Rings Parallel Processing System

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Michael Murray

September 28, 2006

Approved for the Division
(Mathematics)

Acknowledgements

I would like to thank Noah Pepper, my friend and confidant throughout this undertaking. I would like to thank Professor Jim Fix and Professor Tom Wieting, who inspired me to raise my standards of logical rigor with their bullet proof lecture styles. I would like to thank my parents, Bill and Kristen Murray for their continuing support for my education. I would like to thank my high school teachers that were not incompetent, including, but not limited to, Charlie Bissell, Craig Childress, Nate Loskutoff, Steve Maskel, Darren Schecter, and Tim Williams. I would like to thank Richard Feynman for reminding me that the universe is an incredibly simple place.

Preface

I first fell in love with Portland, Oregon when I attended the O'Reilly Open Source Convention in the late summer of 2004. A perfect location for such a convention, Portland is a breeding ground for new and revolutionary ideas; a place where anything goes, as long as excessive creativity is involved. Two years later, and here I am working to earn a BS in Mathematics at Reed College, located in South East Portland, and only a short bus ride from that first open source convention I attended.

We were very lucky with the presenters at this particular convention, at which the entire Dyson family, including Freeman, Esther, and George, attended and gave a keynote presentation. I had an interesting encounter with Freeman himself in an elevator on our way down to the convention center in the Marriott Hotel. I had not realized that the silence of the passengers was due to the presence of this amazing physics mind; at the time I had no idea who he was. At the keynote lecture, I witnessed the audience hang on each word spoken by an aging man who I never could have guessed was a powerhouse of ideas for modern science.

But I digress.

Another excellent keynote speaker that made an appearance was Milton Ngan, the Chief Architect from Weta Digital Ltd. In 2004 the latest buzz on the street was the recent release of the final episode in the motion picture retelling of the Lord Of The Rings. Weta Digital was contracted to set up the super-computing facility used by the CG artists to render the stunning visuals seen in the film. Mr. Ngan explained to the highly intrigued audience that an array of 500 computers running the brand new Red Hat Enterprise Linux were employed for this purpose.

Of course, the single most expensive piece of hardware involved in the project was a machine (I believe purchased from Sun Microsystems, though I can't recall if thats correct) used for moderating the network traffic on the massive local area intranet used at the facility. Computers with the ability to maintain such a large network are incredibly costly and impossible to scale. Mr. Ngan mentioned that, even with so much computing power, rendering frames for the movie was still not nearly fast enough to keep up with the work of the graphic artists. The duration was measured in days or weeks, not minutes or hours. So why not use more computers and make it go faster? The limitation on the scale of any super-computing network is always the capacity of the machine that is used to moderate communication. You don't have to take my word for it, just take a look at what Sun Microsystems has to say about grid computing (`http://www.sun.com/service/grid/`). Sun offers "scalable" and "cost effective" solutions to super-computation with their idea of a computing "grid". This means (and again, you can check this out on their website) that one could buy a truck load of really fast computers for about a thousand US dollars a piece which can mount into racks of 20 to 40 computers each. These racks are little intranets all by themselves and the administrator has essentially two choices: 1) Set up each computer as one device and use them each separately. This requires that a human being perform the task of telling each of your 20 or 40 computers what to do and when to do it. 2) Set up each rack with a Sun Microsystems x4600, a 4 core, 64 bit platform designed by sun to maintain a "grid" of less important computers. The x4600 costs twenty six thousand US dollars.

Imagine the reaction of a project manager to the idea that the system administrator has just purchased 20 computers from Sun Microsystems and spent them time required to them set up in a nice compact rack mount system and everything and now the budget must be doubled because it is necessary to purchase a computer to maintain the network. It doesn't take an expert to be shocked that the computer that maintains the network is valued at higher than the total cost of those computers which are actually performing the required computation. The x4600 will most certainly be kept busy telling 20 different computers what computations need to be performed and when, in addition to collecting the output of these calculations and maintaining the communication between each computer. This incredible investment will be, for all intensive purposes, unusable for anything else. Now I want to make it clear that I am not trying to downplay the need for or efficiency of the product Sun produces. I am about to purchase a whole bunch of their computers, but I will not have to come up with about twenty six grand for an x4600 to tell them all what to do. I'm going to get them to tell each other what to do. And this paper is about exactly how I am going to do it.

2004 was the same year that I read a book called Six Degrees by Duncan Watts. This book presented empirical and logical evidence for the hypothesis that every human being is connected to every other by a chain of friends about six people long. In light of this evidence, it is no surprise that rumors (and other types of information) can spread so rapidly by word of mouth. After reading this text, I set out to solve Mr. Ngan's problem with the logic of Mr. Watts (this "logic" is known as graph theory). And I ended up with a system strangely futuristic, perhaps as something you would expect to hear from the keynote speech of a certain Freeman Dyson.

I hope you enjoy reading this story, but I guarantee you wont have as much fun as I have had writing it.

Table of Contents

Chapter 1: The Living Machine **1**
- 1.1 The Mind of the Machine 1
  - 1.1.1 Time 1
  - 1.1.2 A Node and a Parent 1
  - 1.1.3 Changes in State 2
- 1.2 The Node Iteration 3
  - 1.2.1 A change in j 3
  - 1.2.2 A change in s 4
- 1.3 The Parent Iteration 4
  - 1.3.1 A change in j 4
  - 1.3.2 A change in $\mathbf{g}s$ 4
- 1.4 A Recursion Relation for Node State 5
  - 1.4.1 A Node as a Vector 5
  - 1.4.2 From Probability to Action 5
  - 1.4.3 The Iteration Matrix 7

Chapter 2: Introducing the Children to the Family **9**
- 2.1 Conserving j 9
- 2.2 A Fluid Analogy 10

Chapter 3: Programming the Machine **13**
- 3.1 Object Model 13
  - 3.1.1 Node 13
  - 3.1.2 NodeGroup 14
  - 3.1.3 Job 14
  - 3.1.4 JobFactory 15
  - 3.1.5 Connection 15
- 3.2 From Equations to Data Structures 16
- 3.3 So How are we Going to Refer to all This? 16
- 3.4 Talking to the Machine: The Terminal 17
  - 3.4.1 behave 17
  - 3.4.2 close 17
  - 3.4.3 date 18
  - 3.4.4 dbs start 18
  - 3.4.5 dbs create 18

3.4.6 dbs add . . . 18
3.4.7 giterate . . . 19
3.4.8 inputrate . . . 19
3.4.9 jobtime . . . 20
3.4.10 open . . . 20
3.4.11 peers . . . 20
3.4.12 run test . . . 21
3.4.13 sendtask . . . 21
3.4.14 set . . . 21
3.4.15 status . . . 22
3.4.16 tasks . . . 22
3.4.17 threads . . . 23
3.4.18 uptime . . . 23

Chapter 4: The Lonely Node (Experimental Results) . . . **25**
4.1 Varying J_p . . . 25
4.2 Varying A_s . . . 26
4.3 What Does This Mean? . . . 27

Chapter 5: Using the Machine . . . **29**
5.1 Parallel Processing . . . 29
5.1.1 Other Products and Tools . . . 30
5.1.2 What Makes Rings Different? . . . 30
5.2 Implementing a Task . . . 31
5.2.1 Tasks and Jobs . . . 31
5.2.2 The JobFactory Interface . . . 31
5.2.3 The Job Interface . . . 31
5.2.4 An Example: URL Profiling . . . 33
5.3 Producing Output . . . 40
5.3.1 The JobOutput Class . . . 40
5.3.2 The Client Class . . . 41
5.3.3 The OutputHandler Interface . . . 42
5.3.4 The OutputServer Class . . . 43
5.3.5 Revising the Example . . . 43
5.4 Putting it all Together . . . 45
5.4.1 Running a Task . . . 45
5.4.2 Viewing Node Status . . . 46

Chapter 6: Extending the Machine . . . **49**
6.1 Decentralized Data Storage . . . 49
6.2 Ideas for Extending the Engine . . . 50
6.2.1 A Parallel Operating System . . . 51
6.2.2 A Decentralized Social Networking System . . . 51
6.3 Ideas for Parallel Processing Tasks . . . 52

Chapter 7: The Product . . . **53**

7.1 Plans for the Future 53
7.2 Licensing the Machine 53

Appendix A: Glossary 55

Appendix B: Frequencies and Rates 57

Appendix C: Why Avoid Java Serializable and Externalizable? . . . 59

Appendix D: Configuring and Running the Rings Client 61

Appendix E: Relational Database Tables for DefaultOutputHandler 63

List of Tables

1.1 Node and Parent Properties 2

3.1 Variables and Functions Represented in the Object Model 16
3.2 Variables accessible by the set command 22

4.1 Node properties for varying J_p test 26
4.2 Node properties for varying J_p test 28

5.1 Methods required by the JobFactory interface 48

List of Figures

1.1 A vector describing the state of a node 5
1.2 The iteration matrix 7

2.1 A graduated cylinder, or “bucket” 10
2.2 A rate equation for j 11

3.1 Data Structures in the Java Implementation 14
3.2 The Job and JobFactory Data Structures 15
3.3 The Rings Network Client Terminal: Logging into a Node 17

4.1 Definition of activity rating 25
4.2 Activity Rating for Different Values of J_p 25
4.3 Activity Rating for Different Values of A_s 27
4.4 Activity Rating for More Values of A_s 28

5.1 The JobFactory interface 32
5.2 The Job interface 32
5.3 Example of an encoded task and an encoded job 33
5.4 The run method of the UrlProfilingJob class 34
5.5 The set method of the UrlProfilingJob class 35
5.6 The encode method of the UrlProfilingJob class 36
5.7 The getTaskId method of the UrlProfilingJob class 36
5.8 The getTaskString method of the UrlProfilingJob class 36
5.9 Constructor for the UrlProfilingJob class 37
5.10 The nextJob method of the UrlProfilingTask class 37
5.11 The set method of the UrlProfilingTask class 37
5.12 The encode method of the UrlProfilingTask class 38
5.13 The createJob method of the UrlProfilingTask class 38
5.14 The getTaskId method of the UrlProfilingTask class 38
5.15 The getName method of the UrlProfilingTask class 39
5.16 The getPriority and setPriority methods of the UrlProfilingTask class 39
5.17 The getCompleteness and isComplete methods of the UrlProfiling-Task class 39
5.18 Classes for Storing Output 40
5.19 The JobOutput class 41
5.20 The Client class 42
5.21 The OutputHandler interface 43

5.22 The OutputServer class 44
5.23 The run method of the UrlProfilingJob class 45
5.24 An OutputHandler implementation for UrlProfilingJob 46
5.25 Sending the UrlPofilingTask from the terminal 47
5.26 Configuring the clients to report output 47
5.27 Automatically generated status page 47

Abstract

This paper introduces the theory and implementation of the Rings Parallel Processing System, a tool for load balancing in a decentralized network of computers used to perform computationally intensive tasks. The implementation presented is written in Java. Chapters one and two discuss the mathematical model that is the basis for the load balancing technique, chapter three describes the object model of the Java implementation and explains how the model is represented on the computer, chapter four presents experimental data, chapter five explains how to use the software, chapter six summarizes some ways to extend the product, and chapter seven discusses plans for the project and licensing.

Chapter 1

The Living Machine

1.1 The Mind of the Machine

I plan to outline here a concept that is subject to change and reinterpretation, but that I believe to be essential to understanding a system that was first observed during the summer of 2005 in Palo Alto, California.

I use the term "Living Machine" to mean a non-deterministic **stochastic** process operating on a finite state system. The "system" in our case is a network of connected elements called nodes. The connected structure is known as a graph.

This text contains many abstractions and ideas that seem to have no foundation in reality. Forgive me for my idealism and take my word that everything contained here is intimately involved in every connected structure used for information exchange. I have always believed that one can learn a lot by pretending to know more than is known. For the purpose of this paper, please do your best to pretend that you know what I am talking about. You may discover that you end up knowing more than you thought. If a word confuses you, try looking it up in the glossary (Appendix A). Words or phrases in **bold face** are defined there.

1.1.1 Time

Throughout this document, I will make reference to the concept of time. Time in this case has a particularly abstract meaning. For our purposes, time "happens" in discrete increments and each of these increments has a "before" and an "after". Before a time increment, the state of a connected structure may have one configuration, and after the time increment that configuration may have changed. This document is an explanation of a certain set of rules for what alterations can occur during an interval of time. In other words, what differences there may be in the "before" state and the "after" state.

