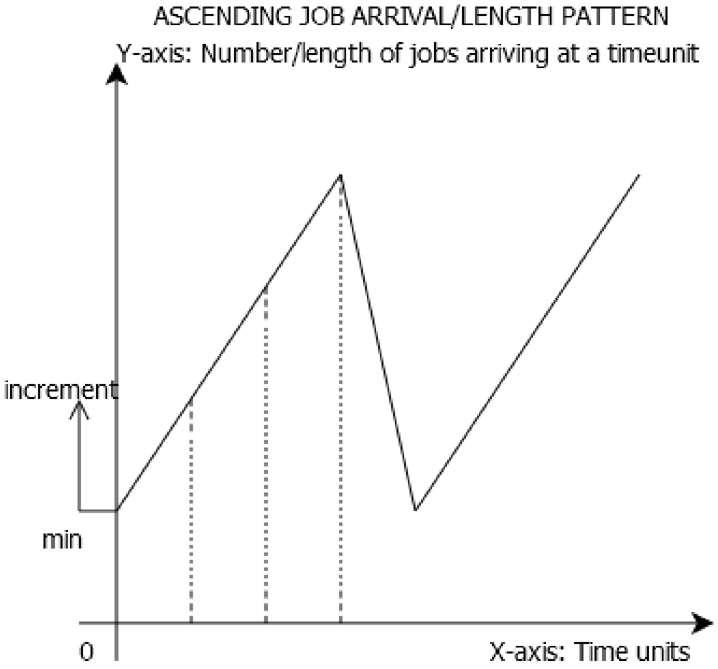

ASCENDING JOB ARRIVAL/LENGTH PATTERN
Y-axis: Number/length of jobs arriving at a timeunit
increment
min
0
X-axis: Time units

Figure 3.6: The ascending pattern selectable in ReGen where users input a minimum value, an increment between time units, and how many points or time units the ascension should last. The pattern then repeats as shown in the figure which is a four point ascension.

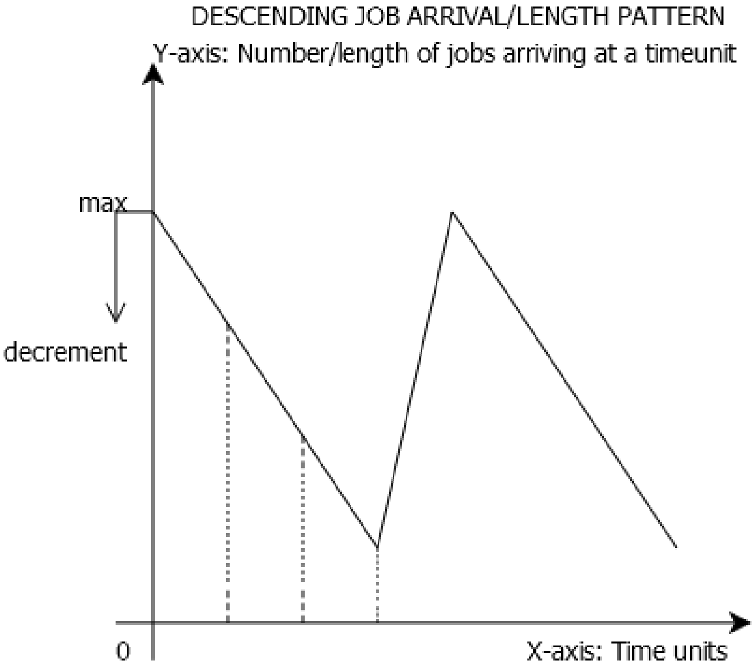


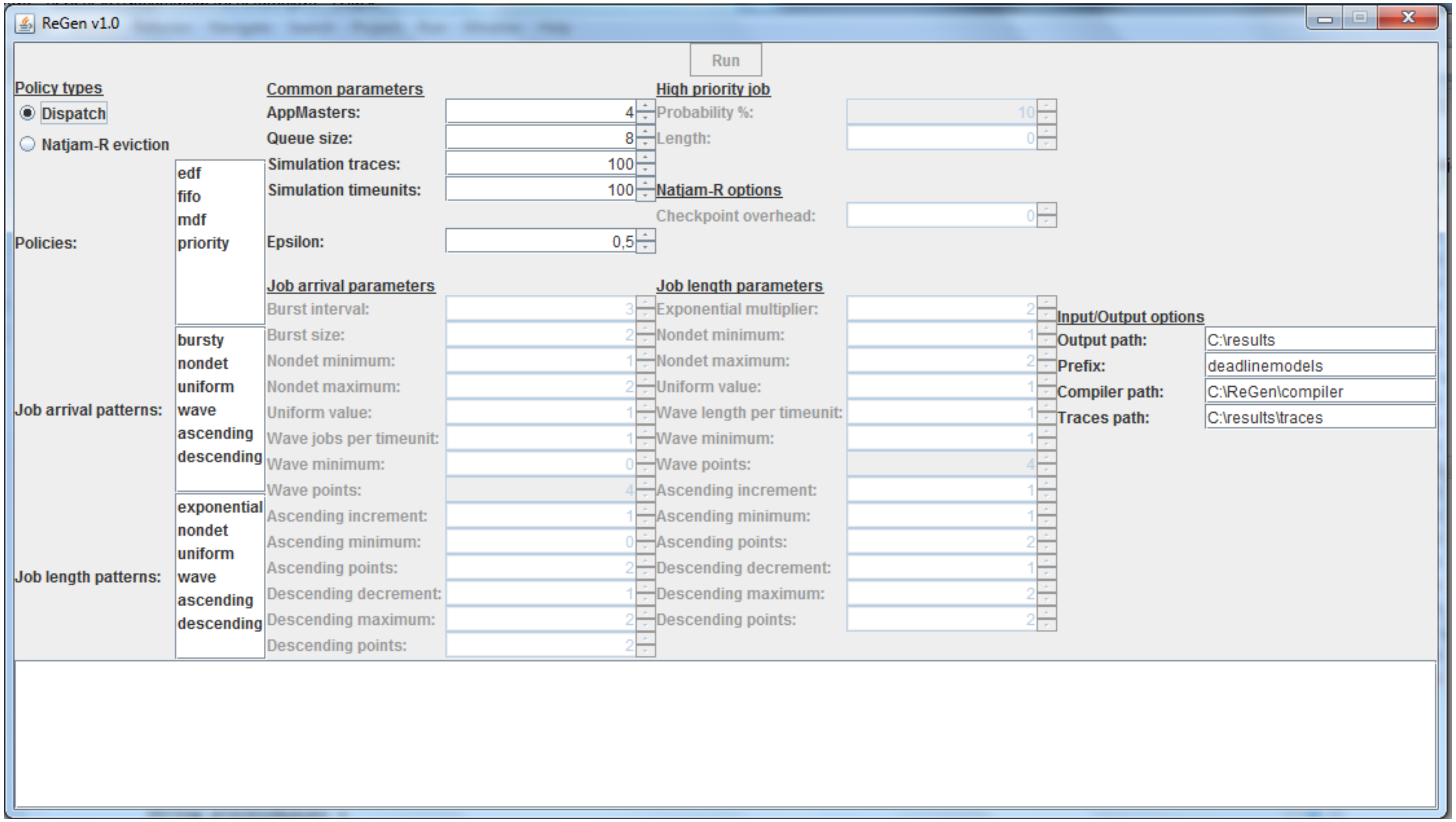
Figure 3.7: The descending pattern in ReGen. For the descent users select a maximum value to descend from, the decrement per time unit and the number of time units to descend before repeating the pattern from the maximum. The figure shows a descent over four time units.

entity modeled so the RM maintains its incoming queue as separate queues with deadlines for jobs in one, and lengths in another. If jobs have priorities, a third queue is added.

Every time unit the RM checks the deadline queue for jobs whose deadlines have run out, then counts them as misses and removes them. Next, the RM checks whether any AMs are free and if so, dispatches jobs to them using the policy.

After dispatching, the RM generates new jobs automatically instead of receiving them from a client actor using different job arrival patterns. We abstracted the client away from the model as it does not change the overall result but will generate more states. After deciding the number of new jobs, the RM decides their lengths using different job length patterns. If a job has high priority, its length will be determined after the lengths of all low priority jobs have been determined. This is in case high and low priority jobs do not follow the same length pattern. The deadline is then computed for each job as $deadline = job\,length * (1 + epsilon)$ where *epsilon* is user defined. Job length is determined using the preselected job length pattern.

The RM maintains queues with AMs status and sets them as busy when jobs are sent to them. When an AM receives a job it will be busy until either the deadline runs out or the job finishes, whichever comes first. Once either happens, the AM sends a message to the RM which counts whether the deadline ran out or the job is completed, and sets the AM



ReGen v1.0

Run

Policy types

● Dispatch

○ Natjam-R eviction

Policies: edf, fifo, mdf, priority

Job arrival patterns: bursty, nondet, uniform, wave, ascending, descending

Job length patterns: exponential, nondet, uniform, wave, ascending, descending

Common parameters

AppMasters: 4

Queue size: 8

Simulation traces: 100

Simulation timeunits: 100

Epsilon: 0,5

Job arrival parameters

Burst interval: 3

Burst size: 2

Nondet minimum: 1

Nondet maximum: 2

Uniform value: 1

Wave jobs per timeunit: 1

Wave minimum: 0

Wave points: 4

Ascending increment: 1

Ascending minimum: 0

Ascending points: 2

Descending decrement: 1

Descending maximum: 2

Descending points: 2

High priority job

Probability %: 10

Length: 0

Natjam-R options

Checkpoint overhead: 0

Job length parameters

Exponential multiplier: 2

Nondet minimum: 1

Nondet maximum: 2

Uniform value: 1

Wave length per timeunit: 1

Wave minimum: 1

Wave points: 4

Ascending increment: 1

Ascending minimum: 1

Ascending points: 2

Descending decrement: 1

Descending maximum: 2

Descending points: 2

Input/Output options

Output path: C:\results

Prefix: deadlinemodels

Compiler path: C:\ReGen\compiler

Traces path: C:\results\traces

Figure 3.8: The ReGen user interface. The leftmost side of the UI is where policies, job arrival and length patterns are selected. The center shows the options available for them, and rightmost for compiling and outputting the results.

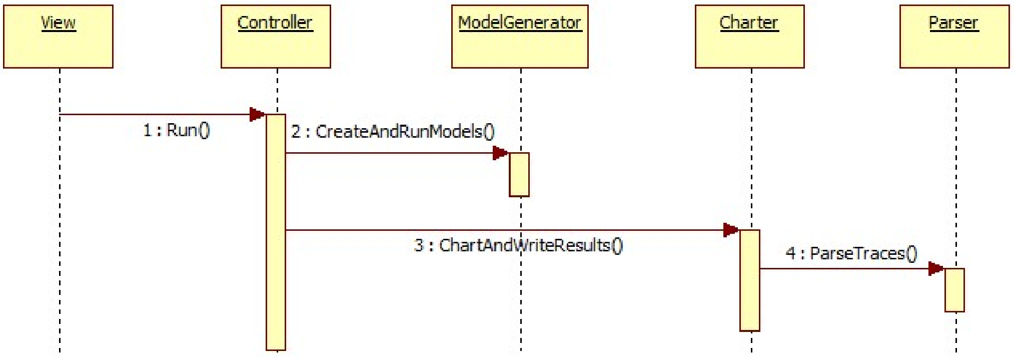


Figure 3.9: ReGen Sequence diagram. The view starts controllers as threads so they can run concurrently. The controller creates the model generator that generates the models and batch files based on the parameters selected in the view. The batches and models are run and the controller waits for them to finish. Once that is done a charter is created that creates a parser that parses the traces from the simulations. The charter then aggregates the results from the parser, writes out charts and text files and terminates.

as free. Jobs are assumed to run on only one AM at a time. Figure 3.13 shows the code template used for the dispatch models.

3.4 Natjam-R

3.4.1 Natjam-R Model Overview

Natjam is built into Hadoop's YARN and supports priority based preemption. Natjam-R is an extension that adds support for deadline based preemption. ReGen can generate models of Natjam-R and currently supports two different policies, Maximum Deadline First and Maximum Laxity First, that are used to select jobs to evict when preemptions occur. ReGen also supports priority based preemption but the experimental results in the next chapter are only for deadlines.

The Natjam model (Figure 3.14) has two actors, ResourceManager (RM) and AppMaster (AM). There is only one RM but there can be many AMs. The message passing layer is the same as for the dispatch policy model except for two additional messages: a *checkpoint* message sent from the AM to the RM when preemption occurs, and the AMs run a *process* message to process jobs every time unit instead of processing jobs in one run until the end. This is so the AMs can check for new *runJob* messages from the RM, in which case they need to preempt their current jobs.

So, for managing messaging the RM now maintains the following queues:

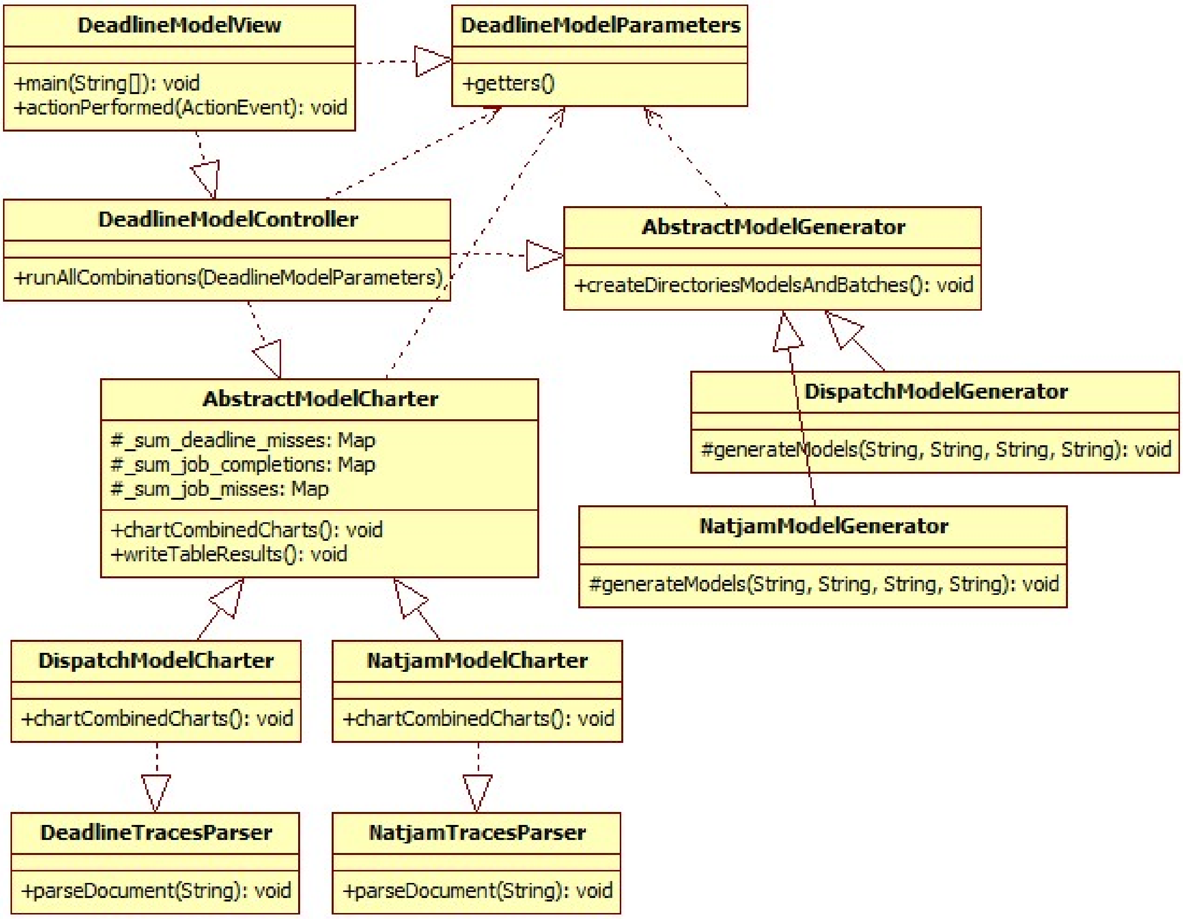

DeadlineModelView
+main(String[]): void
+actionPerformed(ActionEvent): void
DeadlineModelParameters
+getters()
DeadlineModelController
+runAllCombinations(DeadlineModelParameters)
AbstractModelGenerator
+createDirectoriesModelsAndBatches(): void
DispatchModelGenerator
#generateModels(String, String, String, String): void
NatjamModelGenerator
#generateModels(String, String, String, String): void
AbstractModelCharter
#_sum_deadline_misses: Map
#_sum_job_completions: Map
#_sum_job_misses: Map
+chartCombinedCharts(): void
+writeTableResults(): void
DispatchModelCharter
+chartCombinedCharts(): void
NatjamModelCharter
+chartCombinedCharts(): void
DeadlineTracesParser
+parseDocument(String): void
NatjamTracesParser
+parseDocument(String): void

Figure 3.10: ReGen Class diagram. The controller and parameters are instanced by the view. The controller instances the abstract model generators and charters and all three depend on the parameters. There are two types of models available in ReGen, the dispatch policy and Natjam models and this requires different model generators, charters and parsers for each type of policy. The abstract classes contain functionality shared between the derived dispatch and Natjam classes and decide which type to instance based on the parameters from the view.

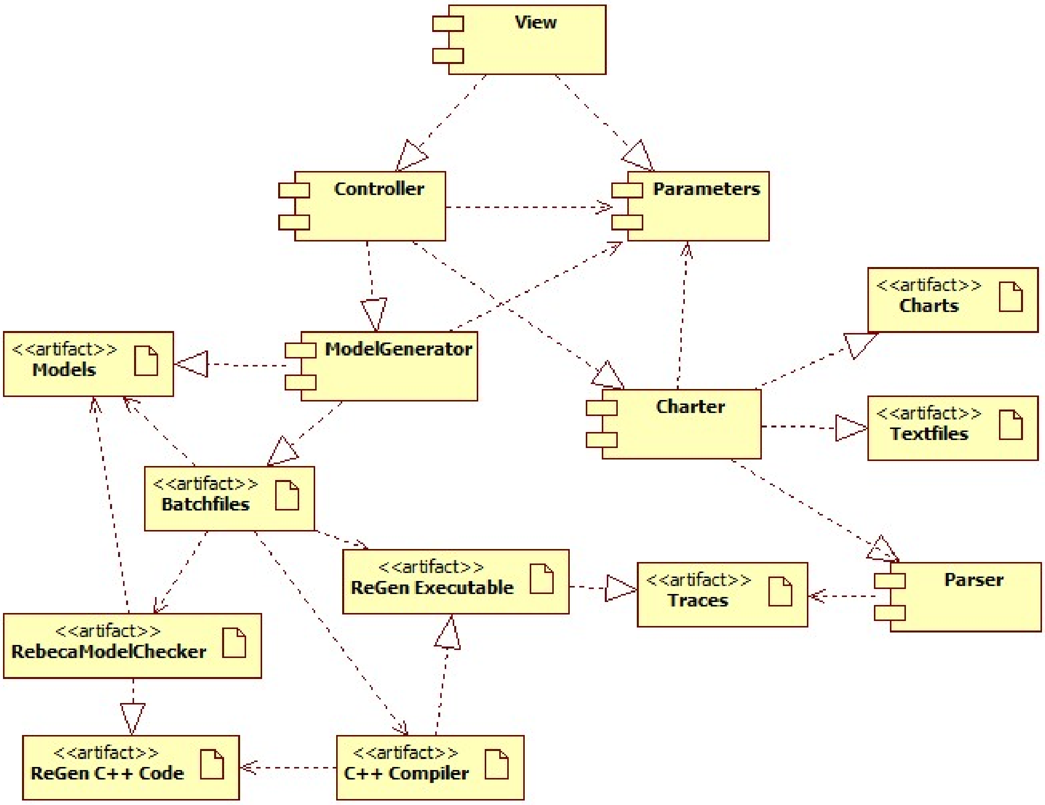


Figure 3.11: ReGen Component diagram. Here we see the same relationship between the view, controller, parameters, model generator, charter and parser as in the class diagram (Figure 3.10). In addition to that we see the artifacts relevant to the system like the models and batch files that are generated by the model generator. The batch files then depend on the models, the supplied Rebeca Model Checker, a C++ compiler (G++ was used), and the ReGen executable created by the compiler from the C++ code to generate the simulation traces. The charter then outputs the charts and text files from the results from the parser parsing the traces.

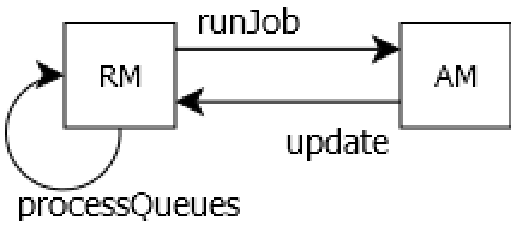
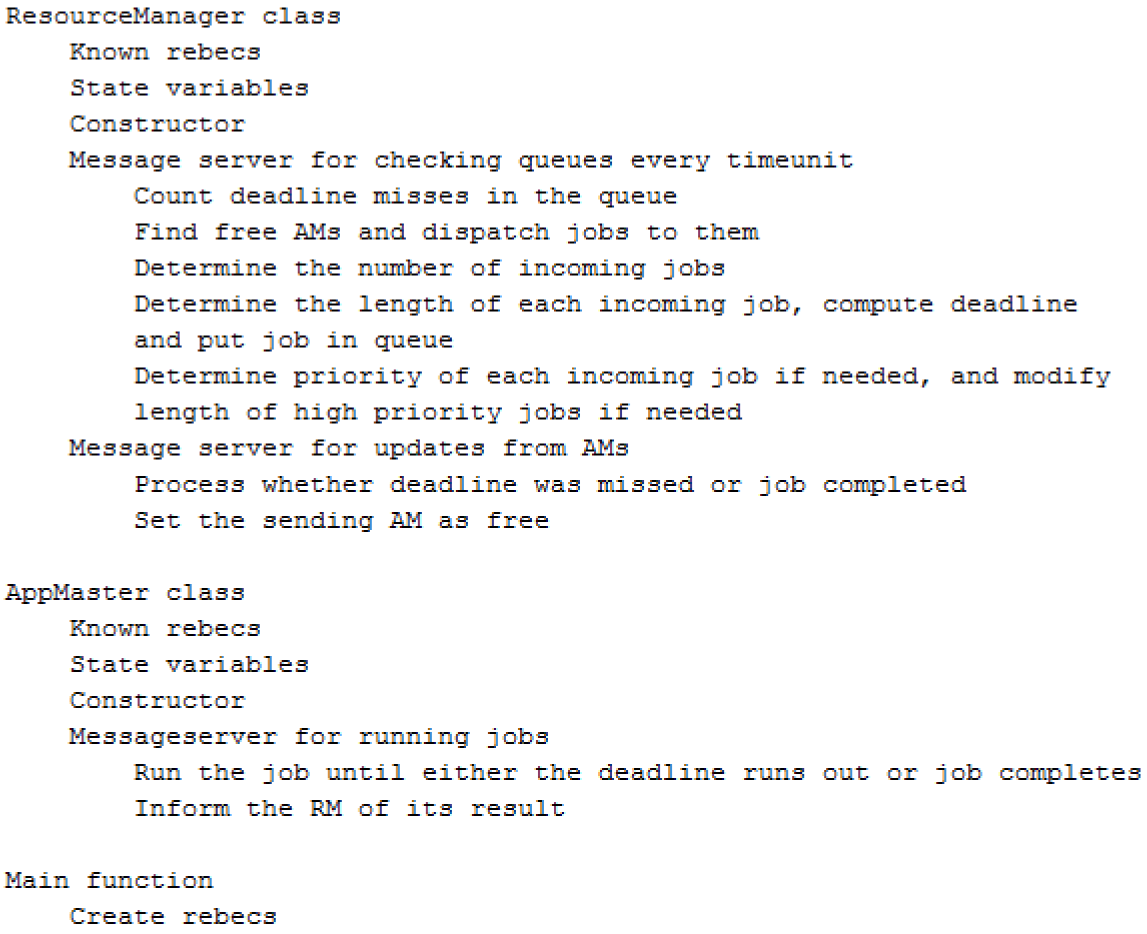


Figure 3.12: Dispatch policy model message passing overview. The RM runs a *processQueues* message every time unit that can send *runJob* messages to AMs which send *update* messages back once jobs complete or their deadlines run out.



```
ResourceManager class
    Known rebecs
    State variables
    Constructor
    Message server for checking queues every timeunit
        Count deadline misses in the queue
        Find free AMs and dispatch jobs to them
        Determine the number of incoming jobs
        Determine the length of each incoming job, compute deadline
        and put job in queue
        Determine priority of each incoming job if needed, and modify
        length of high priority jobs if needed
    Message server for updates from AMs
        Process whether deadline was missed or job completed
        Set the sending AM as free

AppMaster class
    Known rebecs
    State variables
    Constructor
    Messageserver for running jobs
        Run the job until either the deadline runs out or job completes
        Inform the RM of its result

Main function
    Create rebecs
```

Figure 3.13: The code template ReGen generates code into for creation of the models for the dispatch policies. This is done by creating code snippets into the corresponding places in the template using the parameters selected by the user in the UI.

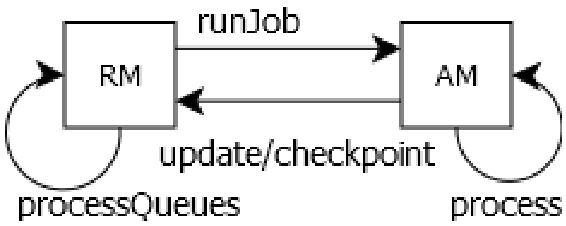


Figure 3.14: Natjam-R model message passing overview. As with the dispatch policies the RM runs a *processQueues* message every time unit. This can cause a *runJob* message to be sent to an AM that starts a job and can preempt a running one. To support preemption the AM now runs a *process* message every time unit instead of waiting until the job completes or the deadline runs out. The AM can now send either updates or checkpoints to the RM depending on whether jobs end or are preempted.

- incoming job deadlines
- incoming job lengths
- incoming job priorities
- checkpoint deadlines
- checkpoint times remaining
- deadlines remaining of jobs running at the AMs
- times remaining of jobs running at the AMs
- types of jobs running at the AMs

Every time unit using these queues, the RM performs the following steps in the below order:

1. Checks the deadlines of all jobs in its queues including the checkpoint queue. If a deadline becomes less than or equal to zero, a deadline miss is counted and the job removed from the queue. Different counters are used for research jobs (low priority) and production jobs (high priority).

2. The RM maintains counters for job status on the AMs instead of receiving periodic heartbeat messages from them to reduce the amount of messages being transmitted and simplify the model. The RM decrements these counters during this step.

3. Sorts the incoming job queue so the earliest deadlines come first.

4. Dispatches all production (high priority) jobs to empty AMs. If there are still production jobs in the queue and no free AMs, the RM preempts research (low priority) jobs based on a predetermined job eviction policy. The AM whose research job was preempted responds to the RM with a checkpoint of the preempted job and its status. Only research jobs can be preempted.

5. If there are still free AMs after all production jobs have been dispatched, the RM will dispatch checkpoints based on Earliest Deadline First (EDF) for Maximum Deadline First (MDF) job eviction policy, and Least Laxity First (LLF) for Maximum Laxity First (MLF) eviction. The laxity of a job is computed as the difference between its deadline remaining and its completion time remaining.

6. Examines the checkpoints for least laxity jobs and preempts research jobs with higher laxity if using MLF. If using MDF, the checkpoints with low deadlines will preempt running jobs with higher deadlines. Both checkpoints and checkpoint queue overflows are counted.

7. If there are still free AMs, the RM will dispatch remaining research jobs from the incoming queue based on EDF.
8. Similar to checkpoints, if there are no free AMs the RM will examine the incoming queue for least laxity jobs and preempt research jobs with higher laxity if using MLF. If using MDF, the incoming jobs with low deadlines will preempt running jobs with higher deadlines.
9. The RM maintains a mutex for each AM to prevent multiple messages being sent to one during a timeunit. During this step, the RM unlocks all mutexes.
10. After dispatching jobs, the RM will determine how many new jobs should be added to the queue, their deadlines and types. Both research and production jobs are kept in the same queue. The RM counts jobs that need to be dropped in case the queue is full.
11. Sends a *processQueues* message to itself which will be executed the next time unit and repeat all the above steps.

When a job completes or a deadline runs out at an AM, it sends a message to the RM which counts whether a miss occurred or the job completed, and updates its information on the AMs status. The RM also computes a margin for completed jobs which is the difference between the deadline remaining and time remaining when the job ended either by completing or its deadline running out.

When a checkpoint is sent to the RM, the status of the AM on the RM is updated.

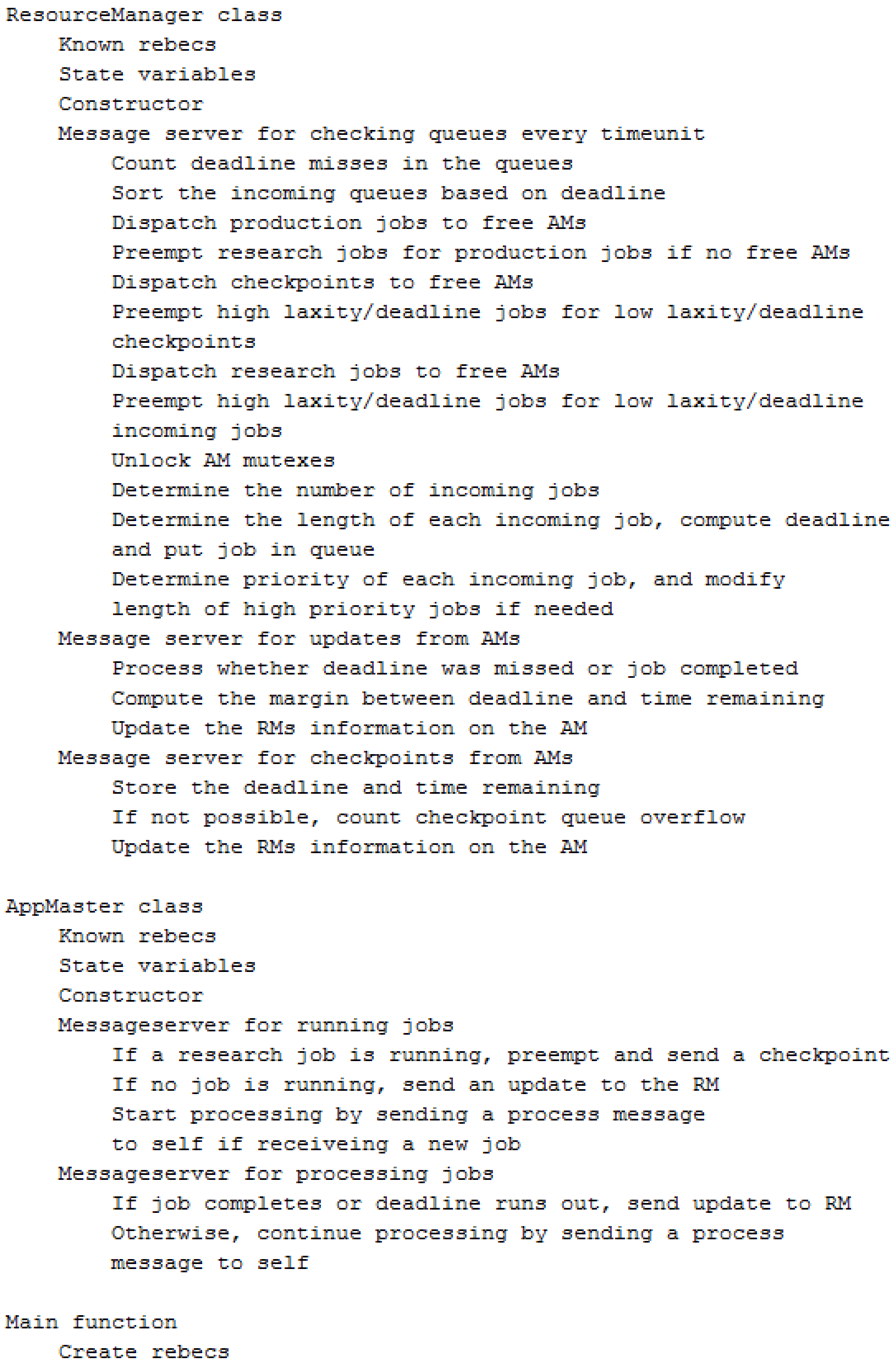
A change from the dispatch models is that AMs now check for completion of jobs every time unit instead of only when the job completes. The Hadoop default interval for checking for job completion is 5000ms (*http://hadoop.apache.org/docs/r2.3.0/api/src-html/org/apache/hadoop/mapreduce/*
Job.html#line.85).

Figure 3.15 shows the code template for the Natjam-R models.

3.4.2 Avoiding Concurrency Problems

Because of the added complexity, the Natjam-R Resource Manager (RM) now needs a mutex for each AppMaster (AM). Otherwise the RM can send multiple messages to an AM during a timeunit. The RM can for example preempt a job running at an AM for a checkpoint with lower laxity, and then preempt again for a job from the incoming queue with even lower laxity. In the current implementation, checkpoints are assumed to have higher priority than new jobs and the order of execution the RM performs every time unit goes from highest priority to lowest.

Secondly, only jobs that have more than 1 unit of time remaining of both deadline and completion are considered for preemption. Otherwise a race condition will occur between the *runJob* and *process* messages at the AM. If the *process* message goes first and there is only 1 unit of time left of either the deadline or completion, it will end, and an *update* message sent to the RM which will set the AM as free. This will lead to an error as the *runJob* message will make the AM busy and not update the AMs status on the RM which will continue to assume the AM is free.



```
ResourceManager class
    Known rebecs
    State variables
    Constructor
    Message server for checking queues every timeunit
        Count deadline misses in the queues
        Sort the incoming queues based on deadline
        Dispatch production jobs to free AMs
        Preempt research jobs for production jobs if no free AMs
        Dispatch checkpoints to free AMs
        Preempt high laxity/deadline jobs for low laxity/deadline
        checkpoints
        Dispatch research jobs to free AMs
        Preempt high laxity/deadline jobs for low laxity/deadline
        incoming jobs
        Unlock AM mutexes
        Determine the number of incoming jobs
        Determine the length of each incoming job, compute deadline
        and put job in queue
        Determine priority of each incoming job, and modify
        length of high priority jobs if needed
    Message server for updates from AMs
        Process whether deadline was missed or job completed
        Compute the margin between deadline and time remaining
        Update the RMs information on the AM
    Message server for checkpoints from AMs
        Store the deadline and time remaining
        If not possible, count checkpoint queue overflow
        Update the RMs information on the AM

AppMaster class
    Known rebecs
    State variables
    Constructor
    Messageserver for running jobs
        If a research job is running, preempt and send a checkpoint
        If no job is running, send an update to the RM
        Start processing by sending a process message
        to self if receiveing a new job
    Messageserver for processing jobs
        If job completes or deadline runs out, send update to RM
        Otherwise, continue processing by sending a process
        message to self

Main function
    Create rebecs
```

Figure 3.15: The code template used by ReGen to generate the Natjam-R models. Larger than the template for the dispatch policies but similar in structure. The Resource Manager has an additional message server for checkpoints and the AppMaster an additional one for the *process* message that processes jobs.