1.1.2 A Node and a Parent

A node is the fundamental element of a connected structure. Nodes maintain reference to zero or more other nodes and together these connected nodes create what is known as a graph. For our purposes, a node is a queue of items. It is unimportant to define what an item is, but it is safe to say that every node contains a certain positive integer number of them, possibly zero. I will denote a node by a bold letter, such as $\mathbf{N}$, and the number of items in the queue of that node by $\mathbf{N}_j$.

The state of a node involves some other quantities which are listed in the table below along with the scheme that will be adopted for denoting these properties. I will most often use the short names of the properties, as I will spend much time talking about one node in particular, and rarely need to distinguish between properties of different nodes at the same time.

Every node has what I call a "parent". This has absolutely nothing to do with the graph or connections between nodes. A "parent" is something entirely separate, though it is essentially a node in and of itself, which is only connected to the one node that names it as a parent. Each node has one parent and the parent has many of the same properties as the node that it corresponds to. Parent properties of a node will be denoted with the bold letter $\mathbf{g}$ prepended. I may also prepend the name of a function with $\mathbf{g}$ if the function pertains to the parent. This is noted in the table below.

Property	Short Name	Parent Property	Description
$\mathbf{N}_j$	j	$\mathbf{g}j$	The number of items stored by a node $\mathbf{N}$
$\mathbf{N}_{Jm}$	J_m	N/A	The number of items desired by a node $\mathbf{N}$
$\mathbf{N}_{Jp}$	J_p	N/A	The percentage of items that will not be relayed by a node $\mathbf{N}$
$\mathbf{N}_{Jt}$	J_t	N/A	The time constant for a node $\mathbf{N}$
$\mathbf{N}_{Rp}$	R_p	$\mathbf{g}R_p$	The relay probability for a node $\mathbf{N}$
$\mathbf{N}_k$	k	N/A	The number of other nodes connected to a node $\mathbf{N}$ in the graph.
$\mathbf{N}_s$	s	$\mathbf{g}s$	The sleep time of a node $\mathbf{N}$
$\mathbf{N}_{Ac}$	A_c	N/A	The activity coefficient for a node $\mathbf{N}$
$\mathbf{N}_{As}$	A_s	N/A	The activity sleep coefficient for a node $\mathbf{N}$
$\mathbf{N}_{Ao}$	A_o	N/A	The activity offset value for a node $\mathbf{N}$
$\mathbf{N}_{\mathbf{g}k}$	N/A	$\mathbf{g}_k$	The number of item generators for a node $\mathbf{N}$

Table 1.1: Node and Parent Properties

1.1.3 Changes in State

The goal of chapters one and two of this text is to model the change in state of a collection of nodes over discrete intervals of time. There are two ways that the state of a node or its parent may change during an interval of time, denoted as Δt:

- A node may choose to change its own state. The probability of this event during an interval of time is given by $\Delta t/s$. This event is referred to as the "node iteration".
- The parent of a node may choose to change its state. The probability of this event during an interval of time is given by $\Delta t/\mathbf{g}s$. This event is referred to as the "parent iteration".
- A node may have its state changed by another node that it is connected to in the graph of nodes. The probability of this event is related to the state of the nodes that are connected to the node in question.

1.2 The Node Iteration

The term "Node Iteration" refers to one process by which a node changes its own state. The node iteration occurs during an interval of time with a probability given by $\Delta t/s$. Three separate actions occur, one after another, during the node iteration. Each of these is listed and described in detail below.

1.2.1 A change in j

During the node iteration the value of j may decrease by one. The probability of this happening is given by a function I will call ρ.

$$\rho(\mathbf{N}) = \rho_0 + \frac{j}{J_m - 1} \cdot \begin{cases} 2R_p & j > J_m, \\ R_p & j > J_p \cdot J_m, \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

This value could be greater than 1.0. From now on, I will assume that a probability of greater than 1.0 is equal to a probability of exactly 1.0. ρ gives the probability of an event, not a decision, but a prediciton. However, if the event that ρ predicts does in fact occur during an interval Δt, another event may occur with a probability given by $\mathbf{g}\rho$.

$$\mathbf{g}\rho(\mathbf{N}) = \begin{cases} 1.0 & k = 0, \\ \mathbf{g}R_p & \text{otherwise} \end{cases} \tag{1.2}$$

Again, this gives the probability that an event occurs and is not decisive. This probability is dependent on the event predicted by ρ actually occurring.

There exists one other way in which the value of j may decrease during an interval Δt, and the probability of this event is independent of the two events just mentioned. The probability that j decreases due to this third type of event is given by the ratio of the interval Δt to the time constant for the node:

$$\frac{\Delta t}{J_t} \tag{1.3}$$

In this special case, unlike any other probability equation given throughout this text, values above 1.0 correspond to multiple events occurring. What I mean is that, during the node iteration, j will always be decreased by $\lfloor \frac{\Delta t}{J_t} \rfloor$ and possibly by decreased by an additional 1.0, with probability given by $\frac{\Delta t}{J_t}$ mod 1.0.

1.2.2 A change in s

During the node iteration the value of s will change according to the following equation. s' refers to the value of s after the node iteration and s refers to the value initially.

$$s' = s \cdot \frac{A_s}{1 - J_p + (j - J_p \cdot J_m)/(A_c \cdot J_m) + A_o} \tag{1.4}$$

1.3 The Parent Iteration

The term "Parent Iteration" refers to the second process by which a node changes its own state. The parent iteration occurs during an interval of time with a probability given by $\Delta t/\mathbf{g}s$. Three separate actions occur, one after another, during the node iteration. Each of these is listed and described in detail below.

1.3.1 A change in j

During the parent iteration there are two ways that the value of j can increase. First, j will always increase by $\mathbf{g}_k$ during the parent iteration, and will increase by an additional one if $\mathbf{g} > 0$.

$$j' = j + \mathbf{g}_k + \begin{cases} 1 & \mathbf{g}j > 0 \\ 0 & \text{Otherwise} \end{cases} \tag{1.5}$$

If $\mathbf{g}j > 0$, $\mathbf{g}j$ will be decreased by one as j is increased. This may be a good time to mention the fact that the sum total value of j for all nodes in a graph is always conserved, except for the addition of $\mathbf{g}_k$ during the parent iteration.

$$\mathbf{g}j' = \mathbf{g}j - \begin{cases} 1 & \mathbf{g}j > 0 \\ 0 & \text{Otherwise} \end{cases} \tag{1.6}$$

1.3.2 A change in $\mathbf{g}s$

During the parent iteration the value of $\mathbf{g}s$ will change according to the following equation. $\mathbf{g}s'$ refers to the value of $\mathbf{g}s$ after the node iteration and $\mathbf{g}s$ refers to the value initially.

$$\mathbf{g}s' = \mathbf{g}s \cdot (1 + \frac{j - J_p \cdot J_m}{A_c \cdot J_m} + A_o) \tag{1.7}$$

1.4 A Recursion Relation for Node State

To analyze the formulas presented I will need to construct a **recursion relation** that completely describes the way that node state changes over the course of many discrete increments of time, Δt. I will begin by constructing **vector quantity** which will be used to store the state of a node, $\mathbf{N}$.

1.4.1 A Node as a Vector

I will represent the state of a node $\mathbf{N}$ as a 6D vector quantity. There are, of course, many more than 6 independent variable quantities associated with the node (as listed in the table above), but I will choose to analyze only the quantities that are effected by the node and parent iterations just described. For the purpose of constructing a recursion relation for node state I will introduce two quantities not previously mentioned. These are Δs and $\Delta \mathbf{g}s$. Δs refers to the duration of time since the last node iteration occurred and $\Delta \mathbf{g}s$ refers to the duration of time since the last parent iteration. The vector representation of node state is given below.

$$\begin{pmatrix} j \\ s \\ \Delta s \\ \mathbf{g}j \\ \mathbf{g}s \\ \Delta \mathbf{g}s \end{pmatrix} \tag{1.8}$$

Figure 1.1: A vector describing the state of a node

1.4.2 From Probability to Action

So far I have provided a detailed mathematical description for the probability of the events the occur during the node and parent iterations. However, to describe the system in terms a **recursion relation** I will need to define functions that determine whether or not the events do occur, based on these probabilities. To do this I will often make use of ζ, a uniform random number in the range $[0, 1)$. I will denote each unique random variable of this sort with a subscript, for example ζ_1, ζ_2, etc.

Enacting The Node Iteration

$$\gamma(\mathbf{N}, \Delta t) = \begin{cases} 1 & \Delta t \geq \mathbf{N}_s - \mathbf{N}_{\Delta s} \\ 0 & \text{otherwise} \end{cases} \tag{1.9}$$

The γ function gives 1 if the node iteration occurs during Δt and 0 if it does not. The function makes this decision by checking if the duration since the last node iteration plus the duration Δt is equal to or greater than the sleep time for the node, $\mathbf{N}$.

$$\delta(\mathbf{N}) = \begin{cases} 1 & \zeta_1 < \rho(\mathbf{N}) \\ 0 & \text{otherwise} \end{cases} \tag{1.10}$$

The δ function gives one if the value of j decreases by one due to the event predicted by the ρ function, given above. Recall that ζ_1 is a uniform random variable.

Enacting the Parent Iteration

$$\gamma_{\mathbf{g}}(\mathbf{N}, \Delta t) = \begin{cases} 1 & \Delta t \geq \mathbf{N}_{\mathbf{g}s} - \mathbf{N}_{\mathbf{g}\Delta s} \\ 0 & \text{otherwise} \end{cases} \tag{1.11}$$

The $\gamma_{\mathbf{g}}$ function gives one if the parent iteration occurs during Δt and 0 if it does not. The function makes this decision by checking if the duration since the last parent iteration plus the duration Δt is equal to or greater than the parent sleep time for the node, $\mathbf{N}$.

1.4.3 The Iteration Matrix

Because the node state is represented as a six dimensional vector, a matrix product which is a function of the previous node state can be used to describe the iterative process explained in sections 1.2-3. The node state after a time interval Δt (denoted $\mathbf{N}'$) is based on the previous node state, $\mathbf{N}$. This involves the product of two functions: A vector valued function $\Gamma(\mathbf{N}, \Delta t)$ and a matrix valued function $\Omega(\mathbf{N}, \Delta t)$. The product is formed:

$$\mathbf{N}' = \Omega(\mathbf{N}, \Delta t) \cdot \Gamma(\mathbf{N}, \Delta t) \tag{1.12}$$

The $'$ denotes the value of a variable after the time increment Δt. Below I have expanded this product using values for Γ and Ω that are described in sections 1.2-3 so that the matrix product represents the recursive process used to update each component of the node state vector. This is shown below.

$$\mathbf{N}' = \begin{pmatrix} j' \\ s' \\ \Delta s' \\ \mathbf{g}j' \\ \mathbf{g}s' \\ \Delta \mathbf{g}s' \end{pmatrix} = \tag{1.13}$$

$$\begin{pmatrix} -\delta(\mathbf{N}) - \lfloor \frac{\Delta t}{J_t} \rfloor - \left[1 \text{ if } \zeta_2 < (\frac{\Delta t}{J_t} \mathrm{mod} 1) \right] & \mathbf{g}_k + [1 \text{ if } \mathbf{g}j > 0] & j \\ s \cdot \frac{A_s}{1 - J_p + (j - J_p \cdot J_m)/(A_c \cdot J_m) + A_o} - s & 0 & s \\ -s & 0 & \Delta t + \Delta s \\ \delta(\mathbf{N}) \text{ if } \zeta_3 < \mathbf{g}R_p & -1 \text{ if } \mathbf{g}j > 0 & \mathbf{g}j \\ 0 & \mathbf{g}s \cdot (\frac{j - J_p \cdot J_m}{A_c \cdot J_m} + A_o) & \mathbf{g}s \\ 0 & -\mathbf{g}s & \Delta t + \Delta \mathbf{g}s \end{pmatrix} \cdot \begin{pmatrix} \gamma(\mathbf{N}, \Delta t) \\ \gamma_{\mathbf{g}}(\mathbf{N}, \Delta t) \\ 1 \end{pmatrix}$$

Figure 1.2: The iteration matrix

By repeatedly applying the iteration matrix given above for a relatively small time increment, Δt, I can see how the state of a node changes over a long period of time. For this recursion relation to properly predict the change in state of a node over time, Δt must be smaller than the value of s or $\mathbf{g}s$.