Chapter 4

Experiments

4.1 Test Setup

For the experiments a Dell Inspiron N5010 was used. It was running Windows 7 Home Premium 64-bit SP1 using an Intel Core i3 M330 CPU at 2.13GHz. It had 4GB of RAM and a 250GB hard drive.

Each experiment on the test setup ran in a number of minutes. Compilation time for each model is short and the traces can become large on the hard drive. The speed of the simulations can run in minutes and could possibly be shortened and the file sizes lessened by writing less to the hard drive. The final state of each simulation only for example, instead of every time unit.

4.2 Dispatch Policy Results

A Monte Carlo simulation is where a simulation is run a number of times and an average result is generated from them (Allen, 2011). In the following experiments the results are represented as the mean number of deadline misses, job request success rate and AppMaster (AM) job success rate.

- Deadline misses are the sum of the number of jobs that missed their deadlines while running and while waiting in a queue.
- Job request success rate is the ratio of jobs submitted to the Resource Manager (RM) that were completed successfully within the deadline.

- AM job success rate is the ratio of jobs submitted from the RM to the AMs that were completed within the deadline.

These numbers are represented by the Y-axis in graphs, and the X-axis shows the number of AMs in the cluster (*Concurrent jobs*). Each model can only run one amount of available AMs as Timed Rebeca does not support dynamic generation of rebecs. The results are therefore also represented by tables that show an average for every number of available AMs. It is worth noting that for every number of AMs available in the experiments, the workload is the same, so the performance of the cluster improves as the number of AMs increases. So, for every size of the cluster (number of AMs), there is a Monte Carlo simulation run and the results for that size are presented on the X-axis. The number of simulations specified is for each number of AMs in an experiment. If there are for example one, two and three AMs in a chart, a total of *3 AMs * 100 simulations per AM = 300 simulations* were run.

For each policy there are generally two types of results: a scenario in favor of and a scenario unfavorable to it. The scenario in favor of one policy can be the one unfavorable to another.

4.2.1 Common parameters to the dispatch policies

Experiments could take time and use both memory and hard drive space. The Monte Carlo error which is the difference between the mean generated by the simulations and the true mean was driven down with a high number of simulations. With the number of simulations used, an example mean of 50 and a high standard deviation of 10, gave us a 99% chance of the true mean being between 47.64 and 52.37 which is 4.73%. This example half width is computed as $(T * S)/\sqrt{n} = (2.365 * 10)/\sqrt{100} = 23.65/10 = 2.365$ where *T* is the student's *T* distribution value for 99 degrees of freedom or 100 - 1 simulations, and 99% probability, *S* is the standard deviation and *n* the number of simulations. For comparison, if we run 20 simulations the half width is $(2.539 * 10)/\sqrt{20} = 5.68$. These 20 simulations would then give us a 99% of the true mean being between $50 - 5.68 = 44.32$ and $50 + 5.68 = 55.68$. Therefore 100 simulations are giving us $47.64 - 44.32 = 3.32\% * 2 = 6.64\%$ more accuracy than 20 in this example.

The size of the experiments, or the number of AMs and time units, was decided based on the performance of the test setup, and attempted to be as large as possible. Incoming jobs were kept as one per time unit to make it easier to estimate whether the results were correct even though it was possible to use other job arrival patterns. Queue size was selected so that if all AMs finished their jobs simultaneously they could all start new jobs immediately and still be able to queue jobs.

Similar reasoning was used for the common parameters for the Natjam-R policies. Table 4.1 shows the common parameters used for the dispatch policies.

Table 4.1: Common parameters to the dispatch policies

Max AppMasters	Queue size	Simulations	Timeunits	Jobs per timeunit
6	12	100	100	1

4.2.2 Earliest Deadline First (EDF)

The Resource Manager (RM) explored the entire incoming job queue and maintained an integer with the lowest deadline found for a job. Once he had gone through the queue, he would remove the job with the lowest deadline from it, find a free AppMaster (AM) and send the job to him. The incoming queue was not kept sorted nor shifted to the front so gaps could appear between jobs. New jobs were inserted as close to the front of the queue as possible, and if multiple jobs had the same earliest deadline, the one nearest to the back of the queue was dispatched.

Scenario in favor of EDF (Tables 4.2 and 4.3, Figures 4.1 to 4.3)

In this scenario the deadlines were high and job lengths nondeterministic. In that way it was likelier that EDF was selecting a deadline that was close to but not less than job length. If the deadlines were low they were more likely to be less than the job length when the earliest deadline was sought at any time in the queue. This would be unfavorable to EDF but might have been favorable to Maximum Deadline First (MDF). A jobs deadline was computed using the formula $deadline = joblength * (1 + epsilon)$ where *epsilon* was selected by the user. That is, the deadline was computed by adding *epsilon* many jobs lengths to the jobs length.

FIFO performed the worst under the selected workload until there were four AMs in the cluster. MDF selected long deadlines and missed short jobs, and FIFO selected the oldest job from the queue and whose deadline had decreased the most. With four AMs the job from the front of the queue had a greater deadline remaining than time remaining in more cases than with fewer AMs. Priority Queue used EDF as tie breaker between jobs of the same priority. It therefore performed better than EDF because fewer low priority

Table 4.2: Parameters for scenario in favor of EDF

Parameters				
Epsilon	High priority job probability	Job length	Job length minimum	Job length maximum
2.0	10%	nondeterministic	1	6

Table 4.3: Results for scenario in favor of EDF

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
EDF	25.96	70.37%	77.20%
FIFO	31.16	64.68%	86.79%
MDF	30.55	66.52%	99.89%
PRIORITY	25.65	71.07%	81.36%

jobs whose deadlines had run out were started and high priority jobs with high deadlines remaining were run instead.

Scenario unfavorable to EDF (Tables 4.4 and 4.5, Figures 4.4 to 4.6)

In a scenario unfavorable to EDF the earliest deadline is always less than its job length so every job misses its deadline. To achieve this, *epsilon* was kept low and job length uniform.

Note how EDF and FIFO were exactly the same in this scenario because both policies were selecting the job from the front of the queue. MDF performed well in this scenario because it selected the latest job which was the least probable to have had its deadline run out. Priority Queue performed better than EDF because high priority jobs were started sooner and their deadlines therefore less likely to run out.

Table 4.4: Parameters for a scenario unfavorable to EDF

Parameters			
Epsilon	High priority job probability	Job length pattern	Job length
0.1	10%	uniform	3

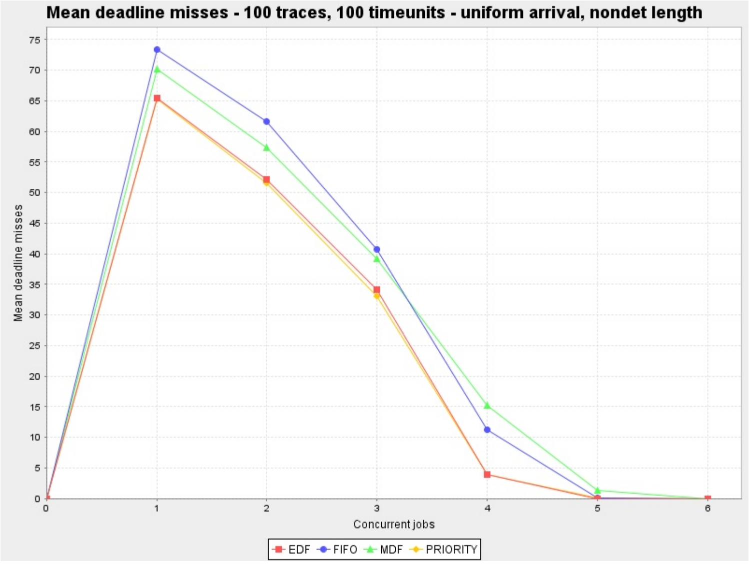


Figure 4.1: Mean number of deadline misses for a scenario in favor of EDF. Most deadline misses occurred when only one AppMaster (AM) was available but their number started dropping off as soon as more jobs could be run concurrently. EDF and Priority Queue were performing similarly because in the experiments EDF was used as a tiebreaker for same priority jobs. With about five available AMs the cluster started handling the entire workload of the experiment. First-In-First-Out (FIFO) can be seen outperforming Maximum Deadline First (MDF) with four AMs. This is because MDF was selecting higher deadlines than FIFO under those conditions and missing short jobs.

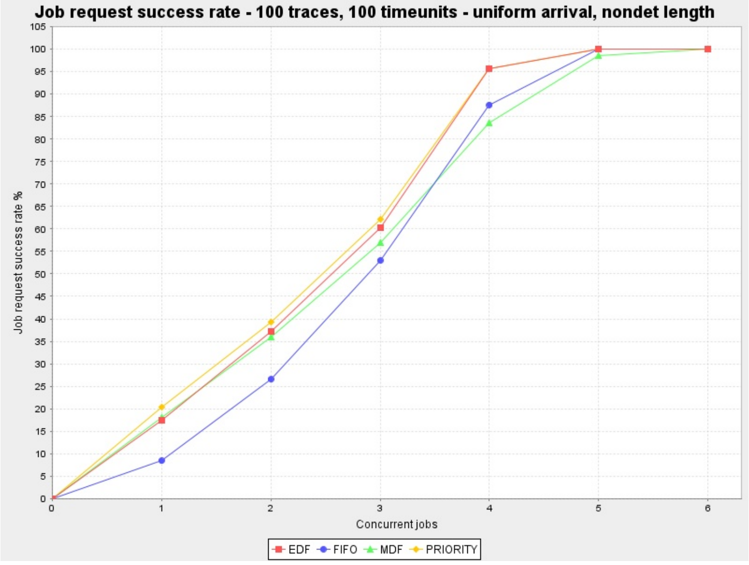


Figure 4.2: Job request success rate for a scenario in favor of EDF. This is the ratio of jobs received by the Resource Manager (RM) that completed successfully. EDF and Priority Queue were performing the best while MDF got worse as more AMs became available. MDF was selecting longer jobs because deadlines were relative to job lengths, and they were new jobs because the cluster was close to handling the entire workload. This increased the chances of old jobs in the queue missing their deadlines. FIFO chose old job requests with low remaining deadlines and therefore low chance of success.

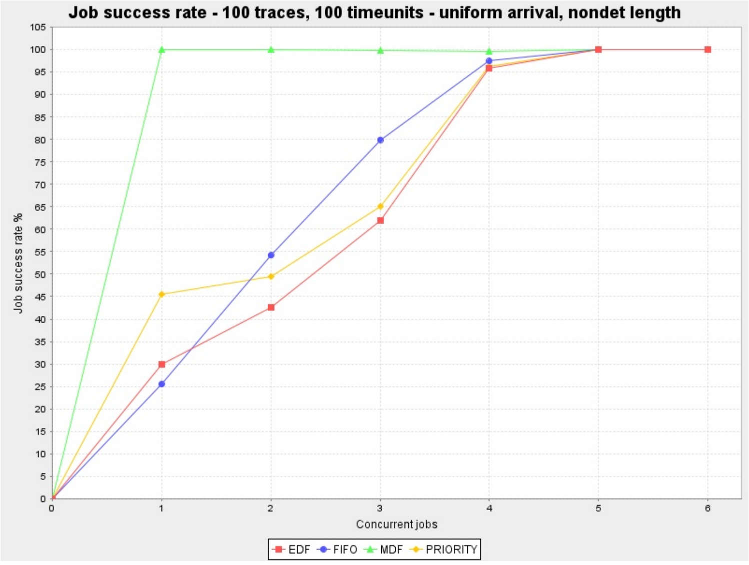


Figure 4.3: AM job success rate for a scenario in favor of EDF. This is the ratio of jobs that were sent from the RM to the AMs and completed successfully. MDF always selected the job with the highest remaining deadline and therefore the one most likely to succeed. Priority Queue scored higher than EDF because 10% of jobs were high priority jobs and were therefore started sooner when they had more of their deadline remaining. Because of the nondeterministic job length pattern, the oldest job in the queue was not necessarily with a low remaining deadline so FIFO had more success than EDF.

Table 4.5: Results for a scenario unfavorable to EDF

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
EDF	38.81	59.15%	62.57%
FIFO	38.84	59.12%	62.54%
MDF	22.71	76.09%	94.84%
PRIORITY	36.73	61.34%	66.54%

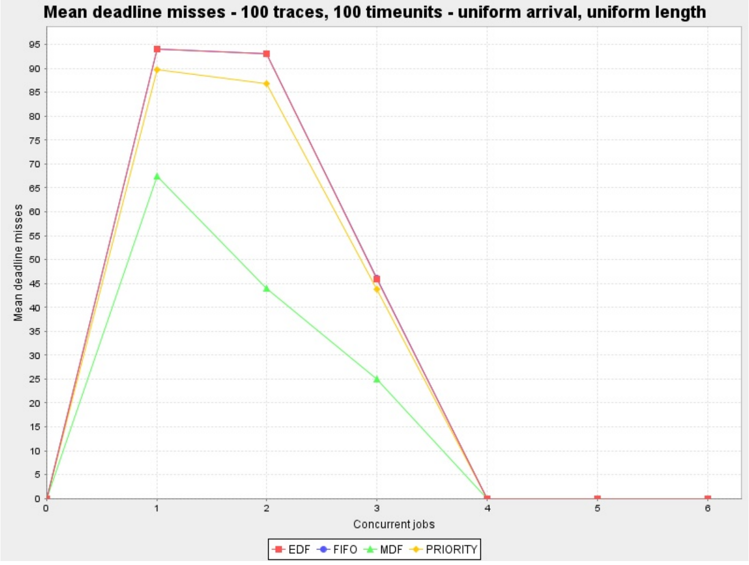


Figure 4.4: Mean number of deadline misses for a scenario unfavorable to EDF. The job length was uniform, deadlines were short and had usually run out by the time EDF and FIFO (overlapping in the figure) started them. This was because the oldest jobs were in the front of the queue. Priority Queue started high priority jobs sooner and was therefore passing more deadlines than EDF and FIFO. MDF started more new jobs and before their deadlines ran out.

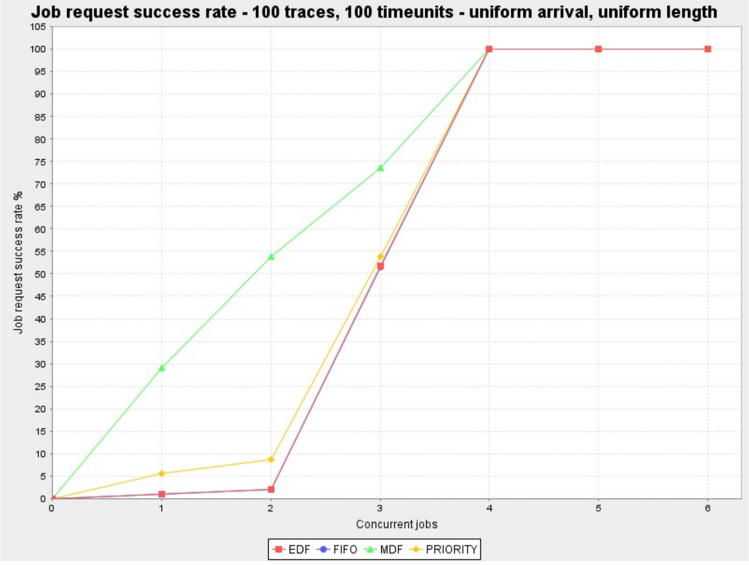


Figure 4.5: Job request success rate for a scenario unfavorable to EDF. Of all jobs sent to the cluster (generated by the RM), EDF and FIFO are completing the least number successfully. In this case, MDF is clearly preferable with the short deadlines and uniform job lengths, especially when the cluster has too few AMs to handle the load. The success of EDF, FIFO and Priority Queue grows fast from two AMs to four but linearly the entire time with MDF.

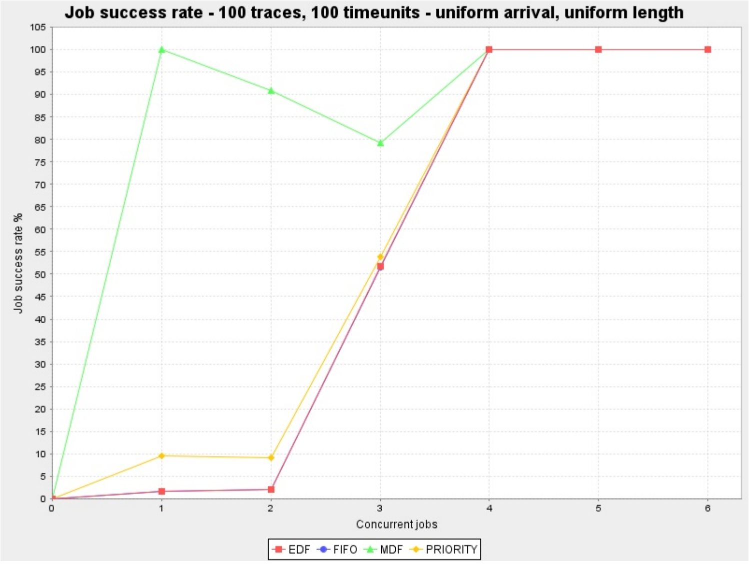


Figure 4.6: AM job success rate for a scenario unfavorable to EDF. MDF always selected the job most likely to succeed when there was only one AM, which in this case was the newest job. With two and three AMs, MDF started selecting older jobs that might have their deadlines already run out. With four AMs every job started was completed under the specified workload.