Chapter 2

Introducing the Children to the Family

So far I have completely described the way in which the state of a node may change due to the internal workings of that node. However, I have not made any mention of the impact of a node on the other nodes in the graph. For this purpose I will introduce the concept of a node family, defined as follows. The immediate family of a node, $\mathbf{N}$, is a set containing the collection of nodes connected to $\mathbf{N}$ in the graph. I will denote the family of a node with a subscript capital K. For example, the family of $\mathbf{N}$ is denoted $\mathbf{N}_K$.

2.1 Conserving j

It was briefly mentioned earlier that the sum total value of $j + \mathbf{g}j$ for all nodes in a graph is conserved except for the possible increase in j due to the parent iteration (when $\mathbf{g}_k > 0$) and the possible decrease in j due to the node iteration (when $\frac{\Delta t}{J_t} > 1$).

In the case when the node iteration occurs and $\delta(\mathbf{N}) > 0$, the value of j for the node decreases. For the sum total of $j + \mathbf{g}j$ for all nodes in the graph to be conserved in this scenario, there must be a corresponding increase in the value of j or $\mathbf{g}j$ for $\mathbf{N}$ or some other node in the graph. I have already mentioned that if j decreases in this way, the value of $\mathbf{g}j$ for that node may increase, with a probability given by $\mathbf{g}R_p$. This is shown in the **iteration matrix**, but let us jog our memory and take another look.

$$\delta(\mathbf{N}) \text{ if } \zeta_3 < \mathbf{g}R_p \tag{2.1}$$

In this case, ζ_3 is the uniform random variable that "determines" whether the value of $\mathbf{g}j$ will increase by the amount $\delta(\mathbf{N})$. If the condition for this increase does not hold, meaning $\zeta_3 \geq \mathbf{g}R_p$, the sum total value of $j + \mathbf{g}j$ for the node $\mathbf{N}$ will not be conserved.

In this case, the node will act in the following way:

- Choose a uniform random node $\mathbf{M} \in \mathbf{N}_K$.
- Increase the value of $\mathbf{M}_j$ by the value $\delta(\mathbf{N})$.

And this is what the family of a node is for: To conserve the sum total value of $j + \mathbf{g}j$ for the graph by accepting an increase in j when a family member experiences a node iteration that decreases j.

2.2 A Fluid Analogy

To provide a more concrete idea of the abstraction presented in chapter one, I would like to propose an analogy for the graph involving fluids. I will venture to say that a node in a graph is analogous to a bucket (or "graduated cylinder", shall I say to be precise). This bucket is filled with a certain amount of water, which one can measure by looking at the gradation markings on the cylinder.

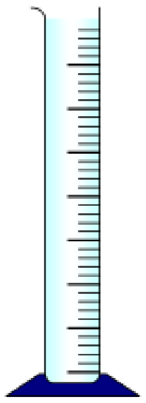

Figure 2.1: A graduated cylinder, or "bucket"

Each one of these horizontal lines on the bucket corresponds to a value of j for the node that the bucket represents. A higher water level in the bucket indicates a larger value of j. Two nodes that are connected to one another in the graph can be imagined as two buckets connected to each other by a complicated sort of pump. This "pump" is analogous to the functionality described in the previous section, where a node may increase the value of j for one of the nodes that it maintains a connection to in the graph.

This theoretical "pump" operates at a rate described by the frequency that the node iteration occurs for the two nodes that are connected to one another. If I simplify and assume, for the moment, that $R_p = 1$, $\mathbf{g}R_p = 0$ and $j > J_m$ for both nodes, $\mathbf{N}$ and $\mathbf{M}$, I can proceed to give a differential equation for the rate of change of $\mathbf{M}_j$ due to the connection between $\mathbf{N}$ and $\mathbf{M}$.

$$\frac{\partial \mathbf{M}_j}{\partial t} = \frac{1}{\mathbf{N}_s(t)} - \frac{1}{\mathbf{M}_s(t)} \tag{2.2}$$

Figure 2.2: A rate equation for j

The functions $\mathbf{N}_s(t)$ and $\mathbf{M}_s(t)$ give the value of s for each node at the time t. These functions are defined by the recursion relation given in the previous chapter. To evaluate $\mathbf{N}_s(t)$, one must recursively apply the iteration matrix calculation for a small interval Δt. There is an important difference between a rate (the left side of equation 2.2) and a frequency (the right side of equation 2.2). For more information on this difference, see appendix B.

Chapter 3

Programming the Machine

Thus far I have described a fairly abstract mathematical construction and provided basically no explanation for the equations in the first two chapters. Now I will provide an example of our construction in the context of information exchange, as it was described in the introduction. To accomplish this I will present the object model for a Java program that makes use of the mathematical model presented in chapters one and two for the purpose of automated load balancing in a parallel processing network across the internet.

I singlehandedly wrote the Java code for the entire implementation of the software I am about to describe. However, my design decisions were heavily impacted by the influence of my colleagues at Reed College and Stanford University. Frequently throughout the rest of the document I will speak in the first person plural, making reference to "we", as oppose to just you the reader and myself. This "we" is meant to include those that I mentioned in the acknowledgments, as all of my peers in the academic community have contributed in some way to the fulfillment of this goal.

3.1 Object Model

Java is an object oriented programming language, meaning that a program is written by constructing data structures that represent real world entities and developing functions that operate on these data structures. There are 5 data types (known as "classes") that represent the mathematical constructs presented thus far. Many more data types and functions are required for the program to be useful and perform the tasks that are required from a parallel processing system, but they are not important for this discussion and will be presented later.

3.1.1 Node

The Node class is one part of the representation of a node in the graph. The Node class keeps track of the properties in the second column of table 1.1 (none of the parent properties). The Node class is responsible for performing the node iteration at the appropriate time. In the context of the vector representation of node state,

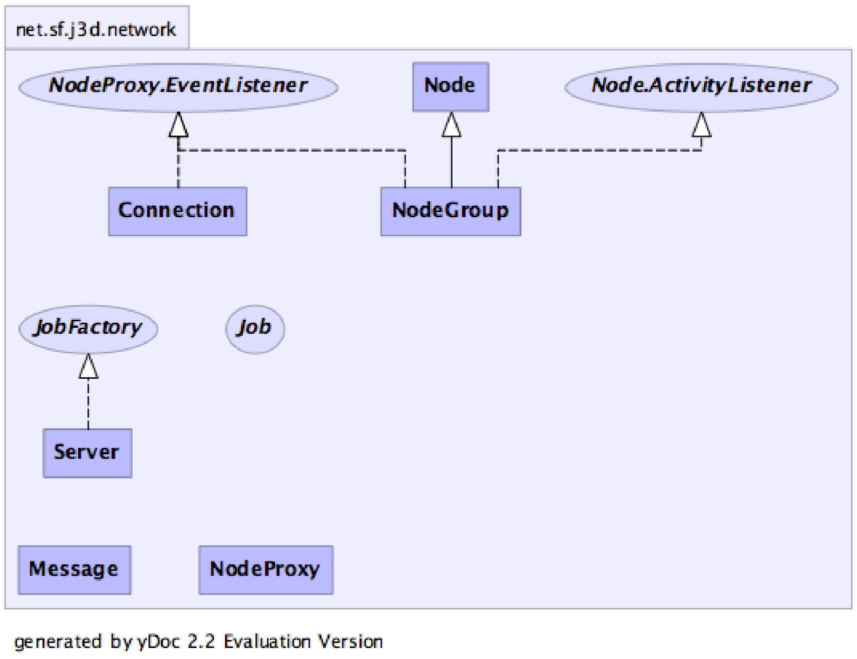

Figure 3.1: Some of the data structures employed by the Java implementation of the model described thus far. The dashed lines connecting object types show the inheritance of types. For example, a NodeGroup object is a type of Node object, as denoted by the solid line connecting these two types.

the Node class represents the first 3 elements of the state vector presented in figure 1.1.

3.1.2 NodeGroup

The NodeGroup class is the other part of the representation of a node in the graph. The NodeGroup class keeps track of the properties in the third column of table 1.1 (the parent properties). The NodeGroup class is responsible for performing the parent iteration at the appropriate time. In the context of the vector representation of node state, the NodeGroup class represents the last 3 elements of the state vector presented in figure 1.1.

3.1.3 Job

The Job interface represents an item attached to a node. You may recall from table 1.1 that j was defined as the number of items for a node. In the Java implementation a Node object maintains j of these Job objects. The Job objects can be moved from node to node across the graph due to the node iteration, which is performed by the Node class.

In the Java implementation, the Job interface extends the Java Runnable interface. This means that a Job object, along with being an “item” stored by a node,

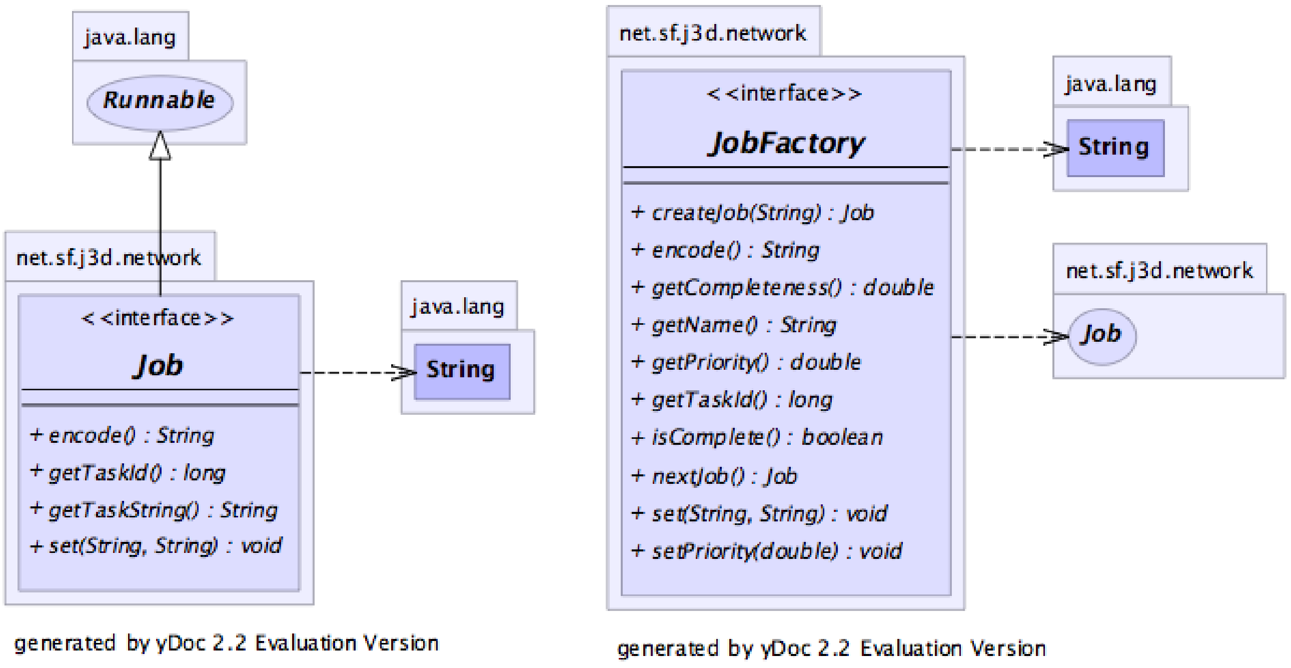

Figure 3.2: The Job and JobFactory Data Structures

is a set of instructions for a computer to execute. The value of J_t given in table 1.1 actually corresponds to the time required to complete the instructions contained in the Job object when it runs on the computer that hosts the node that references it as an item.

3.1.4 JobFactory

You may recall from table 1.1 that $\mathbf{g}_k$ was defined as the number of item generators for a node. The term "item generator" is used because $\mathbf{g}_k$ is related to the frequency that j is increased by the parent iteration. A JobFactory object represents an "item generator". In the Java implementation a NodeGroup object maintains $\mathbf{g}_k$ of these JobFactory objects.

In the Java implementation, the JobFactory interface provides a nextJob function, which returns a new Job object. During the parent iteration performed by the NodeGroup class, a Job object from each JobFactory is requested and these new Job objects are then added to the list of items maintained by the Node class.

3.1.5 Connection

The Connection class is used to represent connections between nodes in the graph. If two nodes are connected, the Node objects that represent them will each maintain a Connection object. This Connection object is used to send and receive Job objects during the node iteration.

In the Java implementation the Connection class makes use of another data structure called a NodeProxy. The NodeProxy handles moving information around the internet so that each node may be hosted by a Node object on a different computer system.

3.2 From Equations to Data Structures: From Math to Science

It may be helpful to provide a more systematic comparison of the mathematical model and the Java implementation. For this, I have provided the table below, which lists the data structures provided in the object model and which variables and functions are represented by each.

Class	Variables	Functions
Node	j, s, J_m, J_p, R_p, k, A_c, A_s, A_o, $\mathbf{g}R_p$	$\rho(\mathbf{N})$, $\mathbf{g}\rho(\mathbf{N})$, s', $\gamma(\mathbf{N}, \Delta t)$, $\delta(\mathbf{N})$
NodeGroup	$\mathbf{g}j$, $\mathbf{g}s$, $\mathbf{g}_k$	j', $\mathbf{g}j'$, $\mathbf{g}s'$, $\gamma_{\mathbf{g}}(\mathbf{N}, \Delta t)$
Job	J_t (also depends on Node)	None
JobFactory	None	None
Connection	None	$\frac{\partial \mathbf{M}_j}{\partial t}$

Table 3.1: Variables and Functions Represented in the Object Model

Notice that there is no need to represent the Δs and $\Delta \mathbf{g}s$ properties from the vector representation. This is because, in the Java implementation, the computer's hardware clock is used to regularly execute the node and parent iterations. A computer program operates in an environment in which certain constructs are externally specified, time being one of them. The mathematical model is an abstraction that does not involve a quantity called "time" unless we carefully define this concept. The computer, however, is a physical machine equipped with a device for measuring actual time. This "external specification" is, perhaps, the difference between math and science.

3.3 So How are we Going to Refer to all This?

The Java implementation of this idea is called Rings. Rings is a commercial product available from Almost Realism. Licensing is through Michael Murray. More information is available in the final chapter, titled "The Product".

```
murraym@oberon.reed.edu$ telnet ball 6767
Trying 134.10.15.90...
Connected to ball.reed.edu (134.10.15.90).
Escape character is '^]'.
Welcome to Rings 0.4 Network Client
[----]>
```

Figure 3.3: The Rings Network Client Terminal: Logging into a Node

3.4 Talking to the Machine: The Terminal

The Java implementation is bundled with a terminal program which allows the user to connect directly to a running Node and modify the software configuration, change the value of variables, and give instructions for processes to execute. This section outlines each command that the terminal accepts and gives the format of the command, a description of what it does, and an example in some cases.

The terminal is accessed by using a telnet client. When a Node is running, simply open a telnet connection to the computer on port 6767. This is usually done with a unix command as shown in figure 3.3.

3.4.1 behave

Form

```
behave <server behavior class>
```

Description

Executes the code contained in the specified server behavior class. For more information on server behaviors, see the generated documentation for the ServerBehavior interface.

3.4.2 close

Form

```
close <id>
```

Description

Closes the connection (NodeProxy) to the peer with index ⟨id⟩ in the peer list.

Example

```
[----]> close 0
Dropped 1 node connections to /171.64.142.95
```

3.4.3 date

Form

```
date
```

Description

Gives the date according to the system clock on the machine running the Server (wrapper for NodeGroup).

Example

```
[----]> date
Sun Aug 13 14:32:21 PDT 2006
```

3.4.4 dbs start

Form

```
dbs start
```

Description

Starts a DBS thread which can handle the JobOutput objects. More information about JobOutput can be found in the section "Producing Output", chapter 5.

Example

```
[----]> dbs start
Started DBS.
```

3.4.5 dbs create

Form

```
dbs create
```

Description

Creates a table in the relational database that can be used to store output. For more information on the table structure, see Appendix E.

3.4.6 dbs add

Form

```
dbs add <output handler class>
```

Description

Adds the specified output handler class as a handler for JobOutput objects. More information about JobOutput can be found in the section “Producing Output”, chapter 5.

Example

```
[----]> dbs add net.sf.j3d.network.tests.UrlProfilingJob$Handler
Added net.sf.j3d.network.tests.UrlProfilingJob$Handler@17e6a96
as a handler for output.
```

3.4.7 giterate

Form

```
giterate
```

Description

Manually executes the parent iteration. This is mostly for tinkering/debugging purposes. If manually invoking the parent iteration is necessary, usually the server configuration needs to be modified.

Example

```
[----]> giterate
Network Node Group: 1 children and 1 server connections.  iteration performed.
```

3.4.8 inputrate

Form

```
inputrate <id>
```

Description

Gives the average number of messages received per minute from a peer with index ⟨id⟩ in the peer list. Theses “messages” may be jobs or just network messages (ping, open connection, send output, etc). This measurement is an average since the last time the inputrate command was executed (for the purpose of creating external monitoring tools).

Example

```
[----]> inputrate 0
0.06732672251870885
```

3.4.9 jobtime

Form

```
jobtime
```

Description

Gives the average time to complete a job by the node(s) on this server. This is analogous to the value of J_t in the mathematical model. The value returned is measured in milliseconds.

Example

```
[----]> jobtime
40689.42069123369
```

3.4.10 open

Form

```
open <host> [port]
```

Description

Opens a connection (NodeProxy) to the specified host. The port is optional, and if left unspecified defaults to 7766.

Example

```
[----]> open 171.64.142.95
Opened host 171.64.142.95
```

3.4.11 peers

Form

```
peers
```

Description

Lists the peers connected to the node(s) on this server.

Example

```
[----]> peers
/134.10.15.90
/134.10.15.93
```

3.4.12 run test

Form

```
run test <time>
```

Description

Sends a test task to the first peer in the peer list for this Sever (wrapper for NodeGroup). A test task produces test jobs that simply instruct the node executing the job to sleep for a predefined number of milliseconds, ⟨time⟩.

3.4.13 sendtask

Form

```
sendtask <id> <task class> [<key1>=<value1> <key2>=<value2> ...]
```

Description

The sendtask command instructs a peer of this node (with index ⟨id⟩ in the peer list) to instantiate a JobFactory object of the ⟨task class⟩ type and add it to the list of JobFactory objects maintained by the NodeGroup of the remote node specified. The set method of the JobFactory that instantiated will be called with the key value pairs that are optionally specified at the end of the command.

Example

```
sendtask 0 net.sf.j3d.network.tests.UrlProfilingTask
dir=http://www.reed.edu/~murraym/images.txt
uri=http://171.64.142.10/
size=200
```

This would instruct the remote node with index 0 in the peer list to construct a UrlProfilingTask. The dir, uri, and size variables will be set using the set method of UrlProfilingTask. Note that when actually entering this into the terminal, it must appear all on one line (line breaks inserted here for typesetting would be replaced with single spaces).

3.4.14 set

form

```
set <key> <value>
```

Description

Sets a variable maintained by the Server, Node or NodeGroup. A list of “keys” for variables are listed below.

Variable From Model	Property ("key")	Name
J_p	nodes.mjp	MinimumJobP
R_p	nodes.relay	RelayP
$\mathbf{g}R_p$	nodes.parp	ParentalRelayP
A_s	nodes.acs	ActivitySleepC
A_o	group.aco	ActivityOffset

Table 3.2: Variables accessible by the set command

Example

```
[----]> set group.aco -0.2
Set group.aco to -0.2
```

3.4.15 status

Form

```
status [file]
```

Description

Prints node status HTML page to the terminal or, optionally, to a file.

3.4.16 tasks

Form

```
tasks
```

Description

Lists the tasks that the nodes maintained by this Server (wrapper for NodeGroup) have worked on. The list also includes tasks that the NodeGroup is producing jobs for (in this case there may be two entries for the same task).

Example

```
[----]> tasks
net.sf.j3d.network.tests.UrlProfilingTask@1a125f0
ImageProfilingTask (1155324825047)
```

3.4.17 threads

Form

```
threads
```

Description

Lists currently running JVM threads that are in the thread group maintained by the Server (wrapper for NodeGroup).

Example

```
[----]> threads
Status Output Thread
Network Server
Resource Server
Server Terminal
HttpCommandServer
NodeProxy for /134.10.15.90
DB Server Thread
7 active threads.
```

3.4.18 uptime

Form

```
uptime
```

Description

Gives the time in minutes since the Server (wrapper for NodeGroup) was initialized.

Example

```
[----]> uptime
Client up for 3005.24695 minutes.
```

Chapter 4

The Lonely Node: Experimental Results for One Server

Here I will present some experimental data collected by running test jobs (see run test command, chapter 3) through one node (one NodeGroup object with one Node object). The test results will show the activity rating of a node over the course of time. Activity rating is a function of j given by the following relation.

$$\text{Activity Rating} = 1 + \frac{j - J_p \cdot J_m}{A_c \cdot J_m} + A_o \tag{4.1}$$

Figure 4.1: Definition of activity rating

4.1 Varying J_p

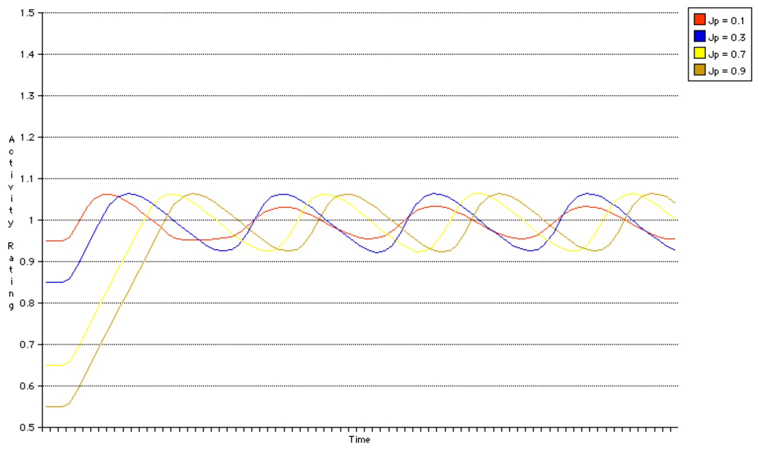

Figure 4.2: Activity rating over time since the initialization of the Server for different values of J_p in each case. Each tick represents one minute of time.

Figure 4.2 shows the effect of altering the value of J_p on a single node's ability to maintain a stable level of jobs in the queue. The values of other parameters for these tests are as follows.

Property	Value
J_m	40 jobs
J_t	10 seconds
R_p	0.0
$\mathbf{g}R_p$	0.0
k	0 other nodes
A_c	2.0
A_s	1.5
A_o	0.0
$\mathbf{g}_k$	1 (the test task)

Table 4.1: Node properties for varying J_p test

Interestingly enough, it can be seen from this graph that the activity rating exhibits a repeating pattern of oscillation with a period of approximately 10 minutes. The oscillation is observed to dampen over longer periods of time, indicating that the recursion relation which defines the change in node state can balance the frequency that the value of j increases with the frequency that the value of j decreases. This does not seem very significant in the case of one node, but the property also holds when other nodes are involved. For a node connected to some members of a complicated graph, the value j varies depending on the peers of the node, as the peers may increase the value of j, and the node in question may also decrease the value of j. Load balancing across the network can be interpreted as an attempt to dampen the oscillation that results from this information exchange. This recursive dampening strategy is reminiscent of solving the **maximum flow problem**.

4.2 Varying A_s

Figure 4.3 shows the effect of altering the value of A_s on a single node's ability to maintain a stable level of jobs in the queue. The values of other parameters for these tests are as follows.

Figure 4.4 shows the same data as figure 4.3, but includes many different values of A_s between 0.3 and 2.0. This data was collected using the same parameters from table 4.2.

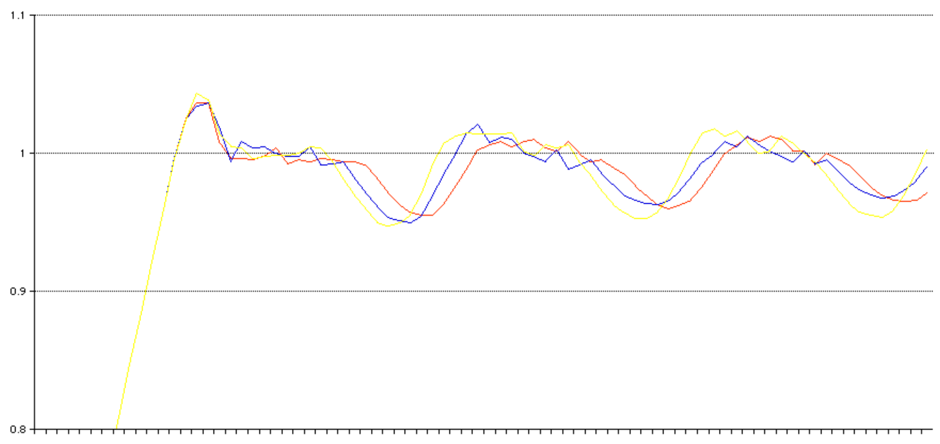

Figure 4.3: Activity rating over time since the initialization of the Server for different values of A_s in each case. Each tick represents one minute of time. Yellow: $A_s = 0.0$, Blue: $A_s = -0.1$, Red: $A_s = -0.3$.