4.2.3 First-In-First-Out (FIFO)

The jobs were dispatched on a first-in-first-out basis. The Resource Manager kept the incoming job queue sorted so the oldest job was in the front of the queue. Jobs arriving were placed in the back of the queue and when a job was removed from the front, all the jobs were shifted to the front.

Scenario in favor of FIFO (Tables 4.8 and 4.9, Figures 4.10 to 4.12)

Same as the scenario unfavorable to MDF.

Scenario unfavorable to FIFO (Tables 4.6 and 4.7, Figures 4.7 to 4.9)

In one scenario unfavorable to FIFO the job from the front of the queue has the maximum deadline and it is less than the job length by a minimal amount. This will make FIFO miss the deadline and the most processing power possible will have been wasted. The Resource Manager did not check whether deadlines had already run out for jobs when dispatching them as he might assume the jobs would later be split into tasks like in Hadoop. In these experiments all jobs ran on one AppMaster each and were not split into tasks to run on other nodes.

Table 4.6: Parameters for a scenario unfavorable to FIFO

Parameters					
Epsilon	High priority job probability	Length pattern	Length increment	Length minimum	Length points
0.5	10%	ascending	1	1	6

Table 4.7: Results for a scenario unfavorable to FIFO

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
EDF	32.61	65.26%	70.52%
FIFO	40.76	56.60%	71.83%
MDF	30.91	67.27%	97.45%
PRIORITY	30.99	67.02%	75.82%

Job length was ascending from 1 to 6 in this scenario, 1, 2, 3, 4, 5, 6, 1..., incremented by 1 every time unit. When FIFO selected long jobs with short deadlines (like a job of length 6) many short jobs that arrived the next time units missed their deadlines. Additionally, the short deadlines ran out faster so there were more long jobs being left in the queue. MDF had more success than EDF because the deadlines were short so MDF was more likely to start jobs who were new and whose deadlines had not passed. Priority Queue (PQ) performed better than EDF because it started the high priority jobs as soon as possible and therefore possibly before their deadlines ran out. They also ran instead of low priority jobs whose deadlines had already run out. PQ was still worse than MDF but the difference in performance between PQ and MDF was less than between PQ and EDF. The difference in performance between FIFO and EDF the second worst policy, was the greatest.
Another scenario unfavorable to FIFO was the one unfavorable to EDF (Section 4.2.2).

4.2.4 Maximum Deadline First (MDF)

The job dispatcher selected the job with the maximum deadline first. The Resource Manager in ReGen did this by iterating through the incoming job queue and keeping an integer for the maximum deadline found. The incoming queue was not kept sorted nor shifted

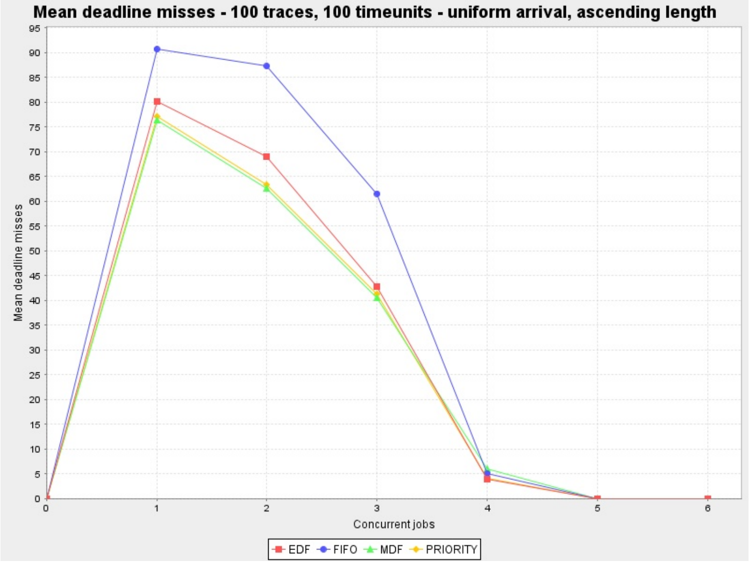


Figure 4.7: Mean number of deadline misses for a scenario unfavorable to FIFO. When FIFO selected jobs from the peak of an ascending job length pattern, the shorter jobs arriving immediately after could miss their deadlines. MDF was best because of the short deadlines it selected jobs whose deadlines had not run out. EDF performed better than FIFO because it was more likely to select short jobs because of their short deadlines and waste less processing time on long jobs and miss short jobs. Priority Queue performed better than EDF as usual because it was more likely to complete high priority jobs instead of low priority jobs whose deadlines might already have run out.

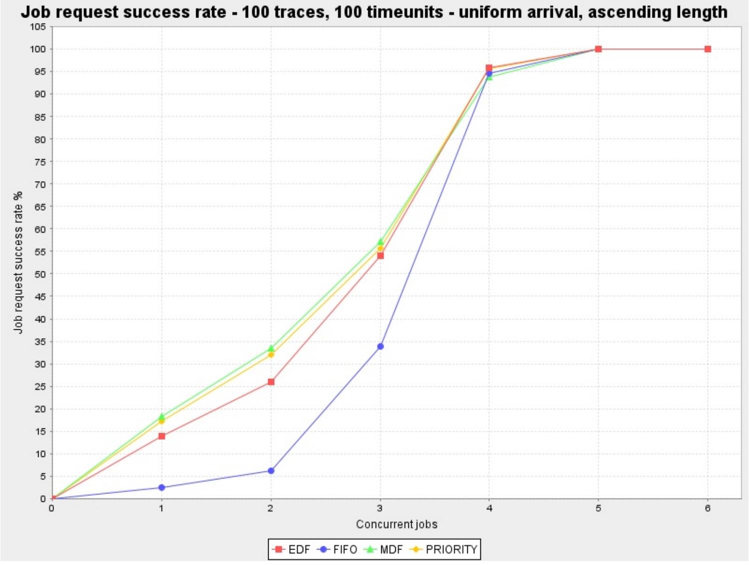


Figure 4.8: Job request success rate for a scenario unfavorable to FIFO. The number of jobs the RM received that completed successfully was low for FIFO with less than three AMs but grew exponentially up to four. The other policies success grew more linearly.

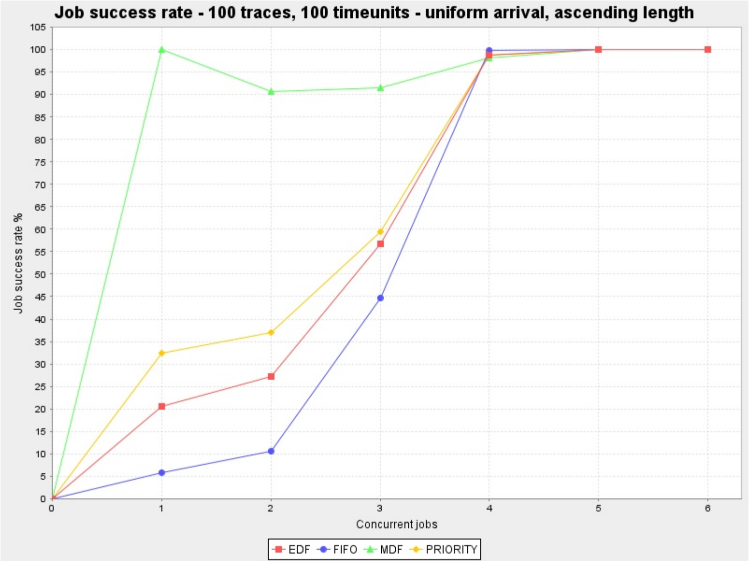


Figure 4.9: AM job success rate for a scenario unfavorable to FIFO. Again we see MDF have the most success with started jobs because it is less likely to start jobs whose deadlines are low. As the number of AMs grew the success rate decreased because MDF started selecting jobs with lower deadlines. The growth of the success rate of the other policies is exponential in all cases and the order of their performance is the same as in the other charts for this scenario.

to the front so gaps could appear between jobs in the queue. New jobs were inserted as close to the front of the queue as possible, and if multiple jobs had the same maximum deadline, the one nearest to the back of the queue was dispatched.

Scenario in favor of MDF (Tables 4.6 and 4.7, Figures 4.7 to 4.9)

Same as the scenario unfavorable to FIFO (Section 4.2.3). The maximum deadline was the same as its job length and not much greater than the other deadlines. Job length was uniform so MDF would always select the latest job.
The low epsilon worked in favor of MDF because the other policies did not select jobs as soon as they arrived and their deadlines were therefore more likely to run out.

Scenario unfavorable to MDF (Tables 4.8 and 4.9, Figures 4.10 to 4.12)

One scenario unfavorable to MDF is where the maximum deadline is less than the jobs length and jobs are being run uberized on a single AM. Then the jobs deadline will run out before the job completes as the AMs can only process 1 unit of time of the job in ReGen. The parameters chosen for this scenario would make job length grow as follows: 0, 2, 4, 8, 16... as job length was computed as $current\ time * exponential\ multiplier$. With

Table 4.8: Parameters for a scenario unfavorable to MDF

Parameters			
Epsilon	High priority job probability	Job length pattern	Exponential multiplier
1.5	30%	exponential	2

Table 4.9: Results for a scenario unfavorable to MDF

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
EDF	4.54	75.09%	79.45%
FIFO	4.54	75.09%	79.45%
MDF	9.50	50.91%	100.00%
PRIORITY	6.77	63.61%	84.63%

a high epsilon, MDF selected a job that had a high remaining deadline and because job lengths grew exponentially, it was always selecting the latest job. MDF therefore missed deadlines for smaller jobs that arrived earlier. FIFO and EDF were performing identically because they were both selecting the oldest job from the queue. Priority Queue (PQ) did not always select the oldest job which increased its success rate for one AM because then the oldest jobs deadline was the most likely to have run out. With more AMs PQ would select the highest priority job which could have any deadline and any length and therefore cause short jobs that could have been completed to be missed. It would also select the oldest job which with more AMs would become more and more likely to be completed successfully and therefore should have focused on them with this type of workload. The performance of EDF and FIFO increased faster with more AMs when compared to PQ and MDF. The performance of all policies increased linearly.

Because job lengths grew exponentially, fewer jobs were processed in the same amount of time units as in the other scenarios. This made the relative difference in performance between PQ and EDF greater than in previous scenarios.

Few scenarios were found unfavorable to MDF.

4.2.5 Priority Queue

A certain probability of jobs being of high priority could be set and their lengths could be set as uniform or to have the same job length patterns as low priority jobs. If there were

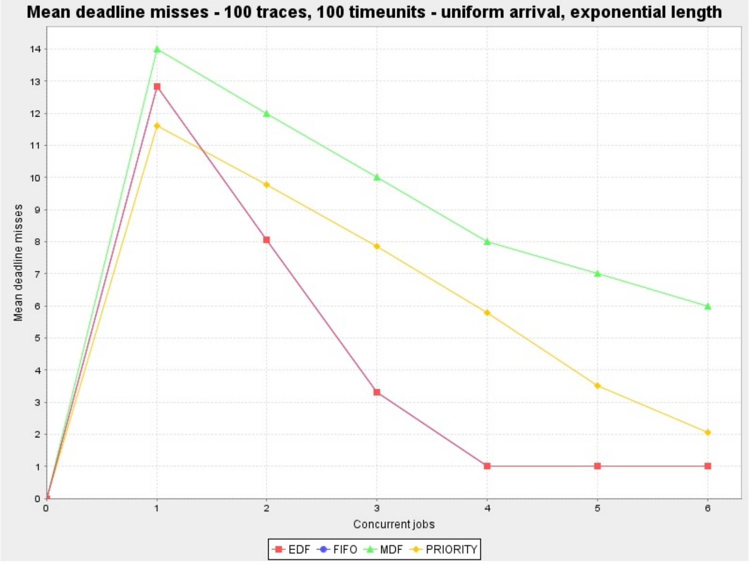


Figure 4.10: Mean number of deadline misses for a scenario unfavorable to MDF. MDF had the highest number of deadline misses because the job length grew exponentially and was therefore selecting the newest job. It took the most time to complete so the shorter jobs that came before were missed. EDF and FIFO were performing identically because they were both selecting the oldest job from the queue. Priority Queue did not always select the oldest job which also was the most likely to have had its deadline run out when only one AM is available and is therefore performing better than EDF and FIFO. As the number of AMs grew, the deadline of the oldest job became less and less likely to run out so when a new high priority job entered the queue, it would be longer and run at the expense of shorter jobs.

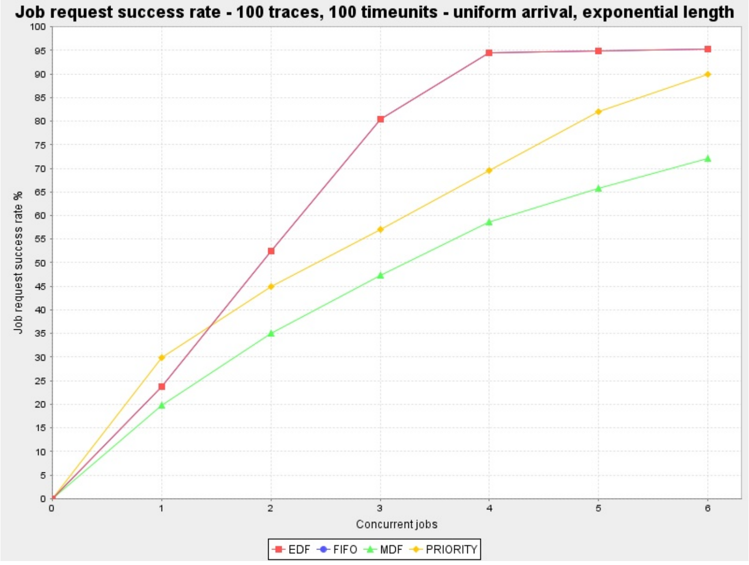


Figure 4.11: Job request success rate for a scenario unfavorable to MDF. Nearly identical pattern to the mean number of deadline misses in Figure 4.10. The result for MDF however curved towards the X-axis suggesting relatively worse performance as the cluster grows with this type of workload.

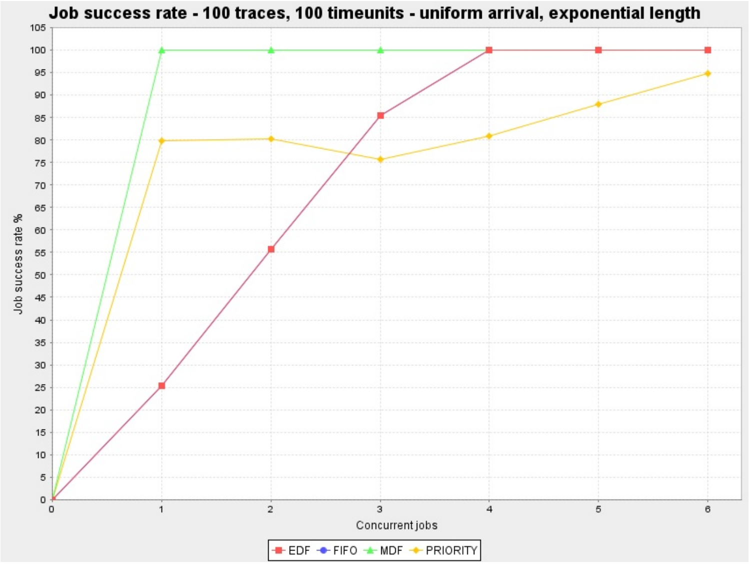


Figure 4.12: AM job success rate for a scenario unfavorable to MDF. Every job sent to the AMs was completed successfully using MDF. Job completion improved for EDF and FIFO with more AMs and with four AMs all jobs sent were being completed within the deadline. Priority Queue (PQ) was sending high priority jobs which were usually new jobs with high remaining deadlines and were therefore completing successfully. The ratio of high priority jobs was unchanged even though the number of AMs increased which worked in favor of PQ until there were three AMs. Then PQ started sending high priority jobs that were possibly long jobs and missing short jobs with short deadlines that could have completed successfully. The success of PQ then grew linearly as the number of AMs increased.

Table 4.10: Parameters for a scenario in favor of Priority Queue

Parameters					
Epsilon	High priority job probability	High priority job length	Low priority job length	Job length minimum	Job length maximum
0.5	30%	1	nondeterministic	1	6

Table 4.11: Results for a scenario in favor of Priority Queue

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
EDF	37.63	59.56%	69.46%
FIFO	42.86	53.95%	72.28%
MDF	34.12	63.48%	96.74%
PRIORITY	28.21	70.10%	84.05%

multiple jobs of the same priority, EDF was used as a tiebreaker. High and low priority jobs both used the same incoming job queue instead of separate queues for each. The incoming job queue was not kept sorted nor was it shifted to the front so there could be gaps between jobs in the queue. New jobs were inserted as close to the front of the queue as possible.

Scenario in favor of a Priority Queue (Tables 4.10 and 4.11, Figures 4.13 to 4.15)

The high priority jobs were short, or 1 unit of time. Priority Queue policy outperformed the other policies because on average the jobs dispatched had shorter job lengths. This scenarios low priority jobs length was nondeterministic which was not favorable to FIFO because it selected the oldest job. The scenario had short deadlines which was unfavorable to EDF and favorable to MDF. The following results display those features, as well as the previous scenarios in this chapter. The probability of jobs being of high priority was high compared to the previous scenarios, or 30%.