4.3 What Does This Mean?

The experimental data presented above indicates that a node attempts to control the number of jobs it stores by altering the value of s and $\mathbf{g}s$ using the iteration matrix presented in chapter one. In terms of a graph of a connected nodes, stress is created by adding jobs to a node more quickly than it can complete them. This stress is dissipated by altering the values of s and $\mathbf{g}s$ and relaying jobs to other nodes. The relaying of a job from one node to another occurs during the node iteration and the value of s controls the frequency with which the node iteration occurs.

Property	Value
J_m	40 jobs
J_p	0.7
J_t	10 seconds
R_p	0.0
$\mathbf{g}R_p$	0.0
k	0 other nodes
A_c	2.0
A_s	1.5
A_o	0.0
$\mathbf{g}_k$	1 (the test task)

Table 4.2: Node properties for varying J_p test

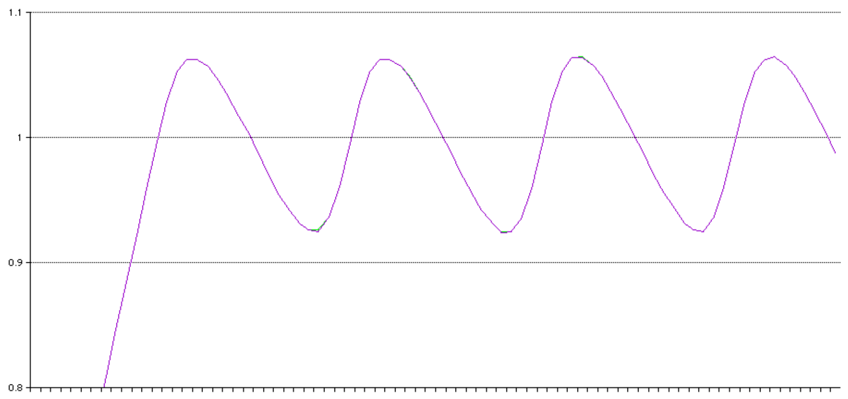

Figure 4.4: Activity rating over time since the initialization of the Server for different values of A_s in each case. Each tick represents one minute of time. The graph includes the data for the following values of A_s: 0.3, 0.5, 0.7, 0.9, 1.0, 1.1, 1.3, 1.5, 1.7, 2.0.

Chapter 5

Using the Machine

5.1 Parallel Processing

When we speak of "grid" or "distributed" computing what we are referring to is the concept of coordinating and sharing computing and data storage resources across dynamic and geographically dispersed organizations. "Grid technologies promise to change the way organizations tackle complex computational problems. However, the vision of large scale resource sharing is not yet a reality in many areas Grid computing is an evolving area of computing, where standards and technology are still being developed to enable this new paradigm." [1]

Parallel processing, or grid computation, is important for a variety of reasons. One of the most notable improvements on current super computing solutions that grid computation can provide is a financial benefit to organizations which require CPU intensive processing that performs in real time or close to it. Efficient load balancing solutions for grid computing are essential for allowing many different software applications to access the computing hardware investment that has been made by their organization. Load balancing is one of the most important techniques allowing users to maximize the utility of their computing environment.

In the case of a centralized grid computing network setup, as described in the preface, load balancing is performed by one computer which maintains communication with every other computer employed for computation. The benefit of this network configuration is that load balancing is relatively easy to implement, but this imparts the cost of maintaining a machine for the purpose of load balancing. A possible solution is the use of a decentralized network configuration, involving each computer communicating with only a few others and autonomous load balancing performed by each computer and its peers.

"Upgrading and purchasing new hardware is a costly proposition, and with the rate of technology obsolescence, it is eventually a losing one. By better utilizing and distributing existing compute resources, Grid computing will help alleviate this problem."[1]

[1] "Grid Computing: The Basics." GRID.ORG. 2004. Accessed from Reed College. 29 Aug. 2006. http://www.grid.org/about/gc/

5.1.1 Other Products and Tools

“Software tools for distributed processing include standard APIs such as MPI and PVM, and open source-based software solutions such as Beowulf and openMosix which facilitate the creation of a sort of ‘virtual supercomputer’ from a collection of ordinary workstations or servers. Technology like ZeroConf (Rendezvous/Bonjour) pave the way for the creation of ad hoc computer clusters. An example of this is the distributed rendering function in Apple’s Shake compositing application. Computers running the Shake software merely need to be in proximity to each other, in networking terms, to automatically discover and use each other’s resources. While no one has yet built an ad hoc computer cluster that rivals even yesteryear’s supercomputers, the line between desktop, or even laptop, and supercomputer is beginning to blur, and is likely to continue to blur as built-in support for parallelism and distributed processing increases in mainstream desktop operating systems. An easy programming language for supercomputers remains an open research topic in Computer Science.”[2]

5.1.2 What Makes Rings Different?

There are three main types of general-purpose super computers. These are:

- Vector processing machines - Allow arithmetic operations to be carried out on many computers simultaneously
- Tightly connected cluster computers - Custom designed computing hardware with the same resources and interface as a normal workstation, but use hardware interconnection between multiple processors which run in parallel, but share the system resources.
- Commodity clusters - A large number of network attached commodity workstations used to execute tasks in parallel

The Rings system is used to facility the necessary communication and distribution of CPU, memory, and hard disk resources to create what is known as a commodity cluster. A commodity cluster is a convenient way to dynamically allocate resources to a multitude of different software projects and applications that require a super computing solution.

Rings may take advantage of networking technologies that permit ad hoc computer clustering. Because a Rings Client only requires one or more peers to be a member of the parallel processing network, any TCP/IP network configuration is supported.

[2] “Supercomputer.” Wikipedia.org. Accessed from Reed College. 29 Aug. 2006. http://en.wikipedia.org/wiki/Supercomputer

5.2 Implementing a Task

Each type of process that is to be performed by the Rings Parallel Processing System must be implemented in the form of a "task".

5.2.1 Tasks and Jobs

A task is implemented by providing a Java class that implements the JobFactory interface. An implementation of the JobFactory interface will require the corresponding implementation of the Job interface. A "Job" represents a small unit of processing work that is independent of other Jobs.

The nextJob method of a JobFactory implementation is invoked during the parent iteration (within the NodeGroup class) for a node. (This is represented by the variable $\mathbf{g}_k$ described in chapter one). The Job produced is then immediately added to the Node maintained by the NodeGroup. The Node is responsible for relaying the Job to other peer nodes or executing the Job by calling the run method for the Job.

5.2.2 The JobFactory Interface

The JobFactory interface requires that the implementation provide a method, nextJob, that generates the next Job that is required to complete the task. The JobFactory must also provide a way of encoding itself in the form of a String so that the task can be sent across the internet. This String has a standard format that is described by the NodeGroup class (see createTask method of the NodeGroup class). The JobFactory must also provide a way of constructing Job objects based on the String that is produced when the Job objects are encoded and sent across the internet. Some other properties must be maintained by a JobFactory implementation and a complete list of methods to be implemented is given in table 5.1.

5.2.3 The Job Interface

The Job interface only requires implementing four methods. The encode method must be implemented so that the Server (or JobFactory, if it is set as the default JobFactory) can reconstruct the task when it is sent across the internet (see the instantiateJobClass method of the Server class). This encoded String must take the same form as the encoded String for a task, a class name followed by a list of key/value pairs separated by colons to be passed to the set method when reconstructing the Job. For an example, see figure 5.2. The getTaskId method must be implemented to tie the Job to the JobFactory instance that produced it. The getTaskString method returns text to be displayed in the list when the tasks command is invoked from the terminal. The set method must be implemented to reconstruct the state of the Job when it is sent across the internet.

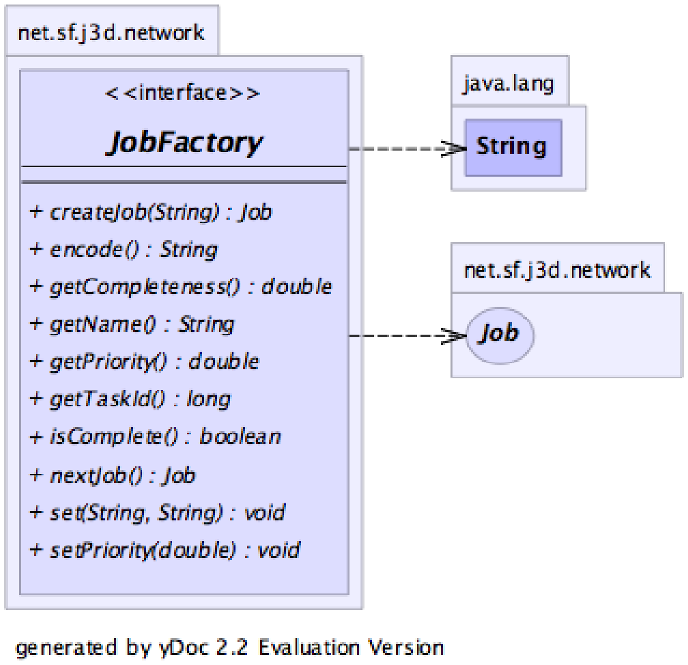

Figure 5.1: The JobFactory interface

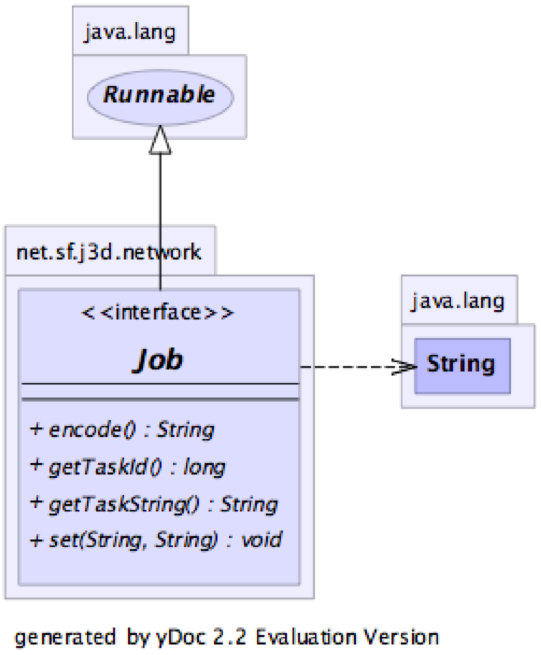

Figure 5.2: The Job interface

```
net.sf.j3d.network.tests.UrlProfilingTask:uri=www.reed.edu:priority=2.0
net.sf.j3d.network.tests.UrlProfilingJob:uri=www.reed.edu:taskid=1155324825047
```

Figure 5.3: Example of an encoded task and an encoded job

5.2.4 An Example: URL Profiling

To illustrate the process of creating a task for the Rings Parallel Processing System we will go through what is required to create a task for profiling a website by repeatedly requesting pages and clocking the time to respond. This task is well suited for the Rings system, because one can construct a network that includes computers from a variety of different physical locations and connection speeds and automatically collect profiling information from a wide variety of possible website users. It seems most logical to start this example by providing the run method for a UrlProfilingJob, as this is the code that is actually executed to perform the test.

UrlProfilingJob.run()

The code for the run method is given in the figure below. This code is relatively simple (and understanding how it works is not central to understanding the Rings system itself). All that is performed here is repeatedly opening a socket stream to a host based on the URL stored by the UrlProfilingJob and timing the response. At the end of this series of requests some statistics are compiled and printed to standard output. What is important to notice is the variables that are referenced, namely this.uri and this.size. These variables are part of the state of the Job implementation and are set using the set(key, value) method. These variables are also included in the String encoded form of the Job so that they will be restored when the Job is transmitted over the internet.