Scenario unfavorable to Priority Queue (Tables 4.12 and 4.13, Figures 4.16 to 4.18)

The high priority jobs were long, or 9 units of time. As expected, the Priority Queue (PQ) policy performed worse with longer high priority jobs. The parameters used besides the

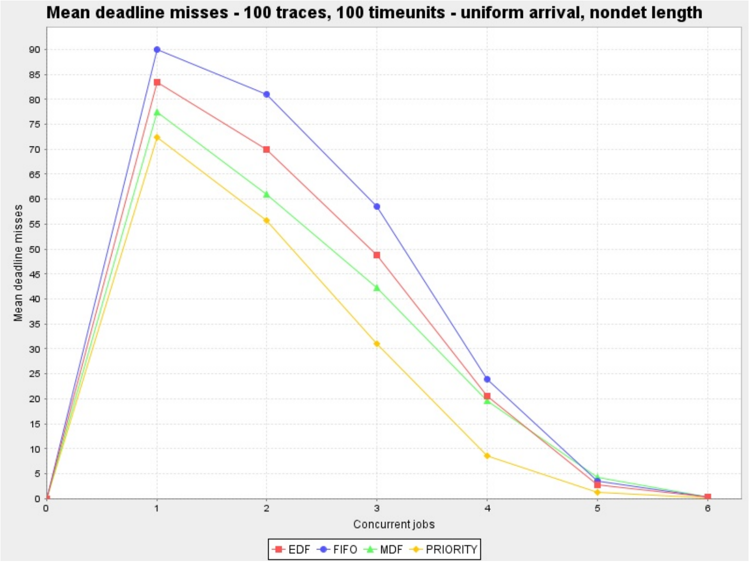


Figure 4.13: Mean number of deadline misses for a scenario in favor of Priority Queue (PQ). Here PQ is performing the best because the high priority jobs were shorter than the low priority jobs. The low priority jobs lengths were nondeterministic and FIFO always selected the oldest jobs whose deadlines remaining were the lowest so it had the highest number of deadline misses. The deadlines were short which favored MDF over EDF. MDF's number of deadline misses decreased more linearly than the other policies as the number of AMs grew.

Table 4.12: Parameters for a scenario unfavorable to Priority Queue

Parameters					
Epsilon	High priority job probability	High priority job length	Job length pattern	Job length minimum	Job length maximum
0.5	30%	9	nondeterministic	1	6

high priority job length were the same as in the scenario in favor of PQ (Section 4.2.5, Table 4.10) so the results for FIFO, EDF and MDF went nearly unchanged.

4.2.6 Summary of the Dispatch Policy Results

Table 4.14 shows the averages of the results for all the dispatch policies. MDF had the least amount of deadline misses, second highest job request success rate and the highest AM job success rate. EDF had the second least deadline misses, the highest job request success rate and the lowest AM job success rate. Priority Queue (PQ) had the third highest

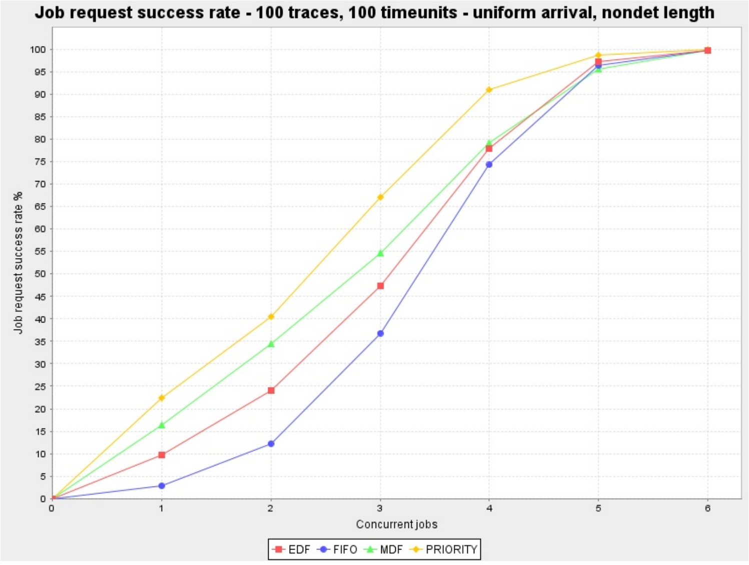


Figure 4.14: Job request success rate for a scenario in favor of Priority Queue (PQ). This pattern is a reverse of the mean number of deadline misses in Figure 4.13. PQ was dispatching the high priority jobs that happen to be shorter than low priority jobs and could therefore service more jobs over time as the AMs freed up faster. FIFO selected the oldest job which also was the most likely to have a deadline miss. The lengths of jobs were nondeterministic so EDF did not necessarily select the oldest job but the deadlines were short so it was likely to select a job whose deadline would run out. The short deadlines worked in favor of MDF because it would avoid selecting jobs whose deadlines had already run out.

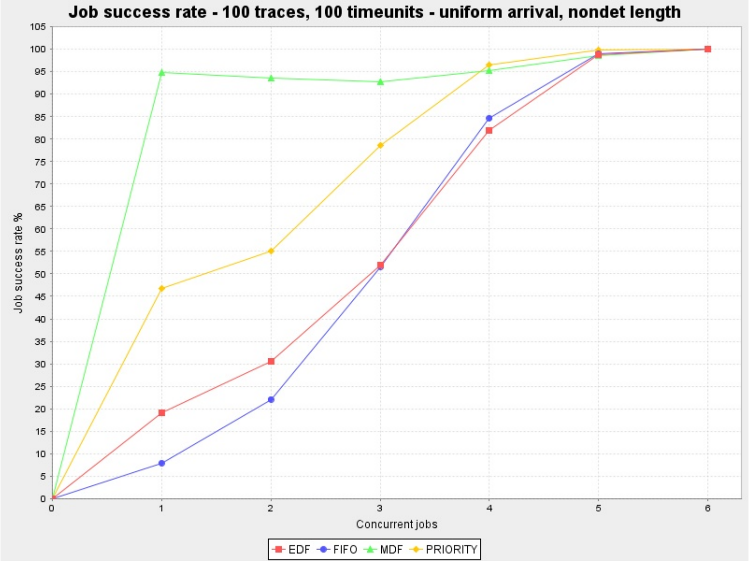


Figure 4.15: AM job success rate for a scenario in favor of Priority Queue (PQ). MDF had the most success completing jobs sent to the AMs but not all of them completed because the job lengths were nondeterministic in this scenario. That meant the maximum deadline could be for a long job that would run out while shorter jobs were ignored. PQ was starting high priority jobs fast and completing them quickly because they were short. The short deadlines were not favorable to EDF and FIFO was dispatching old jobs.

Table 4.13: Results for a scenario unfavorable to Priority Queue

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
EDF	37.51	59.72%	69.64%
FIFO	43.06	53.74%	72.09%
MDF	33.91	63.75%	96.92%
PRIORITY	52.05	42.92%	68.22%

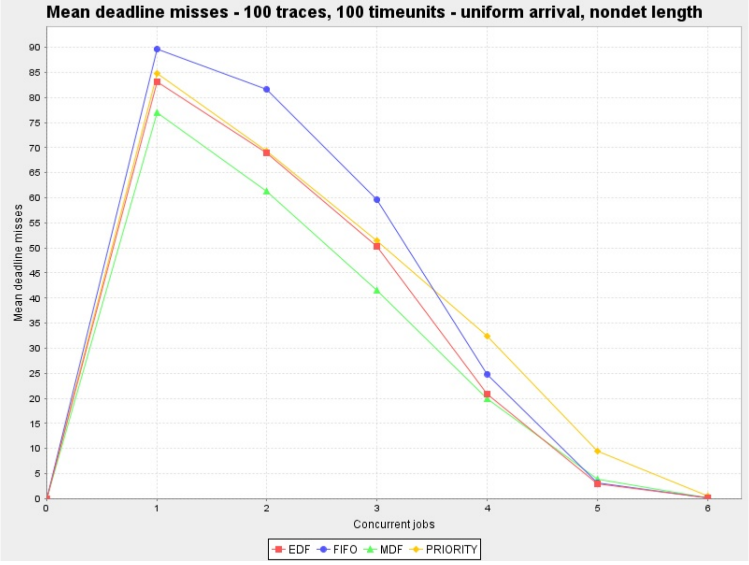


Figure 4.16: Mean number of deadline misses for a scenario unfavorable to Priority Queue (PQ). Nearly identical to Figure 4.13 because the same parameters were used outside of the job length for high priority jobs. PQ is clearly performing worse than before because of the higher job length of high priority jobs. FIFO still performed worse with fewer AMs because it dispatched jobs that were old and therefore with low remaining deadlines. In the meantime PQ dispatched high priority jobs while they had high remaining deadlines and therefore more likely to complete within the timelimit. This then changed with four AMs when FIFO started dispatching jobs before they got too old in the incoming job queue. High priority jobs were long in this scenario so short jobs were being missed by PQ.

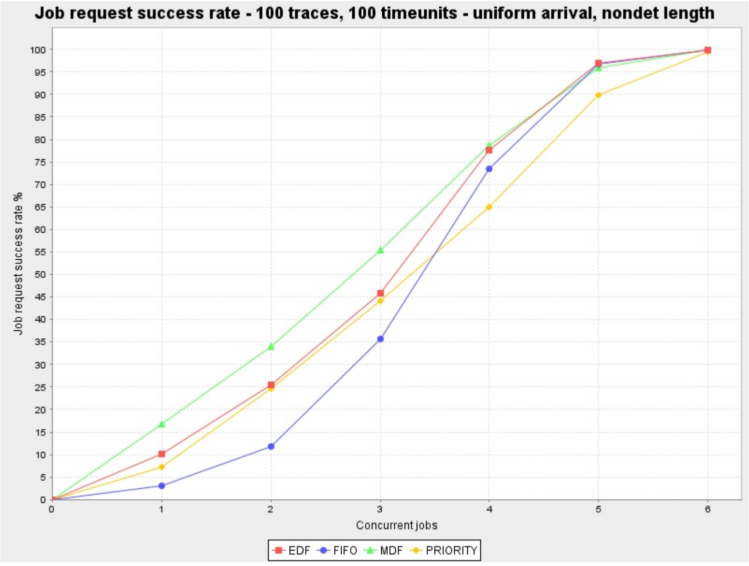


Figure 4.17: Job request success rate for a scenario unfavorable to Priority Queue (PQ). Same pattern for EDF, FIFO and MDF as in Figure 4.14. PQ is completing fewer jobs though because the length of high priority jobs was longer in this scenario. This caused the AMs to process fewer jobs over time than the other policies and short jobs being missed in favor of the longer high priority jobs.

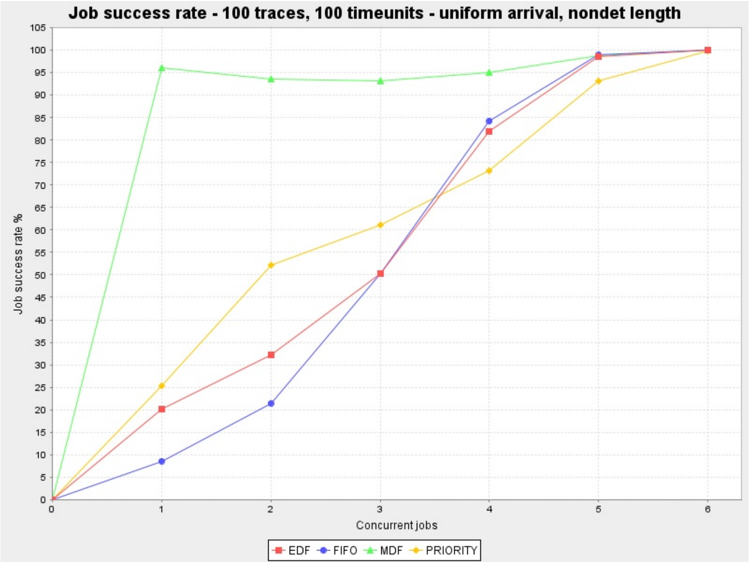


Figure 4.18: AM job success rate for a scenario unfavorable to Priority Queue (PQ). Same pattern for EDF, FIFO and MDF as in Figure 4.15. In the case of PQ however the high priority jobs are now longer so more time is spent on long jobs while the deadlines remaining for low priority jobs decrease. More jobs are therefore being sent to the AMs with low remaining deadlines than before and PQ's performance worse in completing started jobs.

Table 4.14: Summary of results for the dispatch policies

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
EDF	29.51	64.86%	71.47%
FIFO	33.54	60.53%	74.16%
MDF	26.95	64.67%	97.64%
PRIORITY	30.07	62.68%	76.77%

mean deadline misses, the third lowest job request success rate and the second highest AM job success rate. FIFO had the most deadline misses, the least job request success rate and the second lowest AM job success rate. Important to note is that these results were not generated by an exhaustive search of the statespace but from scenarios in favor of and unfavorable to each policy. Overall MDF came out best and FIFO the worst. PQ improved its tiebreaker EDF AM job success rate but lowered the job request success rate. The difference in deadline misses between PQ and EDF was small.

4.3 Natjam-R Job Eviction Policy Results

A job eviction policy determines which running job to evict and make checkpoint of when preemption takes place. The results are represented in the same way as for the dispatch policies, except for an additional mean number of checkpoints chart. They correlate with results from (Cho et al., 2013) in that Maximum Deadline First (MDF) is preferable to Maximum Laxity First (MLF) where laxity is the difference between the deadline remaining and time remaining of a job. In the paper, traces from three MapReduce jobs running on a Yahoo cluster were simulated in Natjam-R and those results showed the MLF jobs running in lockstep which delayed their completion due to a checkpoint overhead.

For MDF, EDF was used to select a job in the checkpoint and incoming job queues to preempt for. The Resource Manager would every time unit scan a queue, find the earliest deadline job, scan the AppMasters and preempt the maximum deadline job. For MLF, Least Laxity First (LLF) was used for dispatching instead of EDF.

4.3.1 Common parameters between the Natjam policies (Table 4.15)

Parameters for the Natjam policies were selected in a similar manner as for the dispatch policies (Section 4.2.1). The *epsilon* which determines deadlines of jobs was the same in both scenarios. No high priority jobs were used because they were not needed for the two particular scenarios but the ReGen software still allows for their usage for the Natjam policies.

Table 4.15: Common parameters between the Natjam-R policies

Max AMs	Queue size	Simulations	Timeunits	Epsilon	High priority job probability
6	12	100	100	1.0	0%

4.3.2 Maximum Deadline First (MDF)

Scenario in favor of Natjam-R MDF (Tables 4.16 and 4.17, Figures 4.19 to 4.21)

In MDF the job with the maximum deadline remaining is evicted first and how much is left of the job is not taken into consideration. In this scenario there was a context switching overhead of 1 time unit that caused MLF to perform poorly because it did more checkpoints than MDF as shown in Figure 4.22. For simplicity the entire context switching overhead was added to the length of the job being preempted at the time of checkpointing and not to the job being started. Restarting a job from a checkpoint also carries an overhead in Hadoop and was included in the length added to the job while checkpointing in ReGen. Therefore, checkpointed jobs were assumed to cost more processing time by the cluster than their original job lengths.

The problem with MLF was that it would start lower laxity jobs and checkpoint higher laxity ones and there could be many checkpoints made before any job was completed. Laxity decreased faster in the queues because the job time remaining went unchanged while the deadline remaining was decreasing over time. While a job was being processed by an AppMaster (AM), both the deadline remaining and time remaining were decreasing simultaneously so the laxity stayed the same. MLF was therefore swapping jobs and adding overhead. Conversely, MDF would preempt a high deadline job for a lower deadline job and not preempt them for each other again because the deadlines decreased at the same rate.

Table 4.16: Parameters for a scenario in favor of Natjam-R MDF

Parameters				
Job arrival pattern	Jobs per timeunit	Job length pattern	Job length	Context switching overhead
uniform	1	ascending	5, 6, 7, 5...	1

Table 4.17: Results for a scenario in favor of Natjam-R MDF

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
MDF	59.19	31.44%	37.01%
MLF	76.39	11.88%	12.77%

4.3.3 Maximum Laxity First (MLF)

Scenario close to being in favor of Natjam-R MLF (Tables 4.18 and 4.19, Figures 4.23 to 4.25)

One scenario close to being in favor of MLF had uniform job arrival and descending job length. In this scenario there was no context switching overhead so no penalty was paid for checkpoints. The job length pattern was descending, *epsilon* was 1.0 which meant deadlines were twice the job length so normally the deadlines arriving were lower than the ones that came earlier. MDF was therefore likely to preempt because it used EDF for dispatching jobs. The laxity of jobs waiting in the incoming job queue and the checkpoint queue decreased faster than the laxity of jobs running at the AMs. This made MLF prone to preempt while using Least Laxity First (LLF) dispatching. Additionally, because of the job length pattern and LLF, the laxity of a job arriving was usually less than that of the previous job when it arrived. This increased the rate of checkpoints made by MLF even further.
No scenarios were found where MLF outperformed MDF.

Table 4.18: Parameters for a scenario close to being in favor of Natjam-R MLF

Parameters				
Job arrival pattern	Jobs per timeunit	Job length pattern	Job length	Context switching overhead
uniform	1	descending	6,5,4,3,2,1,6...	0

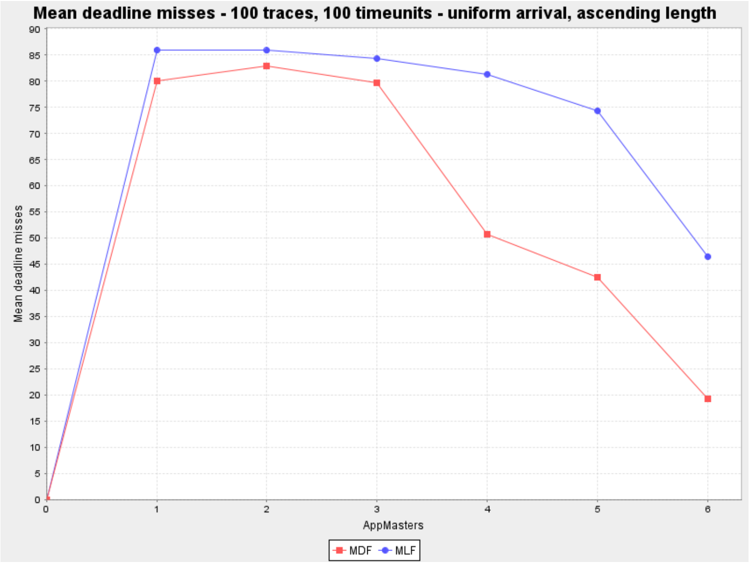


Figure 4.19: Mean number of deadline misses for a scenario in favor of Natjam-R MDF. The performance of the policies was similar for clusters with up to three AMs while the workload was too much. At around four AMs the mean number of deadline misses dropped faster for MDF than for MLF. MDF was allowing the jobs with the lowest deadlines to finish instead of preempting them with jobs of lower laxity and process them in lockstep as MLF did.