UrlProfilingJob.set(String key, String value)

The set method provided by a Job implementation is used to automatically reconstruct the state of a Job instance when it is transmitted across the internet. Many Java programmers will be familiar with a Serializable or Externalizable class. For more information about why a Job is NOT Serializable or Externalizable and instead uses the set method to restore state, see appendix C. As you can see, the set method allows three different fields of the UrlProfilingJob object to be initialized. These are id, uri, and size. The id field is required by ALL Job implementations, as every Job must provide a getTaskId method that returns the unique ID of the task that produced the Job. Each Job object is responsible for storing this id and returning it when the getTaskId method is called. Every JobFactory object is, in turn, responsible for setting the ID for each Job object it produces. Jobs with apparently non-unique IDs may be eliminated by a ServerBehavior implementation that is used to “clean house”.

The uri and size fields are unique to the UrlProfilingJob class. The uri property is the unique resource identifier of the web page to be profiled by the UrlProfilingJob

```
public void run() throws RuntimeException {
    long start, end, tot = 0, bs = 0;

    for (int i = 0; i < this.size; i++) {
        start = System.currentTimeMillis();

        String d[] = this.uri.split("\\\\");
        StringBuffer b = new StringBuffer();
        for (int j = 0; j < d.length; j++) b.append(d[j]);
        uri = b.toString();

        try {
            InputStream in = new URL(this.uri).openStream();
            while (in.available() > 0) { in.read(); bs++; }
            in.close();
        } catch (MalformedURLException murl) {
            throw new RuntimeException("UrlProfilingJob -- " + murl.getMessage());
        } catch (IOException e) { }

        end = System.currentTimeMillis();
        tot += end - start;

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
    }

    if (this.size <= 0) return;

    long avgTime = tot / this.size;
    long avgBs = bs / this.size;

    StringBuffer b = new StringBuffer();
    b.append(this.uri.substring(this.uri.lastIndexOf("/")));
    b.append(":");
    b.append(this.size);
    b.append(":");
    b.append(avgTime);
    b.append(":");
    b.append(avgBs);
    b.append(":");

    System.out.println(b.toString());
}
```

Figure 5.4: The run method of the UrlProfilingJob class

```
public void set(String key, String value) {
    if (key.equals("id")) {
        this.id = Long.parseLong(value);
    } else if (key.equals("uri")) {
        this.uri = value;
    } else if (key.equals("size")) {
        this.size = Integer.parseInt(value);
    }
}
```

Figure 5.5: The set method of the UrlProfilingJob class

and the size property is the number of requests to make.

UrlProfilingJob.encode()

The encode method of a Job implementation is used to convert the Job to a String representation that can be transmitted across the internet. The String returned by the encode method must take a particular form if the Job is to be reconstructed by the Server. The Server class provides a static method, instantiateJobClass, that constructs and initializes a Job object based on a String representation with a specific format. A JobFactory implementation is required to provide a createJob method that performs this same operation (but the format is left up to the architect of the JobFactory implementation), however it is most often convenient to have the JobFactory implementation simply return the value returned by the static instantiateJobClass method the Server class. If a programmer does decide to create a different way of encoding and decoding jobs, the JobFactory used to decode Jobs must be registered with the currently running Server. This is not scalable, as it is tough to ensure that a JobFactory is registered with every running Server in a network and we are looking to provide a computing solution that is adaptable to many types of tasks. The encode method for UrlProfilingJob produces a String of the form ⟨net.sf.j3d.network.tests.UrlProfilingJob⟩:id=⟨task id⟩:uri=⟨uri⟩:size=⟨size⟩, which is a valid encoded Job String that can be decoded by the instantiateJobClass method of the Server class. Notice that the encode method contains the same fields as those that are initialized by the set method. (This is how the instantiateJobClass method will reconstruct the state of the Job after it is transmitted.)

UrlProfilingJob.getTaskId()

The getTaskId method of a Job implementation returns the unique ID of the task that produced the Job object. This should be set using the set method and encoded using the encode method so that the Job maintains reference to the task that produced it when it is transmitted across the internet.

```
public String encode() {
    StringBuffer b = new StringBuffer();

    b.append(this.getClass().getName());
    b.append(":id=");
    b.append(this.id);
    b.append(":uri=");
    b.append(this.uri);
    b.append(":size=");
    b.append(this.size);

    return b.toString();
}
```

Figure 5.6: The encode method of the UrlProfilingJob class

```
public long getTaskId() { return this.id; }
```

Figure 5.7: The getTaskId method of the UrlProfilingJob class

UrlProfilingJob.getTaskString()

The getTaskString method of a Job implementation should return what would be returned by the toString method of the JobFactory object that produced the Job object.

Constructor for UrlProfilingJob

One important requirement of a Job implementation that has not been mentioned thus far is that a no argument constructor must always be provided. This is essential for the Server.instantiateJobClass method to reconstruct the Job when it is transmitted across the internet. However, in addition to a no argument contructor, UrlProfilingJob has a constructor that is used by UrlProfilingTask.

UrlProfilingTask.nextJob()

The nextJob method of a JobFactory implementation returns the next Job object that is to be executed to complete the task represented by the JobFactory. In the case of UrlProfilingTask, each Job produced is the same (each opens the same URL some number of times and reports the profile results. Ideally the Jobs will end up on many different computers so that the profile effectively represents response time for

```
public String getTaskString() { return "ImageProfilingTask (" + this.id + ")"; }
```

Figure 5.8: The getTaskString method of the UrlProfilingJob class

```
public UrlProfilingJob(long id, String uri, int size) {
    this.id = id;
    this.uri = uri;
    this.size = size;
}
```

Figure 5.9: Constructor for the UrlProfilingJob class

```
public Job nextJob() { return new UrlProfilingJob(this.id, this.uri, this.size); }
```

Figure 5.10: The nextJob method of the UrlProfilingTask class

a variety of different locations.) The Job is constructed using data from the fields of the UrlProfilingTask. These properties of the task are set with the set method (and encoded with the encode method).

UrlProfilingTask.set(String key, String value)

The set method provided by a JobFactory implementation is used to automatically reconstruct the state of a JobFactory instance when it is transmitted across the internet. The functionality is the same as the set method of Job (see above).

UrlProfilingTask.encode()

The encode method provided by a JobFactory implementation is used to convert the JobFactory to a String representation that can be transmitted across the internet. The functionality is the same as the encode method of Job (see above).

UrlProfilingTask.createJob(String data)

The createJob method provided by a JobFactory implementation constructs and initializes a Job object based on the encoded String form of the Job. Most often this method will just make a call to the static method Server.instantiateJobClass.

```
public void set(String key, String value) {
    if (key.equals("id")) {
        this.id = Long.parseLong(value);
    } else if (key.equals("uri")) {
        this.uri = value;
    } else if (key.equals("size") {
        this.size = Integer.parseInt(value);
    }
}
```

Figure 5.11: The set method of the UrlProfilingTask class

```
public String encode() {
    StringBuffer b = new StringBuffer();

    b.append(this.getClass().getName());
    b.append(":id=");
    b.append(this.id);
    b.append(":uri=");
    b.append(this.uri);
    b.append(":size=");
    b.append(this.size);

    return b.toString();
}
```

Figure 5.12: The encode method of the UrlProfilingTask class

```
public Job createJob(String data) { return Server.instantiateJobClass(data); }
```

Figure 5.13: The createJob method of the UrlProfilingTask class

UrlProfilingTask.getTaskId()

The getTaskId method of a JobFactory implementation returns a unique ID for this task. This ID is usually generated by making a call to System.currentTimeMillis() when the task is initialized. The task should also provide support for setting the ID using the set method and should include the ID in the encoded String form of the JobFactory.

UrlProfilingTask.getName()

The getName method of a JobFactory implementation should return a unique name for the task. To make the name unique, the ID of the task is usually included in the name, though this is not required.

Methods of UrlProfilingTask for Priority and Completeness

A JobFactory implementation is responsible for keeping track of a decimal value for the priority of the task (default should be 1.0) and the task "completeness", or how many of the Jobs for the task have been sent as a percentage of the total number of Jobs that need to be completed to complete the task. The getPriority and setPriority methods simply return and set the priority property of the task,

```
public long getTaskId() { return this.id; }
```

Figure 5.14: The getTaskId method of the UrlProfilingTask class

```
public String getName() { return "UrlProfilingTask (" + this.id + ")"; }
```

Figure 5.15: The getName method of the UrlProfilingTask class

```
public void setPriority(double p) { this.pri = p; }
public double getPriority() { return this.pri; }
```

Figure 5.16: The getPriority and setPriority methods of the UrlProfilingTask class

which UrlProfilingTask stores as a private field. The getCompleteness method of

```
public double getCompleteness() { return 0; }
public boolean isComplete() { return false; }
```

Figure 5.17: The getCompleteness and isComplete methods of the UrlProfilingTask class

UrlProfilingTask returns 0.0 always, because UrlProfilingTask is designed to run indefinitely (until stopped). The isComplete method returns false always.

In this section we have described in complete detail how each required method of Job and JobFactory is implemented to create a task that performs URL profiling. This source code is included as part of the Rings API and has been tested with existing websites. Next we will discuss how we can make one very important improvement to this task; something we really can't do without in a large scale network.

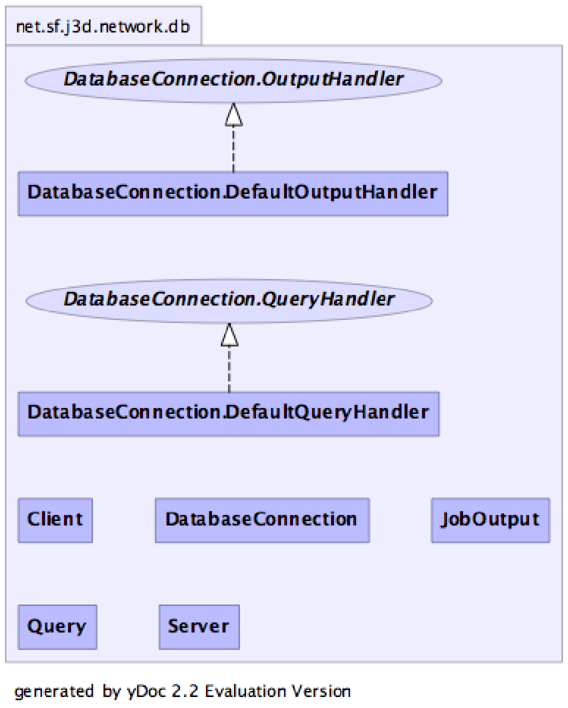
net.sf.j3d.network.db
DatabaseConnection.OutputHandler
DatabaseConnection.DefaultOutputHandler
DatabaseConnection.QueryHandler
DatabaseConnection.DefaultQueryHandler
Client
DatabaseConnection
JobOutput
Query
Server
generated by yDoc 2.2 Evaluation Version

Figure 5.18: Classes for Storing Output

5.3 Producing Output

Next we will discuss how we can make one very important improvement to the UrlProfilingTask just described. A drawback of the UrlProfilingJob class is that the run method prints to standard output. What this means is that each machine that executes a job for this task will have the statistics collected by the job displayed on that machine. This does not seem very logical, as we would most likely want the data that is produced by each job to end up in one place, and often that place is the machine where the jobs originated.

We have considered a few different approaches for collecting and storing whatever "information" may be produced by the execution of a job. The simplest of these methods is presented here, and another is presented in chapter 6. What is outlined here are the Java classes and interfaces provided by the Rings system for the purpose of storing the output produced by a job. There is also a caching system which can be used to persist data during one session of the Rings server, which is separate from the output storage system

5.3.1 The JobOutput Class

The JobOutput class is used to provide a universal way for different Job implementations to encapsulate whatever output needs to be stored by Jobs executing across multiple machines. The JobOutput object provides the getOutput and setOutput

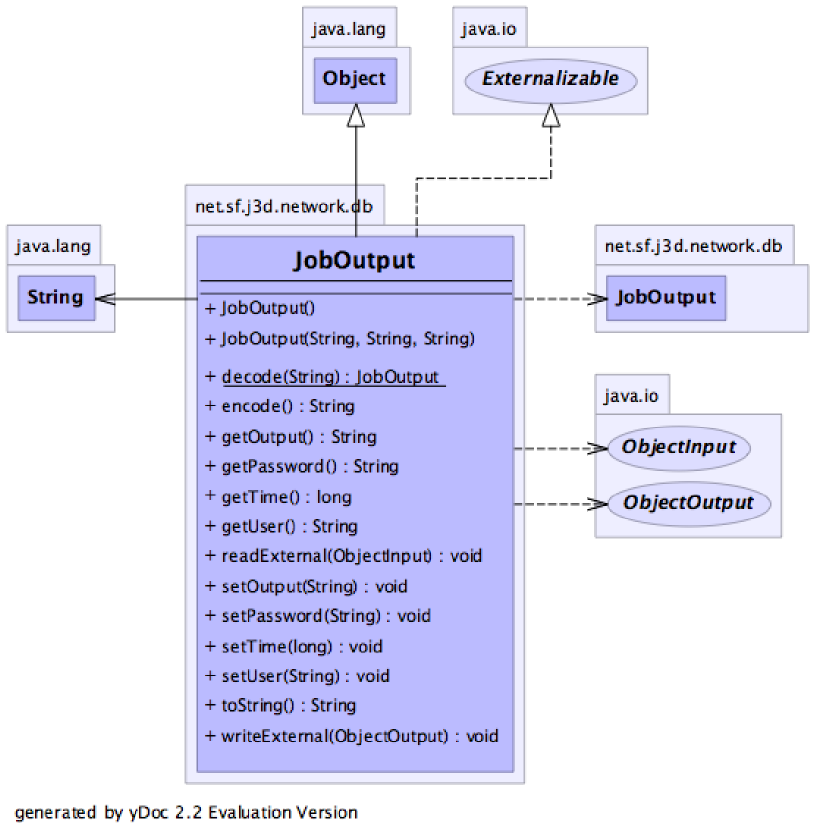

Figure 5.19: The JobOutput class

methods which provide access to a String field of the JobOutput object which can be used to store whatever information a Job instance needs to persist after execution of the Job.