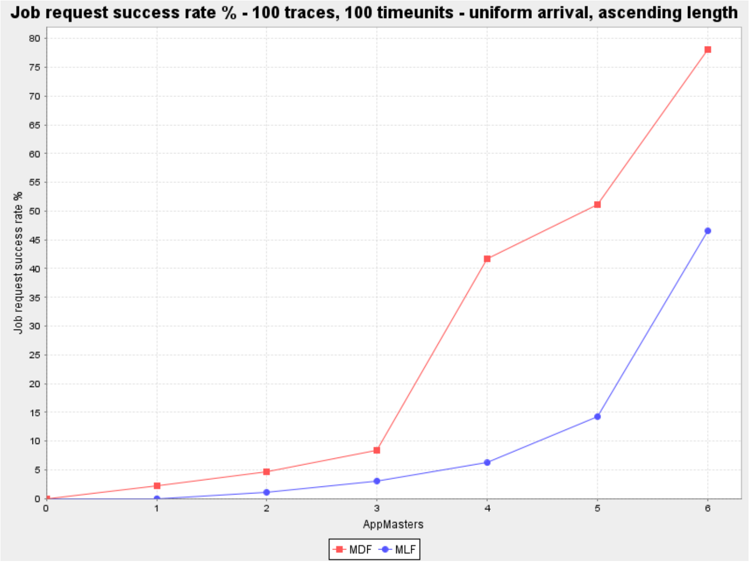


Figure 4.20: Job request success rate for a scenario in favor of Natjam-R MDF. Nearly an inverse of the mean number of deadline misses in Figure 4.22. MDF saw a jump in the ratio of job requests completed with four AMs under the selected workload while MLF's increased more gradually. MLF processed jobs in lockstep while MDF allowed the lowest deadline jobs to finish. More AMs allowed MLF to simply process more jobs in lockstep instead of greatly improving the success of job requests.

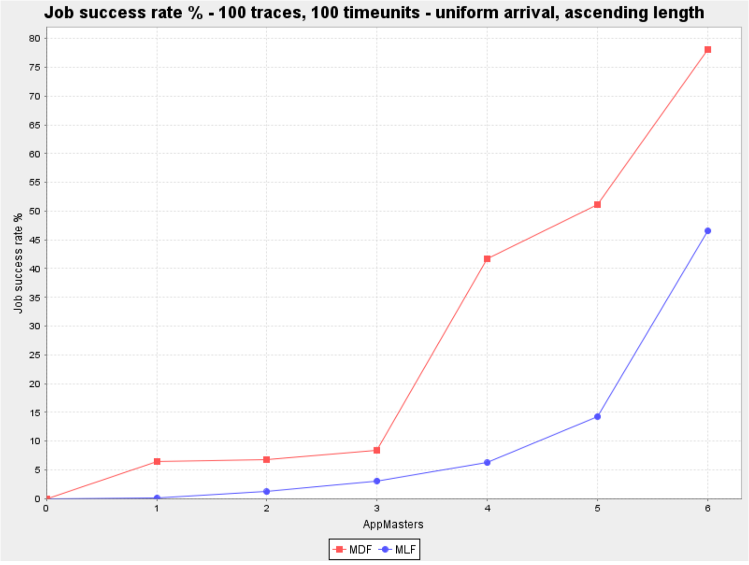


Figure 4.21: AM job success rate for a scenario in favor of Natjam-R MDF. Similar pattern as the two previous figures, Figure 4.19 and Figure 4.20. The jobs sent to the AMs had low deadlines because EDF dispatch was used for MDF eviction. This caused MDF to complete few jobs sent to the AMs before these early deadlines ran out until more AMs became available. Least Laxity First (LLF) dispatch was used for MLF eviction and constantly switching between jobs in the queues and those running at the AMs wasted processing time and created an overhead for MLF.

Table 4.19: Results for a scenario close to being in favor of Natjam-R Maximum Laxity First (MLF)

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
MDF	34.72	62.63%	67.85%
MLF	37.54	59.49%	65.71%

4.3.4 Summary of the Natjam-R Eviction Policy Results

For the Natjam-R scenarios, MDF had fewer deadline misses, higher job request success rate and higher AM job success rate (Table 4.20). In all cases the difference between the two policies was large.

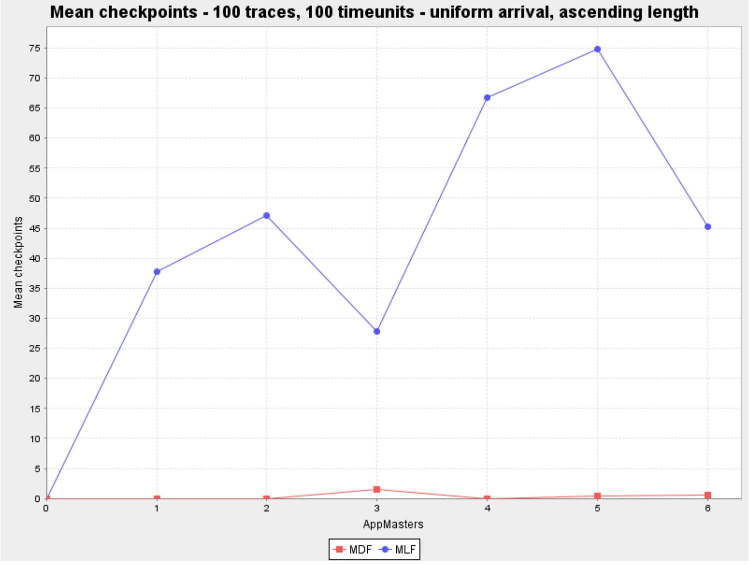


Figure 4.22: Mean number of checkpoints for a scenario in favor of Natjam-R MDF. In this scenario the parameters chosen caused higher deadlines to enter the incoming job queue. As MDF used EDF as a dispatching policy few preemptions took place because the jobs in the incoming job queue had higher deadlines than the jobs running at the AMs. MLF on the other hand made more preemptions because it used Least Laxity First (LLF) for dispatching. As laxity was computed as the difference between deadline remaining and job completion time remaining it decreased faster in the incoming job queue and checkpoint queue than for jobs running at the AMs. An integer for deadline remaining was decreased by 1 every time unit for every job, and an integer for job completion time remaining was decreased by 1 every time unit for every job being processed at an AM. There were three points on the job length ascension (lengths 5, 6, and 7) so with three AMs the lowest laxity jobs could complete before jobs in the incoming and checkpoint queues would reach low laxity enough to preempt the ones being processed. This caused fewer checkpoints for three AMs. With two AMs the cluster was not handling the load well enough so more preemptions were required between the AMs, the incoming and checkpoint queues. With four AMs there were more running jobs to choose from to preempt.

Table 4.20: Summary of results for the Natjam-R eviction policies

Results			
Policy	Mean deadline misses	Job request success rate	AM Job success rate
MDF	46.96	47.04%	52.43%
MLF	56.97	35.69%	39.24%

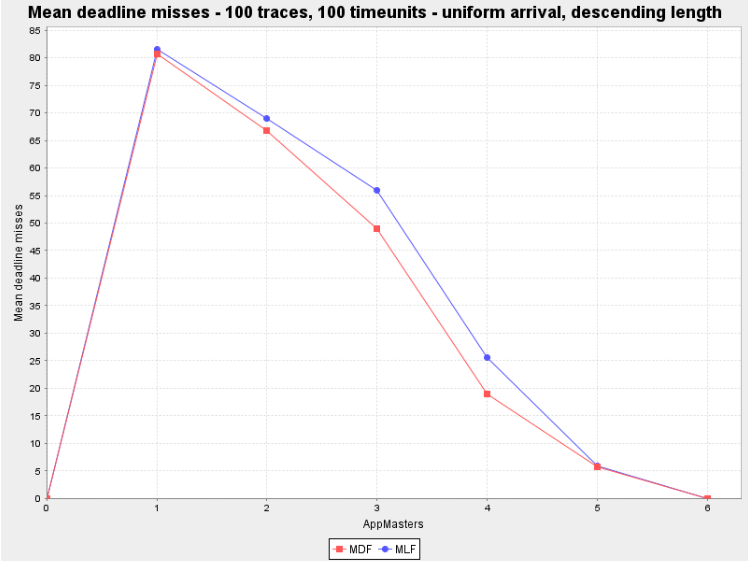


Figure 4.23: Mean number of deadline misses for a scenario close to being in favor of Natjam-R MLF. The difference between the policies was small for any size of the cluster. MDF still had fewer deadline misses even though both policies were preempting to make room for most new jobs. The descending job length pattern caused jobs with lower deadlines to enter the incoming job queue so the EDF dispatch policy used with MDF eviction created a scenario unfavorable to MDF compared to other scenarios. The same happened with MLF which used LLF for dispatching and as the jobs entering the incoming job queue had lower laxities, MLF preempted for them. MLF also processed started jobs in lockstep which caused even further deadline misses.

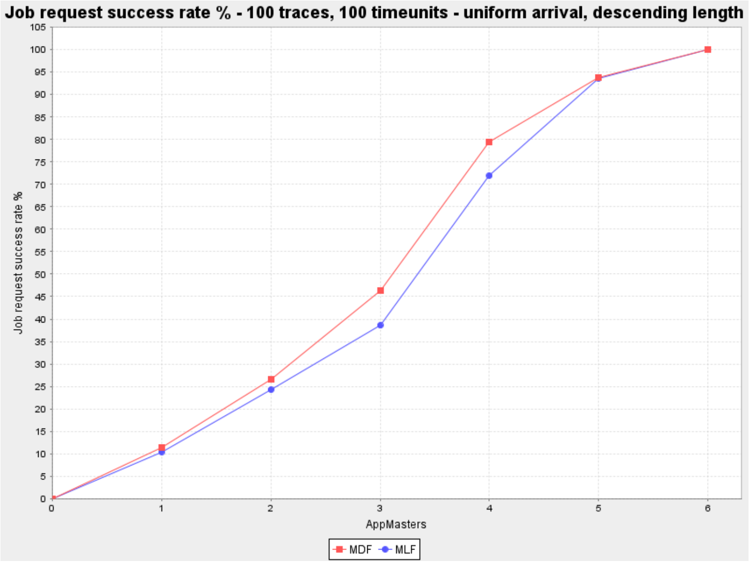


Figure 4.24: Job request success rate for a scenario close to being in favor of Natjam-R MLF. An inverse of the mean number of deadline misses in Figure 4.23. The number of jobs sent to the RM that completed successfully increased with more AMs and MDF performed better than MLF. The job length pattern caused the deadlines of job requests to arrive lower with time so MDF using the EDF dispatch policy tended to preempt for newly arrived jobs. MLF using LLF did the same as well as running jobs in lockstep causing even less success with job requests than this unfavorable scenario for MDF.

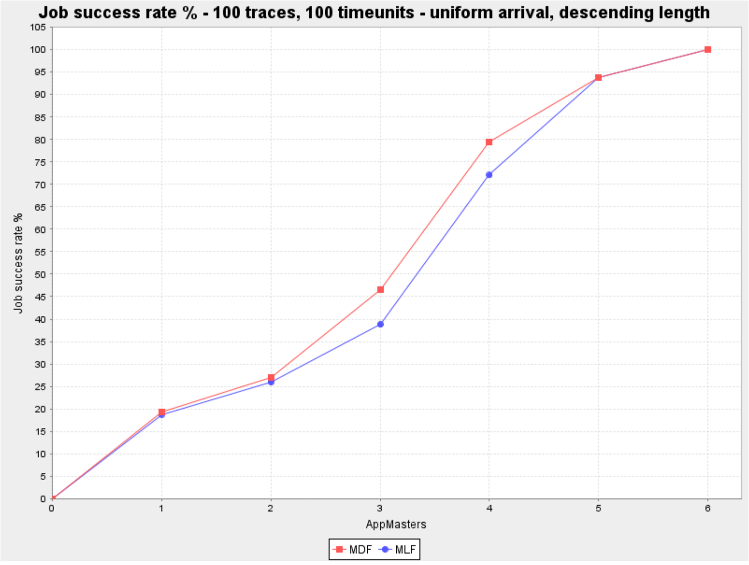


Figure 4.25: AM job success rate for a scenario close to being in favor of Natjam-R MLF. Similar pattern as Figure 4.25. More jobs sent to the AMs were completed within their deadlines using MDF even though both policies suffered from the same problem of having to preempt when new jobs arrived because of the descending job length pattern. Both deadlines and laxities became lower for arriving jobs and EDF was used for job dispatching for MDF, and LLF was used for MLF. This made the scenario unfavorable to both policies but MLF processed jobs from the AMs, the incoming job queue and the checkpoint queue in lockstep which made the difference.

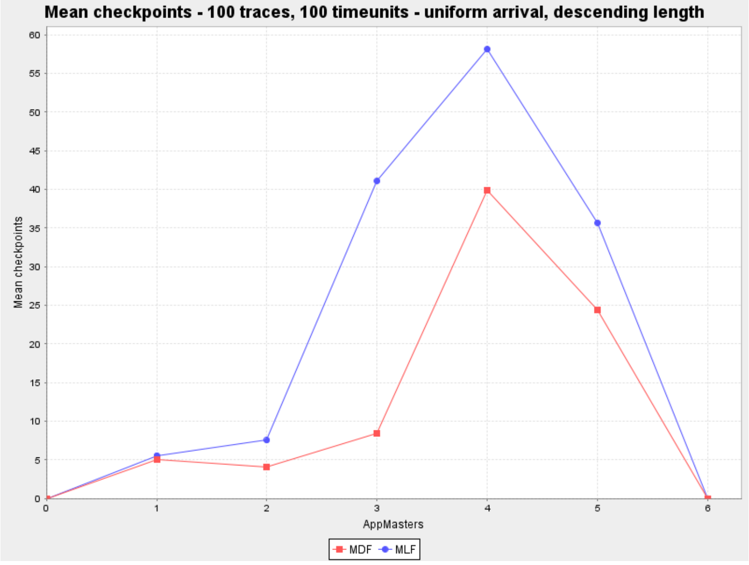


Figure 4.26: Mean number of checkpoints for a scenario close to being in favor of Natjam-R MLF. Clear differences can be seen from the checkpoint chart in Figure 4.22. MDF had to make more checkpoints using EDF for dispatching jobs while the job length pattern was descending. Arriving jobs deadlines and laxities were lower across time units which was unfavorable to both policies but MLF which used LLF for dispatching proceeded to run jobs in lockstep. The peak with four AMs was because the workload was still too high for a cluster of that size but more AMs were available to preempt jobs from. The workload is relatively less with more than four AMs so there are fewer new jobs to select from to preempt jobs running at the AMs. With six AMs there were no preemptions required as there was always a free AM available to run any new job.

Chapter 5

Conclusions

5.1 Conclusion and Future Work

5.1.1 Conclusion

In this thesis we presented the ReGen software that uses the actor model to analyze different scheduling and eviction policies in a Hadoop MapReduce system. Code templates for Timed Rebeca models were introduced, and we produced evidence that suggests correctness of our implementation.

We examined four different scheduling policies and compared them under different workloads. We showed that they can outperform each other under certain circumstances, and the results from those experiments indicate MDF performs better overall, especially with short deadlines. EDF can on the other hand be used for long deadlines. Priority Queue's performance is mostly affected by its priority tiebreaker policy and other factors such as the length of high priority jobs. FIFO is not recommended for deadline scheduling. That is because of the obvious reason that FIFO dispatches the oldest job and the results show it has the most deadline misses. We therefore conclude that from the examined policies the choice is mainly between MDF and EDF depending on the length of deadlines and other factors. In general, if deadlines are short, use MDF, if deadlines are long, use EDF but factors such as the size of the cluster and the workload affect the risk of the cluster overloading. With a high risk of cluster overloading, MDF provides fewer deadline misses because it chooses the jobs most likely to succeed. With a low risk, EDF serves more job requests because it chooses more short jobs.

Additionally, we compared two job eviction policies and as there were no scenarios found in favor of MLF we conclude that MDF is preferable for preemption, at least in Natjam-R.

Finally, we conclude it is possible to model large and complex systems using Timed Rebeca with a code generator. This could open up new areas of research where using model checkers might be more beneficial than using real systems or simulators.

5.1.2 Future work

Explore adding a third actor to the models that runs tasks for the AMs. This could be used to compute things such as job completion times, utilization, cost and more. Additionally, different topology could be set up where tasks would be scheduled to run within the same rack of a cluster to avoid moving data between racks which would be expensive. Conversely, anti-colocation could be implemented to study load balancing and utilization, and preemption could be used to move tasks and jobs around. One way to implement this could be arrays within each rebec that list the distances to every other actor and therefore the network delay which would be modeled using the *after* command when messages are sent.