Again, a design choice was made to provide only a String field for persistent output. This is not a requirement because the JobOutput class, as opposed to the Job and JobFactory interfaces, does take advantage of Java serialization through the Externalizable interface (see Appendix C). It is reasonably easy to replace the functionality provided by JobOutput, or to extend the JobOutput class. Classes which extend JobOutput can be used with the other classes provided by the Rings system without hassle, but the implementor is responsible for providing the methods required by the Java Externalizable interface. The JobOutput class has been extended in one case to store image output produced by a ray tracing task. This implementation uses Java collections, which are quite conveniently transmitted by the writeExternal and readExternal methods.

5.3.2 The Client Class

The Client class moderates a connection to an output host where JobOutput objects can be sent and persisted. The Client class provides the writeOutput method for

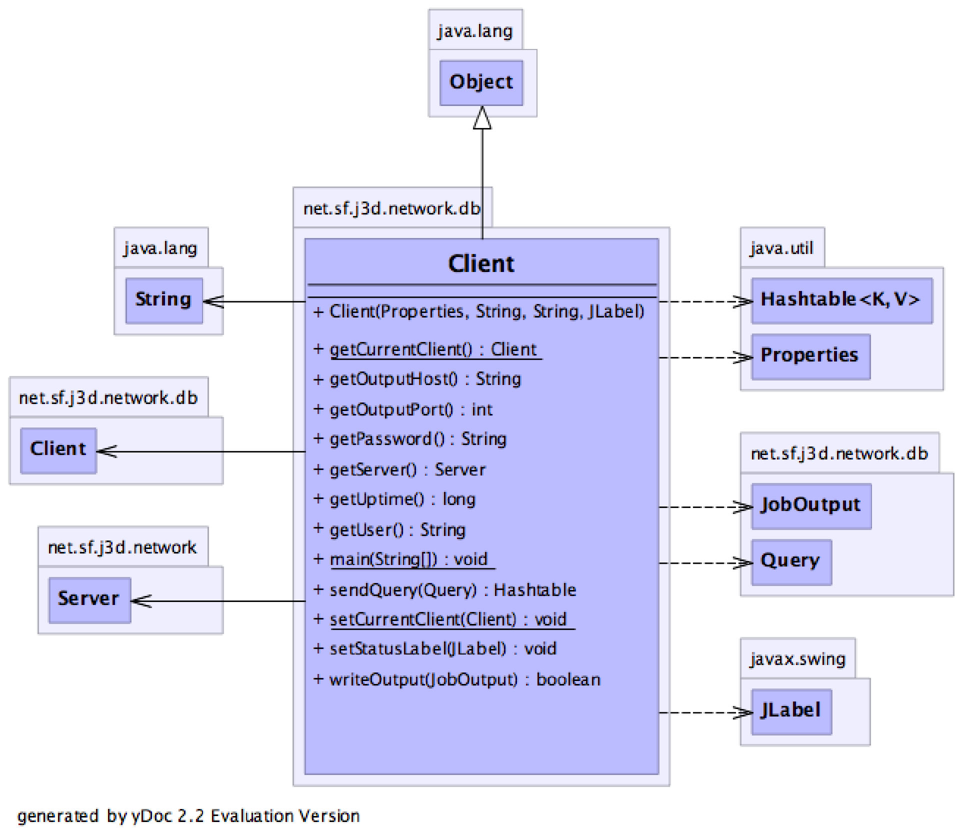

Figure 5.20: The Client class

sending output and the sendQuery method for requesting persisted data.

5.3.3 The OutputHandler Interface

An interface is used in Java to define a general category of object that shares certain properties with all objects in the same category. When an object is in a category defined by a Java interface, we say the class for that object implements the interface. The OutputHandler interface can be implemented by classes that provide a method for persisting JobOutput data or by classes that need to be notified when information arrives to be persisted. The OutputHandler interface specifies the storeOutput method which is notified when a JobOutput object is recieved by the output server.

A DefaultOutputHandler is provided to store persisted data in a relational database using JDBC. This output handler can be initialized in the configuration file for the Rings server. For more information about the table structure required for this output handler, see appendix E.

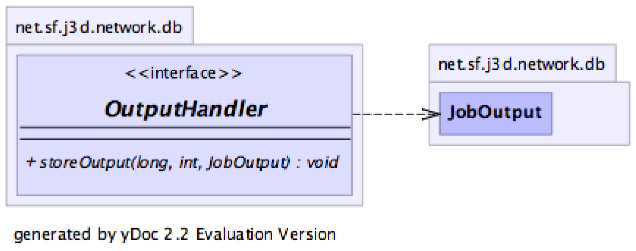
net.sf.j3d.network.db
<<interface>>
OutputHandler
+ storeOutput(long, int, JobOutput) : void
net.sf.j3d.network.db
JobOutput
generated by yDoc 2.2 Evaluation Version

Figure 5.21: The OutputHandler interface

5.3.4 The OutputServer Class

The OutputServer class accepts connections from Client instances running on remote machines and receives encoded JobOutput objects. The thread maintained by the OutputServer class can be started at the time that a Rings client is initialized. For more information on configuring the Rings client, see appendix D. An OutputServer can also be initialized from the terminal and OutputHandlers can be initialized and added once the OutputServer has been initialized. For more information on working with the terminal see the dbs commands described in chapter 3.

5.3.5 Revising the Example

Now we will update the UrlProfilingJob class to send the statistics it produces using a running Rings client. We will also create a class for handling the output produced by the UrlProfilingJobs that execute across many computers in a Rings network.

The new UrlProfilingJob.run()

At the very end of execution, the run method in our previous example printed some statistics to standard output. We will replace this print statement with usage of the Client and JobOutput classes. The revised source code is shown in figure 5.23 (Some of the code has been omitted to keep this figure compact).

An OutputHandler for UrlProfilingJob

The DefaultOutputHandler can be used to store the information reported by UrlProfilingJob and this information can be extracted from the relational database at a later point when another application can be used to aggregate the data and display it in some convenient fashion. However, for the purpose of demonstration, we will implement a custom OutputHandler for the UrlProfilingJob output. This output handler will open an output stream to save the profiling data to a tab delimited file, from which it could be imported into a spread sheet program such as included with OpenOffice. The source code for this OutputHandler is shown in figure 5.24.

There are a few important things to notice about the the UrlProfilingOutputHandler class. First of all, a no argument constructor is provided. This is

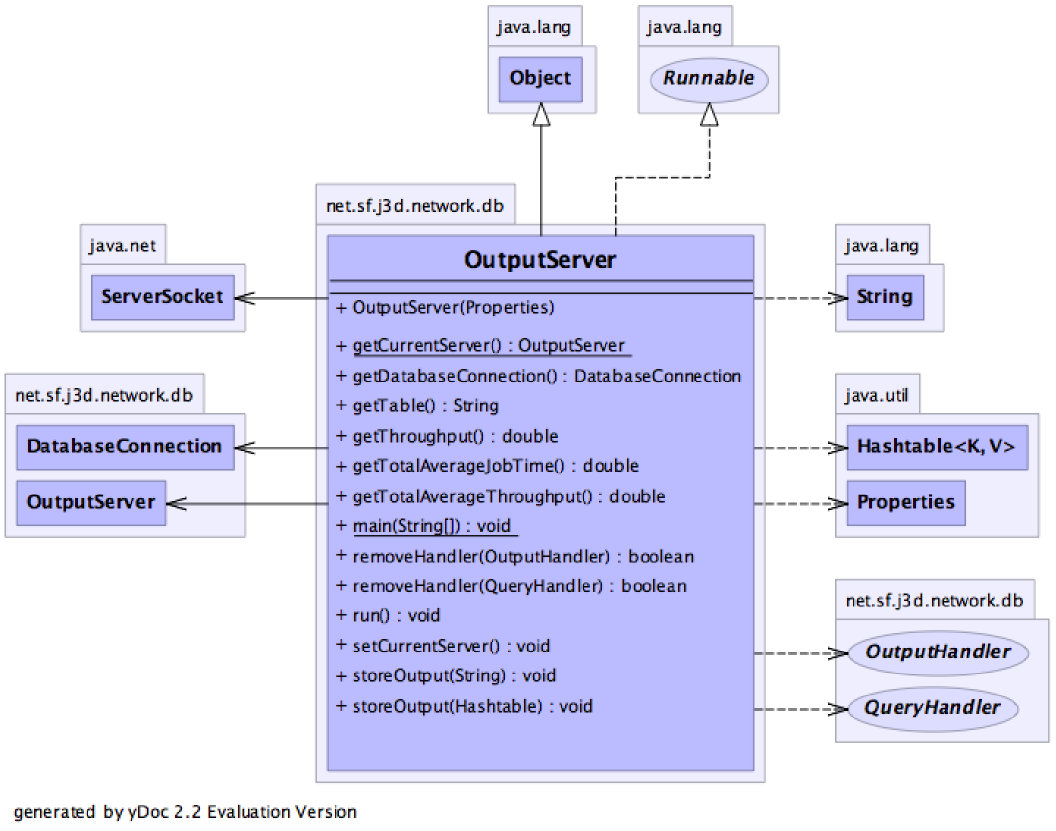
java.lang
Object
java.lang
Runnable
net.sf.j3d.network.db
OutputServer
+ OutputServer(Properties)
+ getCurrentServer() : OutputServer
+ getDatabaseConnection() : DatabaseConnection
+ getTable() : String
+ getThroughput() : double
+ getTotalAverageJobTime() : double
+ getTotalAverageThroughput() : double
+ main(String[]) : void
+ removeHandler(OutputHandler) : boolean
+ removeHandler(QueryHandler) : boolean
+ run() : void
+ setCurrentServer() : void
+ storeOutput(String) : void
+ storeOutput(Hashtable) : void
java.net
ServerSocket
net.sf.j3d.network.db
DatabaseConnection
OutputServer
java.lang
String
java.util
Hashtable<K, V>
Properties
net.sf.j3d.network.db
OutputHandler
QueryHandler
generated by yDoc 2.2 Evaluation Version

Figure 5.22: The OutputServer class

```
public void run() throws RuntimeException {
    long start, end, tot = 0, bs = 0;

    ...
    ...
    ...
    ...

    StringBuffer b = new StringBuffer();
    b.append(this.uri.substring(this.uri.lastIndexOf("/")));
    b.append(":");
    b.append(this.size);
    b.append(":");
    b.append(avgTime);
    b.append(":");
    b.append(avgBs);
    b.append(":");

    System.out.println(b.toString());
}
```

Figure 5.23: The run method of the UrlProfilingJob class

required for the output handler to be initialized at run time from the Rings terminal. Also access to the print stream is contained in a synchronized block. This is necessary to prevent concurrent threads from attempting to write to the file. Any code contained in the sotreOutput method of an OutputHandler must be thread safe, as multiple threads may be configured to use the same output handler, not to mention that separate threads are used for simultaneous client connections.

5.4 Putting it all Together

In the previous section we made minor modifications to the UrlProfilingJob class and created a UrlProfilingOutputHandler. Now we are ready to test the UrlProfilingTask by starting the Rings client and sending the task to a remote machine. For more information on configuring and initializing a Rings client, see appendix D.