The actors could also be made heterogeneous in their efficiency and power consumption amongst other things. A multiplier for each rebec could be used for example to lengthen or shorten jobs or tasks sent to it. A value could be implemented for each rebec that is added to a sum of total power consumption every time unit the rebec processes jobs. The utilization (CPU, RAM, power, cost, etc.) could then be fitted into the charts or text files. This type of implementation could be accomplished by generating many models each with a different permutation of numbers of job and task running actors, and the results from the models aggregated. Dynamic creation of rebecs might make the process simpler.

For curved job arrival and length patterns, like splines and sine waves, higher workloads would be required in ReGen. This would mean more jobs arriving, more AppMasters running concurrently, and therefore larger models which would require more time, memory and hard drive space. The implementation for the curved patterns would also need to be added.

The code in ReGen that generates the job arrival and job length patterns could possibly be re-used in other projects. We would also like to add more policies to the software, MLF and LLF dispatching for example.

5.2 Discussion

As the variables we manipulated are more abstract than real world values, the question arises how valid are the results? We argue we can draw conclusions from them because they are generated under the same experimental conditions however abstract. There are no known defects and our models are assumed to capture the real world systems functionality as much as is necessary for our conclusions. We also like to point out that the conclusions drawn are not from an exhaustive search of the statespace but from scenarios in favor of and unfavorable to each policy. The results can however be reproduced and further exploration of the state space is possible so the ReGen software itself is also an interesting result that can be developed further. If it were developed further it could provide insights into more features of computer clusters as listed in the introduction and conclusion of this thesis.

The performance of MLF was found to be lower than expected. Changes or optimizations to it might be required for it to be preferable to other job eviction policies.

Clearly, real-world-scenarios provide important results but they do not allow further exploration of the state space. An obvious benefit of exploring the state space in our case is detection of, and verification of removal of, race conditions that can negatively affect the system.

As the models are more abstract than the real world systems, we would like to claim our results are not specific to the YARN and Natjam-R example, but that they can apply to other systems that dispatch or evict jobs using the above implemented policies.

Currently the Rebeca model checker writes every time unit of every simulation onto hard drive which is good for debugging but slows the experiments down. It might speed them up if only the last time unit of each simulation were written down, instead of all of them.

Lastly, dynamic generation of rebecs in Timed Rebeca could be an alternative to generating multiple models for each number of Application Masters in our examples. If rebecs were dynamically generated they could possibly also be killed so rebecs running tasks could be preempted instead of having a process messageserver run every time unit in the case of the Natjam-R models.

Bibliography

Aceto, L., Reynisson, A. H., Sirjani, M., Cimini, M., Jafari, A., Ingolfsdottir, A., & Sigurdarson, S. H. (2014). Modelling and simulation of asynchronous real-time systems using timed rebeca. *Science of Computer Programming*, *89, Part A*(0), 41 - 68. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0167642314000239` (Special issue on the 10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2011)) doi: http://dx.doi.org/10.1016/j.scico.2014.01.008

Allen, T. T. (2011). *Introduction to discrete event simulation and agent-based modeling*. Springer.

Bell, W. H., Cameron, D. G., Capozza, L., Millar, A. P., Stockinger, K., & Zini, F. (2002). Simulation of dynamic grid replication strategies in optorsim. In *Journal of high performance computing applications* (pp. 46–57). Springer-Verlag.

Bux, M., & Leser, U. (2013). Dynamiccloudsim: Simulating heterogeneity in computational clouds. In *Proceedings of the 2nd acm sigmod workshop on scalable workflow execution engines and technologies* (pp. 1:1–1:12). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2499896.2499897` doi: 10.1145/2499896.2499897

Buyya, R., & Murshed, M. (2002). Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE (CCPE*, *14*(13), 1175–1220.

Calheiros, R. N., Ranjan, R., Beloglazov, A., Rose, C. A. F. D., & Buyya, R. (2010). *Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, software: Practice and experience.*

Casanova, H., Giersch, A., Legrand, A., Quinson, M., & Suter, F. (2014, June). Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, *74*(10), 2899-2917. Retrieved from http://hal.inria.fr/hal-01017319

Chen, W., & Deelman, E. (2012). Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *Proceedings of the 2012 ieee 8th international conference on e-science (e-science)* (pp. 1–8). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/eScience.2012.6404430 doi: 10.1109/eScience.2012.6404430

Cho, B., Rahman, M., Chajed, T., Gupta, I., Abad, C., Roberts, N., & Lin, P. (2013). Natjam: design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Socc'13* (p. 6-6).

Dean, J., & Ghemawat, S. (2004). Mapreduce: simplified data processing on large clusters. In *Osdi'04: Proceedings of the 6th conference on symposium on operating systems design and implementation.* USENIX Association.

Dumitrescu, C., & Foster, I. T. (2005). Gangsim: a simulator for grid scheduling studies. In *Ccgrid* (p. 1151-1158). IEEE Computer Society.

Frey, S., & Hasselbring, W. (2011). The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, *4*(3 and 4), 342-353.

Garg, S. K., & Buyya, R. (2011). Networkcloudsim: Modelling parallel applications in cloud simulations. In *Proceedings of the 2011 fourth ieee international conference on utility and cloud computing* (pp. 105–113). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/UCC.2011.24 doi: 10.1109/UCC.2011.24

Hewitt, C. (1972). Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot. *MIT Artificial Intelligence Technical Report 258, Department of Computer Science.*

Khamespanah, E., Sabahi-Kaviani, Z., Khosravi, R., Sirjani, M., & Izadi, M. (2012). Timed-rebeca schedulability and deadlock-freedom analysis using floating-time transition system. In *Proceedings of the 2nd edition on programming systems, languages and applications based on actors, agents, and decentralized control abstractions, agere! 2012, october 21-22, 2012, tucson, arizona, USA* (pp. 23–34). Retrieved from http://doi.acm.org/10.1145/2414639.2414645 doi: 10.1145/2414639.2414645

Kliazovich, D., Bouvry, P., & Khan, S. U. (2012). Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, *62*(3), 1263-1283.

Lim, S.-H., Sharma, B., Nam, G., Kim, E.-K., & Das, C. R. (n.d.). Mdcsim: A multi-tier data center simulation, platform. In *Cluster* (p. 1-9). IEEE.

Núñez, A., Vázquez-Poletti, J. L., Caminero, A. C., Castañé, G. G., Carretero, J., & Llorente, I. M. (2012, March). icancloud: A flexible and scalable cloud infrastructure simulator. *J. Grid Comput.*, *10*(1), 185–209. Retrieved from `http://dx.doi.org/10.1007/s10723-012-9208-5` doi: 10.1007/s10723-012-9208-5

White, T. (2012). *Hadoop, the definitive guide, 3rd edition.* O'Reilly Media.

Wickremasinghe, B., Calheiros, R. N., & Buyya, R. (2010). Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *Proceedings of the 2010 24th ieee international conference on advanced information networking and applications* (pp. 446–452). Washington, DC, USA: IEEE Computer Society.

Appendix A

ReGen User Manual

A.1 Policy Options

A.1.1 Dispatch

No preemption used. If jobs are started they get to finish or their deadlines run out.

A.1.2 Natjam-R eviction

For MDF and MLF, jobs can be preempted for other jobs to run. They are checkpointed at the RM until resources become available. They are then restarted where they left off. A job can be preempted multiple times. If there are no checkpoints to restart, EDF is used to dispatch new jobs.

A.1.3 Dispatch policies

Multiple policies can be selected by holding down the CTRL key and selecting with the mouse.

EDF jobs are dispatched on an earliest deadline first basis.

FIFO jobs are dispatched on a first-in-first-out basis.

MDF jobs are dispatched on a maximum deadline first basis.

Priority jobs have a low or high priority. High priority jobs are dispatched as soon as resources are available. If two or more jobs have the same priority, EDF is used to select between them.

A.1.4 Natjam-R policies

MDF jobs are evicted based on a maximum deadline first policy

MLF jobs are evicted based on a maximum laxity first policy where $laxity = deadline - jobs\ projected\ completion\ time$.

A.1.5 Job arrival patterns

Lists the available job arrival patterns. Multiple job arrival patterns can be selected by holding down the CTRL key.

Bursty Jobs come in bursts with a fixed interval and a separate fixed amount.

Nondet The number of jobs arriving every timeunit is nondeterministic.

Uniform The number of jobs arriving every timeunit is uniform.

Wave Job arrival follows a wave pattern and goes systematically from a fixed lowest point to a fixed maximum point.

Ascending The number of job arrivals ascends from a lowest number to a highest number repeatedly.

Descending The number of job arrivals descends from a highest point to a lowest point. Number of jobs less than 0 is set as 0.

A.1.6 Job length patterns

Lists the available job length types. Multiple job length patterns can be selected by holding down the CTRL key.

Exponential Job length grows exponentially.

Nondet Job length is nondeterministic with a minimum and maximum length.

Uniform Job length is uniform.

Wave Job length follows a wave pattern from a lowest point to a highest point and back down again.

Ascending Job length ascends to a highest point and starts back from the lowest point.

Descending Job length descends from a highest point to a lowest point (minimum 0) and starts back from the highest point.

A.2 Common parameters

Parameters shared between all policies, job arrival patterns and job length patterns.

A.2.1 Max AppMasters

The maximum amount of AppMasters, or concurrent jobs.

A.2.2 Queue size

The size of the incoming and checkpoint queues.

A.2.3 Simulation traces

The number of simulations to run.

A.2.4 Simulation timeunits

The amount of timeunits for each simulation.

A.2.5 Epsilon

The deadline of each job is computed as $job\ length * (1 + epsilon)$.

A.3 Job arrival and length parameters

A.3.1 Burst interval

The amount of timeunits between bursts of jobs.

A.3.2 Burst size

The amount of jobs per burst.

A.3.3 Nondeterministic job arrival/job length

The number of new job arrivals and their lengths can be nondeterministic. Each element has the same probability of selection.

Nondet minimum

The minimum amount.

Nondet maximum

The maximum amount.

A.3.4 Uniform value for job arrival/length

The number of jobs arriving each timeunit and their lengths can be uniform. There are separate values for job arrival and job length.

A.3.5 Wave job arrival/job length

Wave jobs/length per timeunit

The increment or decrement on the wave depending on whether it's ascending or descending.

Wave minimum

The lowest point on the wave.

Wave points

The number of points on the wave. For example jobs/length per timeunit 2, minimum 3 and points 6 generates 3, 5, 7, 9, 7, 5 and repeats.

A.3.6 Ascending job arrival/job length

Ascending increment

Additional jobs/length for each point.

Ascending minimum

The lowest point.

Ascending points

The number of points. For example an increment of 2, minimum 3 and points 4 generates 3, 5, 7, 9 and repeats.

A.3.7 Descending job arrival/job length

Descending decrement

Jobs/length decrement each timeunit.

Descending maximum

The highest point.

Descending points

The number of points. For example a decrement of 2, maximum 9 and points 4 generates 9, 7, 5, 3 and repeats itself. Jobs/length does not go below 0.

A.3.8 Exponential multiplier

Job length is computed as $current\,time\ *\ exponential\,multiplier$. For example an exponential multiplier of 2 will start at timeunit 0 and generate job lengths of 0, 2, 4, 8, 16...

A.4 High priority job options

If two or more jobs are of the same priority, EDF is used to select between them.

A.4.1 Probability %

The probability of a job being of high priority.

A.4.2 Length

The length of high priority jobs. Selecting 0 will give high priority jobs the same length pattern as the low priority jobs.

A.5 Natjam-R options

A.5.1 Checkpoint overhead

The penalty in timeunits for each context switch. This includes both preempting the job and restarting it. The penalty is added at the time of preemption.

A.6 Input/Output Options

A.6.1 Output path

Output directories and artifacts will be created here.

A.6.2 Prefix

A prefix for the name of directories created.

A.6.3 Compiler path

The path to the files needed for compiling the Rebeca code. January 19, 2015, these files are:

- rmc-2.5.0-SNAPSHOT.jar
- g++.exe
- cygiconv-2.dll
- cygintl-3.dll
- cygwin1.dll

A.6.4 Traces path

The traces from a simulation will be output here. Must be on the same hard drive as the Output path.

A.7 Results window

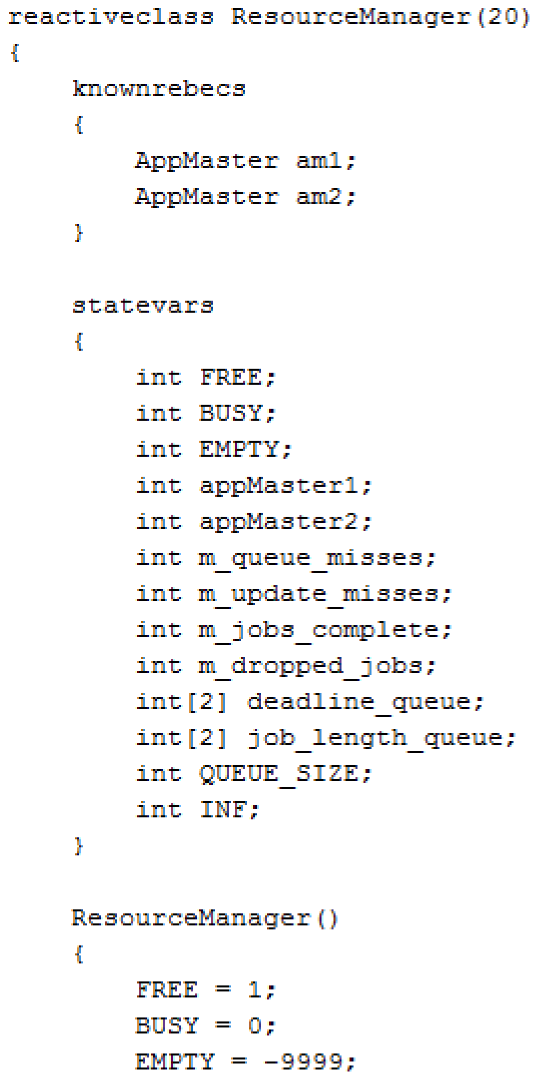
Messages from threads and other components. Threads are named by their Output directory.

A.8 Run button

Runs a test using the set parameters. Is disabled until at least one dispatch policy, one job arrival type and one job length type is selected.
Multiple runs using different parameters can be run simultaneously if there is enough RAM available. Change the output and traces paths to avoid overwriting other results if doing multiple simultaneous runs.

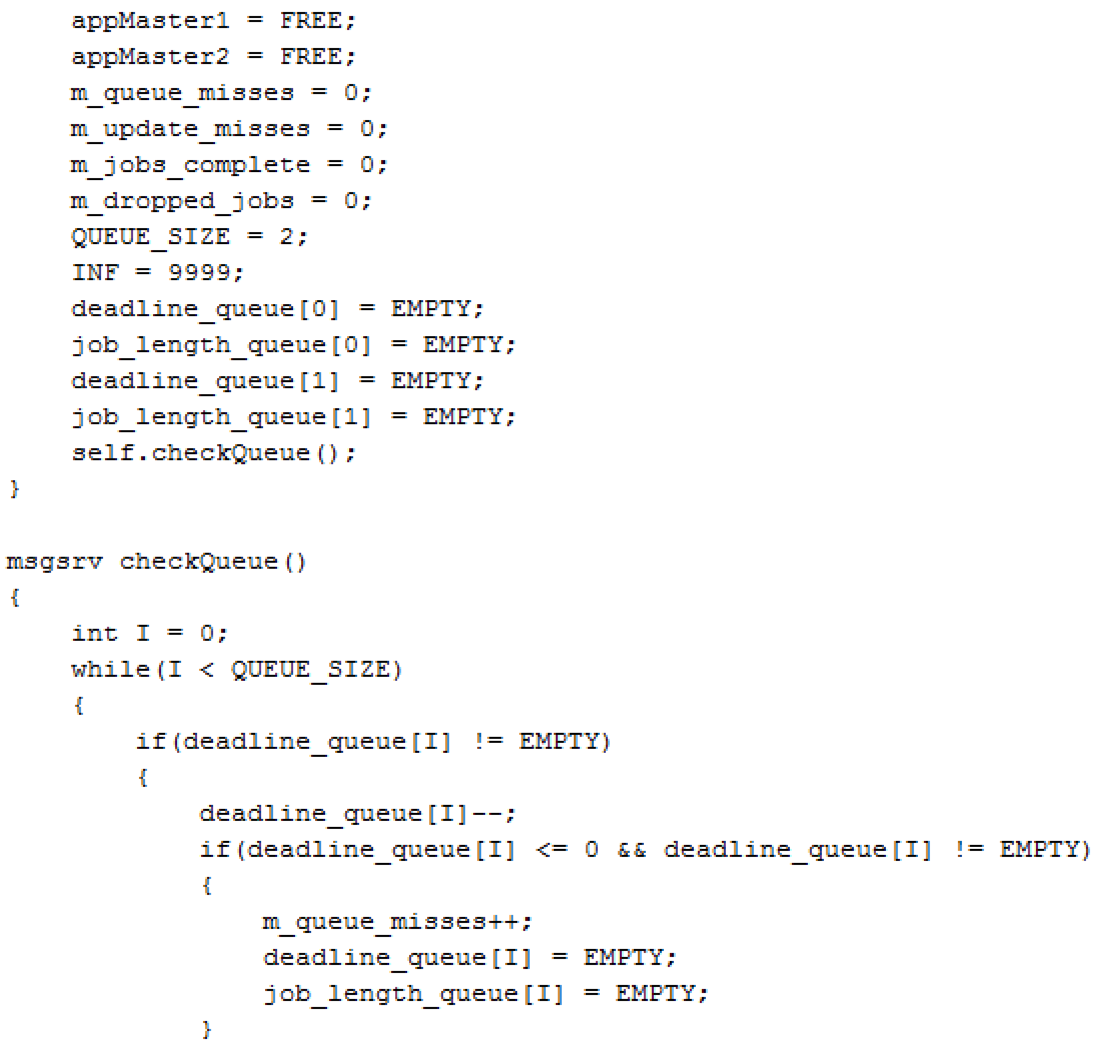
Appendix B

ReGen Generated Code Sample



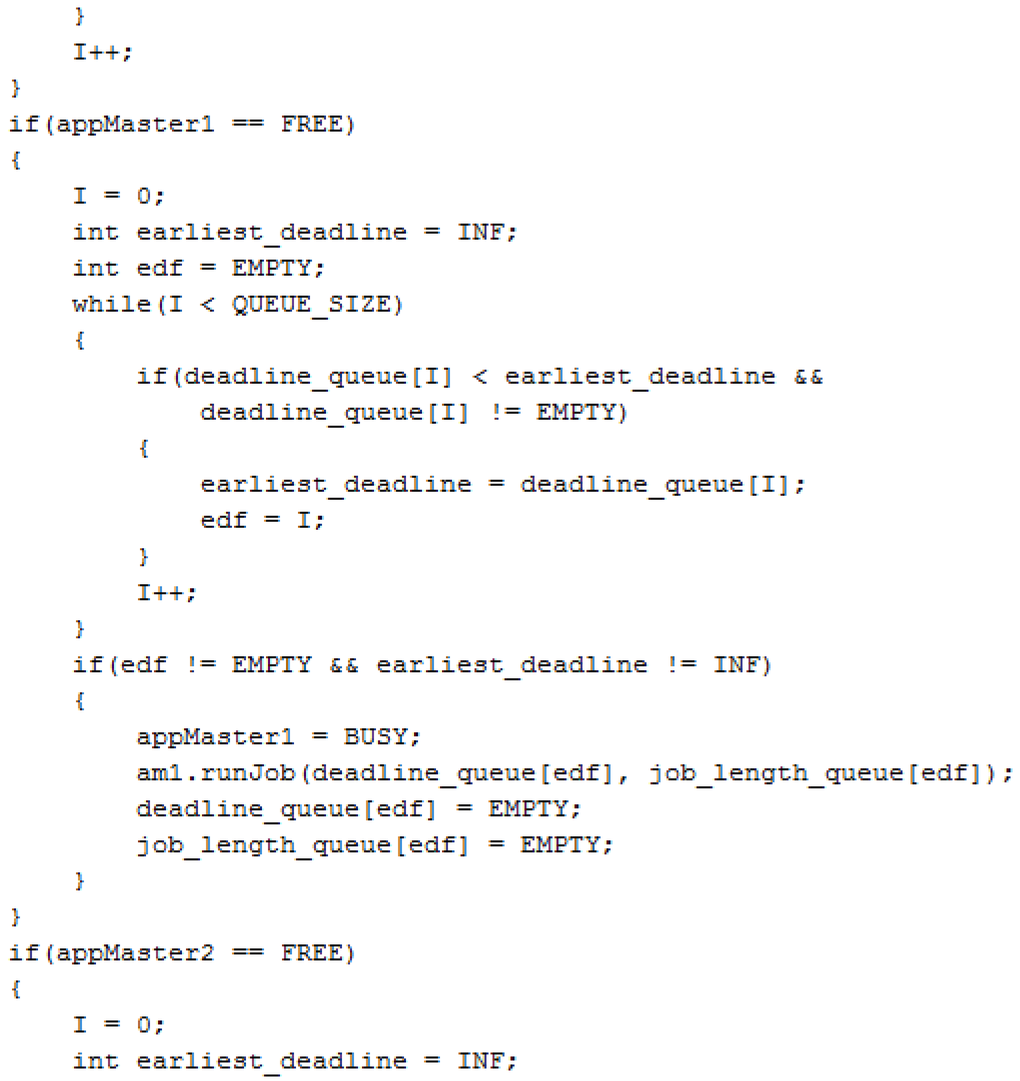
```
reactiveclass ResourceManager(20)
{
    knownrebecs
    {
        AppMaster am1;
        AppMaster am2;
    }

    statevars
    {
        int FREE;
        int BUSY;
        int EMPTY;
        int appMaster1;
        int appMaster2;
        int m_queue_misses;
        int m_update_misses;
        int m_jobs_complete;
        int m_dropped_jobs;
        int[2] deadline_queue;
        int[2] job_length_queue;
        int QUEUE_SIZE;
        int INF;
    }

    ResourceManager()
    {
        FREE = 1;
        BUSY = 0;
        EMPTY = -9999;
```

Figure B.1: Code sample of an EDF dispatch policy model generated by ReGen, part 1.



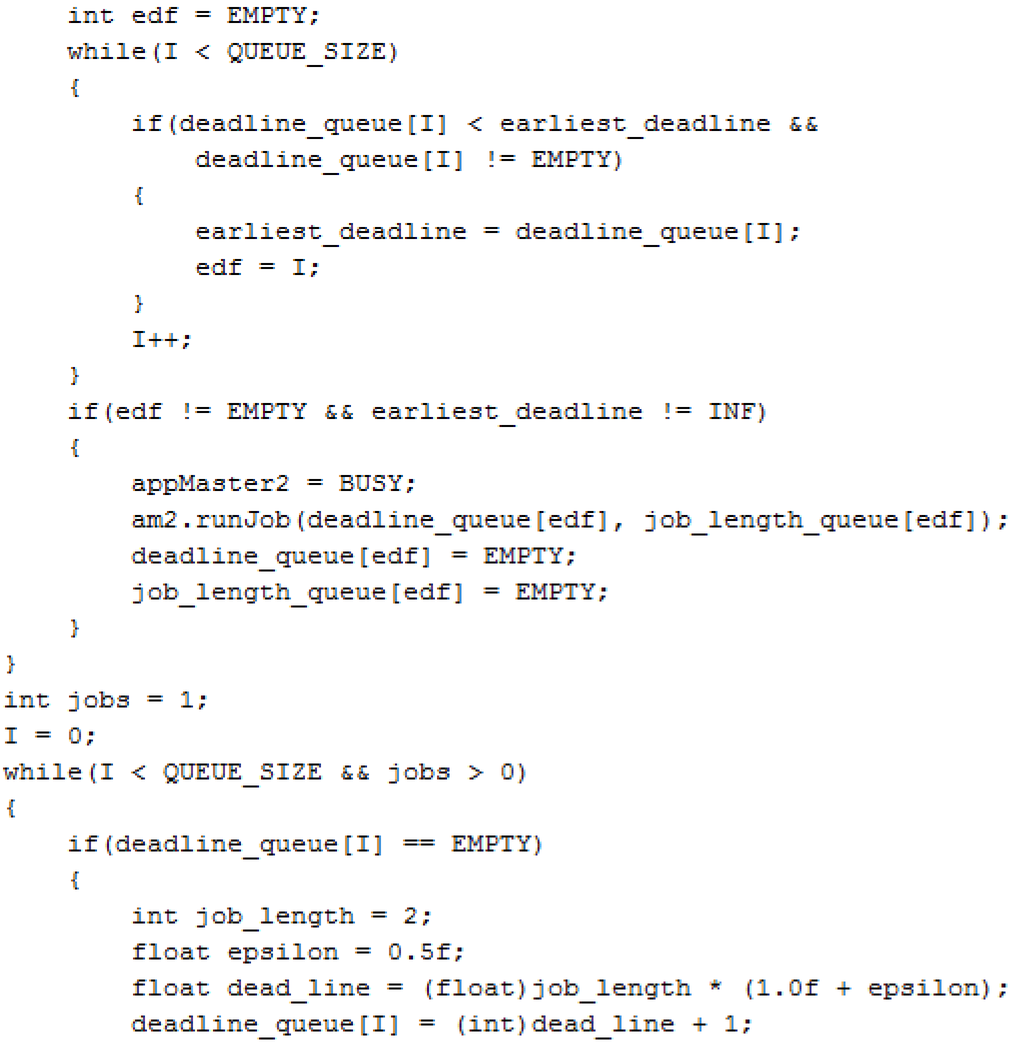
```
    appMaster1 = FREE;
    appMaster2 = FREE;
    m_queue_misses = 0;
    m_update_misses = 0;
    m_jobs_complete = 0;
    m_dropped_jobs = 0;
    QUEUE_SIZE = 2;
    INF = 9999;
    deadline_queue[0] = EMPTY;
    job_length_queue[0] = EMPTY;
    deadline_queue[1] = EMPTY;
    job_length_queue[1] = EMPTY;
    self.checkQueue();
}

msgsrv checkQueue()
{
    int I = 0;
    while(I < QUEUE_SIZE)
    {
        if(deadline_queue[I] != EMPTY)
        {
            deadline_queue[I]--;
            if(deadline_queue[I] <= 0 && deadline_queue[I] != EMPTY)
            {
                m_queue_misses++;
                deadline_queue[I] = EMPTY;
                job_length_queue[I] = EMPTY;
            }
```

Figure B.2: Code sample of an EDF dispatch policy model generated by ReGen, part 2.



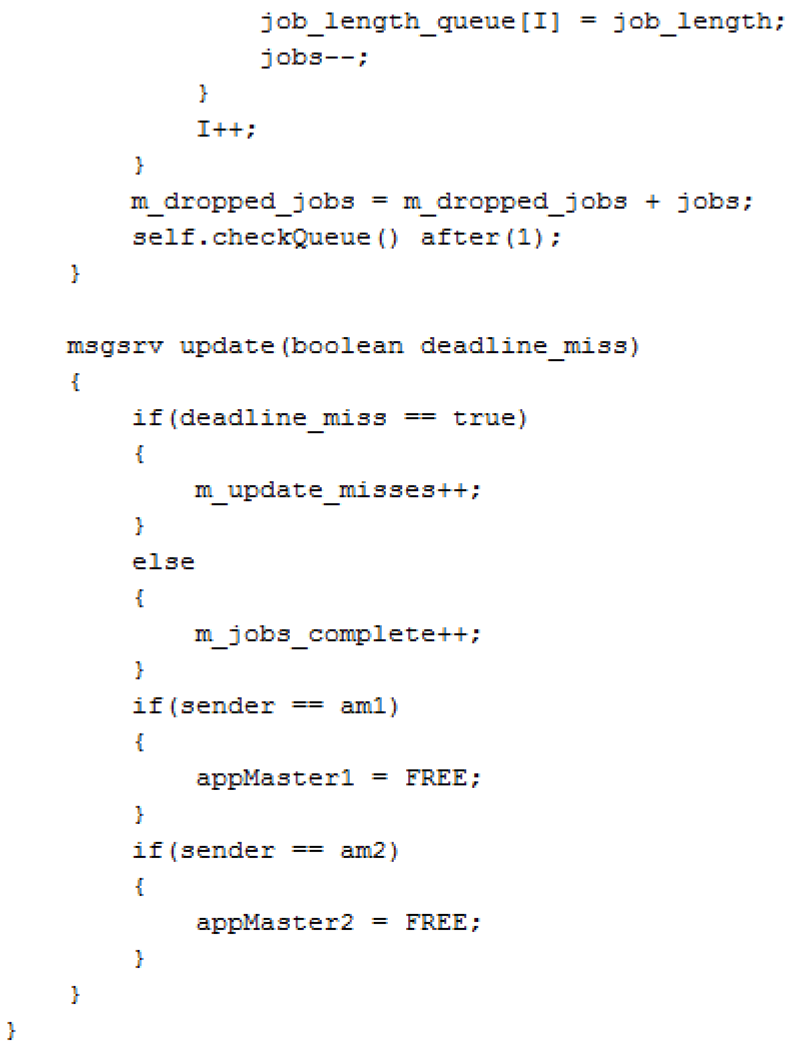
```
    }
    I++;
}
if(appMaster1 == FREE)
{
    I = 0;
    int earliest_deadline = INF;
    int edf = EMPTY;
    while(I < QUEUE_SIZE)
    {
        if(deadline_queue[I] < earliest_deadline &&
            deadline_queue[I] != EMPTY)
        {
            earliest_deadline = deadline_queue[I];
            edf = I;
        }
        I++;
    }
    if(edf != EMPTY && earliest_deadline != INF)
    {
        appMaster1 = BUSY;
        am1.runJob(deadline_queue[edf], job_length_queue[edf]);
        deadline_queue[edf] = EMPTY;
        job_length_queue[edf] = EMPTY;
    }
}
if(appMaster2 == FREE)
{
    I = 0;
    int earliest_deadline = INF;
```

Figure B.3: Code sample of an EDF dispatch policy model generated by ReGen, part 3.



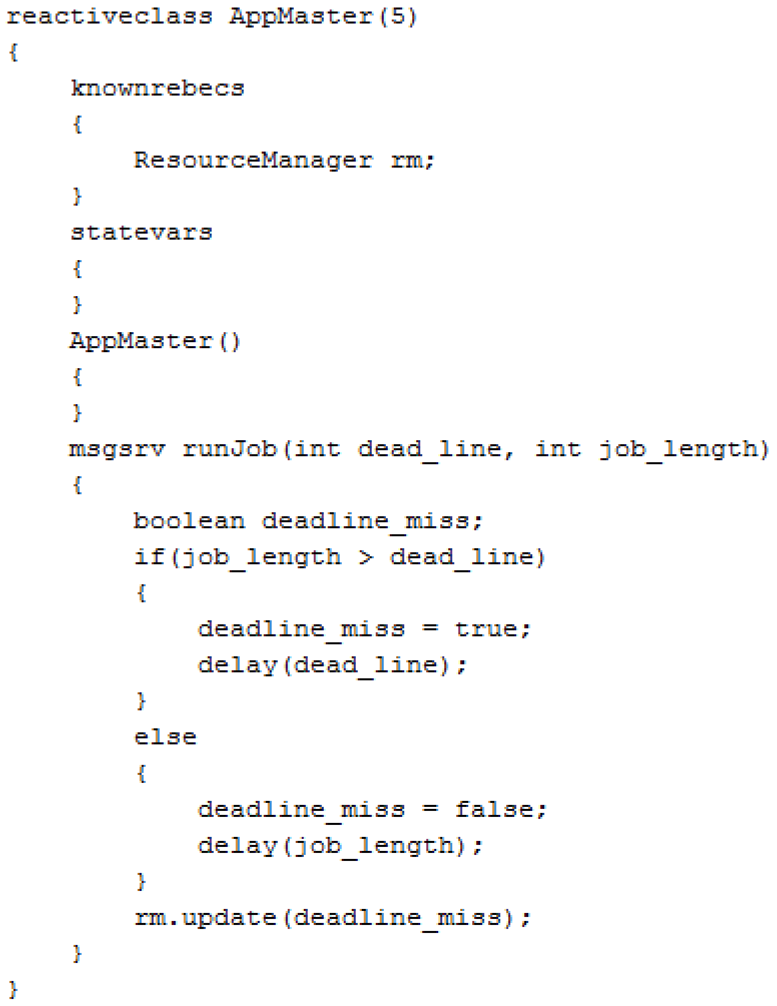
```
    int edf = EMPTY;
    while(I < QUEUE_SIZE)
    {
        if(deadline_queue[I] < earliest_deadline &&
            deadline_queue[I] != EMPTY)
        {
            earliest_deadline = deadline_queue[I];
            edf = I;
        }
        I++;
    }
    if(edf != EMPTY && earliest_deadline != INF)
    {
        appMaster2 = BUSY;
        am2.runJob(deadline_queue[edf], job_length_queue[edf]);
        deadline_queue[edf] = EMPTY;
        job_length_queue[edf] = EMPTY;
    }
}
int jobs = 1;
I = 0;
while(I < QUEUE_SIZE && jobs > 0)
{
    if(deadline_queue[I] == EMPTY)
    {
        int job_length = 2;
        float epsilon = 0.5f;
        float dead_line = (float)job_length * (1.0f + epsilon);
        deadline_queue[I] = (int)dead_line + 1;
```

Figure B.4: Code sample of an EDF dispatch policy model generated by ReGen, part 4.



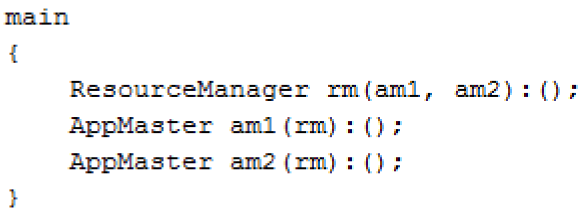
```
                job_length_queue[I] = job_length;
                jobs--;
            }
            I++;
        }
        m_dropped_jobs = m_dropped_jobs + jobs;
        self.checkQueue() after(1);
    }

    msgsrv update(boolean deadline_miss)
    {
        if(deadline_miss == true)
        {
            m_update_misses++;
        }
        else
        {
            m_jobs_complete++;
        }
        if(sender == am1)
        {
            appMaster1 = FREE;
        }
        if(sender == am2)
        {
            appMaster2 = FREE;
        }
    }
}
```

Figure B.5: Code sample of an EDF dispatch policy model generated by ReGen, part 5.



```
reactiveclass AppMaster(5)
{
    knownrebecs
    {
        ResourceManager rm;
    }
    statevars
    {
    }
    AppMaster()
    {
    }
    msgsrv runJob(int dead_line, int job_length)
    {
        boolean deadline_miss;
        if(job_length > dead_line)
        {
            deadline_miss = true;
            delay(dead_line);
        }
        else
        {
            deadline_miss = false;
            delay(job_length);
        }
        rm.update(deadline_miss);
    }
}
```

Figure B.6: Code sample of an EDF dispatch policy model generated by ReGen, part 6.



```
main
{
    ResourceManager rm(am1, am2):();
    AppMaster am1(rm):();
    AppMaster am2(rm):();
}
```

Figure B.7: Code sample of an EDF dispatch policy model generated by ReGen, part 7.

School of Science and Engineering
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539