5.4.1 Running a Task

To send the UrlProfilingTask we must initialize two Rings clients. These two client can be on the same machine, or on separate machines. Figure 5.25 shows the process of (1) logging in to one client, (2) opening a connection to the other client, (3) listing the peers of the local client, (4) starting the output server, (5) initializing a UrlProfilingOutputHandler, and (6) sending a new UrlProfilingTask to the remote client.

```
public class UrlProflingOutputHandler implements OutputHandler {
    PrintWriter out;

    public UrlProfilingOutputHandler() throws IOException {
        this.out = new PrintWriter(
        new BufferedWriter(new FileWriter("url-profile" +
        System.currentTimeMillis() + ".txt")));
    }

    public void storeOutput(long time, int uid, JobOutput output) {
        String s[] = output.getOutput().split(":");

        StringBuffer b = new StringBuffer();
        b.append(time);
        b.append("\t");

        for (int i = 0; i < s.length; i++) {
            b.append(s[i]);
            b.append("\t");
        }

        String bs = b.toString();
        synchronized (this.out) { this.out.println(bs); this.out.flush(); }
    }
}
```

Figure 5.24: An OutputHandler implementation for UrlProfilingJob

Figure 5.26 shows the line that must be present in the Rings client configuration file for each client so that the output produced by each UrlProfilingJob is sent to the proper server. For more information about the Rings configuration file, see appendix D.

Once this is done, the UrlProfilingTask should begin producing jobs and the jobs should be distributed and executed. Data will then be written out to a text file on the host where the output server is running.

5.4.2 Viewing Node Status

While the Rings client is running, it produces an HTML file that contains information about the current status. For more information about configuring where this HTML file resides, see appendix D. An example of this status page is shown in figure 5.27.

```
murraym@oberon.reed.edu$ telnet ball 6767
Trying 134.10.15.90...
Connected to ball.reed.edu (134.10.15.90).
Escape character is '^]'.
Welcome to Rings 0.4 Network Client
[----]> open ring
Opened host ring
[----]> peers
ring/134.10.15.94
[----]> dbs start
Started DBS.
[----]> dbs add net.sf.j3d.network.tests.UrlProfilingOutputHandler
Added net.sf.j3d.network.tests.UrlProfilingOutputHandler@17e6a96 as a handler for outpu
[----]> sendtask 0 net.sf.j3d.network.tests.UrlProfilingTask uri=http://www.reed.edu si
Started sendtask thread: 1154986910663@ring/134.10.15.94
```

Figure 5.25: (1) logging in to one client, (2) opening a connection to the other client, (3) Listing the peers of the local client, (4) starting the output server, (5) initializing a UrlProfilingOutputHandler, and (6) sending a new UrlProfilingTask to the remote client.

```
# Connection info for output host.
servers.output.host=ball.reed.edu
```

Figure 5.26: Configuring the clients to report output

Figure 5.27: Automatically generated status page

Method	Description
createJob(String) returns Job	Converts the specified String representation of a Job into a Job object for this task.
encode() returns String	Converts this task into a String representation that can be transmitted across the internet. This encoding takes the form "⟨class name⟩ : ⟨key1⟩ = ⟨value1⟩ : ⟨key2⟩ = ⟨value2⟩ [...]" Where ⟨class name⟩ is the full Java class name for the JobFactory implementation, and ⟨key1⟩ = ⟨value1⟩ is a key/value pair to be passed to the set method of the JobFactory implementation to reconstruct the state of the object on the remote machine. The [...] denotes that as many key/value pairs can be specified as are required to restore the state of the object. A backslash (\) may be used to escape the colon (:) character. The colon character is automatically ignored if it comes before two forward slashes, in the case of a url. For an example, see figure 5.2.
getCompleteness() returns double	Returns a value between 0.0 and 1.0 that corresponds to the percentage of Jobs for this task that have been sent out already (requested through the nextJob method).
getName() returns String	Returns a name for this task. This is the name that will show up in the task list when the tasks command is invoked on the terminal.
getPriority() returns double	Returns the priority of this task. The priority controls how many jobs will be requested from this task each iteration. If priority is 1.0, one job will be requested each parent iteration.
getTaskId() returns long	Returns a unique ID for this task. This is usually generated by calling System.currentTimeMillis().
isComplete() returns boolean	Returns true if this task has no more Jobs to send out.
nextJob() returns Job	Returns the next Job to completed for this task.
set(String key, String value)	Sets a property of this task. The set method is called by NodeGroup when reconstructing the task from its encoded String form.
setPriority(double)	Sets the priority of this task.

Table 5.1: Methods required by the JobFactory interface

Chapter 6

Extending the Machine

The Rings Server is a very extendable package. Most modules are nicely encapsulated and replaceable with custom solutions. Also, Rings provides an API for caching frequently used data on a local machine and this API. Other APIs of this sort can be developed completely separately from Rings and used by executing Jobs at runtime. The ServerBehavior interface allows modules to be developed that can be accessed using the terminal command *behave* (see chapter 3). One use of a ServerBehavior is to initialize other threads that may be required for custom Job implementations.

A ServerBehavior can be performed using the terminal or upon instruction from a remote Rings client instance. This means that a user can execute a ServerBehavior that propagates through the entire network and reconfigures each client as the message cascades across the node graph.

6.1 Decentralized Data Storage

One extension of the Rings System that has been scoped for design and implementation during 2006 and early 2007 is an extension of the output server system to provide a mechanism for decentralized data storage. What this means is that a small amount of disk space would be reserved on each machine running the Rings client and this disk space would be used to store information shared across the network. The decentralized data storage solution would provide a safe way to store large quantities of information, as the data can be spread out across as many machines as needed and each chunk of data would be stored in a few different locations to prevent data loss in case of a failure.

The implementation of this data storage system involves creating a network file system. The index (directory/file structure) of the filesystem is known by all nodes on the network, but the data is stored whereever is convenient based on popularity and network traffic time.

The current plan for decentralized data storage involves a two tiered system as summarized below.

- The ResourceDistributionTask class maintains the index of all files contained in the distributed file system. The ResourceDistributionTask is responsible for notifying the node peers of new files and files that have been removed. Also, the ResourceDistributionTask maintains a collection of Job instances that are cycled into the queue of a running Rings client as necessary for moving local data to peer machines. These Job instances (ResourceDistributionJob) are designed to select the least recently/frequently accessed pieces of data and move them to peer computers. The remote machine persists the data using whatever storage method is configured for the Rings client. This data can be retrieved by a ResourceDistributionTask executing on the remote machine. The storage method could be a relational database using DefaultOutputHandler, a collection of data files in a local directory or any other way of storing information on the local machine.

- The DistributedResource class represents a file on the network file system. Many DistributedResource instances are referenced by a ResourceDistributionTask object which maintains the directory structure of the network file system. The data maintained by a DistributedResource instance can be requested from the Resource Server provided as part of the main Rings Server (for more information on initializing the Resource Server, see appendix D). When this request occurs, a local instance of ResourceDistributionTask is queried for local data that pertains to the request. Then that request is cascaded to the peers of the local client to fill in the other required information. Finally, when the request has propagated far enough to be completely populated with the proper data, the request is returned with the data. Once data is requested and populated in this manner, it is stored in memory briefly. This cached data then "competes" in a popularity contest with data stored on the local disk by the ResourceDistributionTask. The in memory cache may be committed to the disk cache (local OutputServer) if space and popularity permit. Once in the disk cache, the data will be broken into uniform size chunks and distributed among peers as described above.

- A plugin for the Slide content repository (`http://jakarta.apache.org/slide/`) will be provided as a front end to the ResourceDistributionTask class. When a Rings client is running on the local machine, webdav can be used to mount a network drive that will present the distributed file system as network attached storage. This plugin will make modifications to the file system by notifying the currently running ResourceDistributionTask.

6.2 Ideas for Extending the Engine

The future of the Rings Parallel Processing system may hold a great many things and at the moment we are in the midst of choosing what direction is most important to take the product so as to provide for the needs of the research and commercial communities that require grid computing. We are also taking into consideration the needs of an online community that may be used to empower graphic artists and independent researchers with the ability to easily develop a connected network of peers willing to donate unused processing power and disk space to meet the needs of personal grid computing on a tight budget. Contained here are some of the ideas that have been brainstormed for the distant future (2008 and onward) of this project.

6.2.1 A Parallel Operating System

The Rings Server terminal provides some of the functionality expected from a shell to an operating system such as Unix, Linux, or Solaris. One thing that has been proposed is extending the provided terminal to include the ability to run executable files stored on the local machine. A special runtime environment could be used to allow multiple threads of a normal unix executable to run in parallel on separate computers connected as peers in the Rings network. With extensive further development, we see the possibility of distributing the Rings system along with a compact operating system (perhaps some version of Solaris) and marketing the product as a parallel operating system, though this is much further in the future.

6.2.2 A Decentralized Social Networking System

In chapter 5 it was mentioned that the Rings Server automatically produces an HTML status page containing information about the current state of the running application. In the future we plan to hire a web designer to create a set of JSP pages that can be used to display this information instead. The JSP pages can be populated with information directly from the application and will also have access to the core functionality provided by the terminal. This will provide a very convenient interface for the application that is accessible from anywhere.

Taking this a step further, it has been proposed that Rings be set up to provide a way for the user to configure a personal information page that would be similar to those found on the social networking site `facebook.com`. This website is popular among college students and is used to keep in touch with friends from all over the world. One draw back of social networking sites, however, is that the information resides all in one place. If each individual were to use their own computer to host their own pages many of the restrictions imposed by the website administrators would be liberated and members of the site would have the freedom to set up their pages how ever they wish.

With the Rings a task could be programmed to index all of the information provided by the users of the social networking system. This indexing process would take place on the computers running the application, meaning that the social networking system would be truely democratic: owned and operated by the users of the system. No advertisers would be able to buy space on this network (and if they did, it would be directly from the users) and no restrictions on the content would be applied except those that were universally agreed upon by a large majority of the users (restrictions would come in the form of Rings tasks that move around the network and enforce the rules. Voting would be done by submitting tasks to enforce rules.)

Another feature of this system is that the content repository described in section 6.1 would be accessible by all users, meaning the social networking could take on a component of file sharing and backup as well. This is something that simply cannot be offered by current social networking websites. Also, members would be able to take advantage of the unused processing power of their peers in the network simply by using the JSP forms provided to send tasks (such as digital image synthesis) to their peers. This could be a most productive and cost effective tool for graphic artists.

6.3 Ideas for Parallel Processing Tasks

Chapter 7

The Product

The Rings Server and the classes described in this document make up the Rings Parallel Processing system. The Rings system is proprietary software, though we do have plans for a free single user license available to the non commercial user. However, this package will not stand alone, but only operate as part of an public file sharing and parallel processing network.

7.1 Plans for the Future

Our efforts during the end of 2006 and through the early part of 2007 will be focused on implementing the decentralized content repository described in chapter 5. During the later part of 2007 the software will be tested with personal computers on the Reed College campus, where we will set up an online community of parallel processing nodes. This will allow us to test the scalability of the Rings Parallel Processing system and also support the Rings Photon Field Simulation project. We will use the content repository to create a file sharing network for the Reed community as a public service and as a way to encourage the use of our product.

At the end of 2007, when the product has been tested by the community at Reed, we will begin soliciting the attention of a venture capitalist to allow us to fund further testing of the product and to market to research facilities where grid computing is currently necessary.

In parallel with our work to start a company and market the tool as a stand alone system, we will work to create an internet wide online community for parallel processing. This would be an attractive tool to digital artists, who would have the ability to take advantage of unused processing power of their peers in the online community. The media that is produced by the system would be conveniently stored and backed up with the provided content repository. This is also a marketing tool and a means for testing the Rings system on a large scale.

7.2 Licensing the Machine

Appendix A

Glossary

- Iteration Matrix - A matrix which describes the change in state of a node and a node parent due to the node and parent iterations. A matrix product, given in equation 1.4.3, involving the iteration matrix is used to calculate the new state of a node and its parent from the current state, described by a 6D vector quantity.

- Matrix - In mathematics, a matrix (plural matrices) is a rectangular table of numbers or, more generally, a table consisting of abstract quantities that can be added and multiplied. Matrices are used to describe linear equations, keep track of the coefficients of linear transformations and to record data that depend on two parameters. Matrices can be added, multiplied, and decomposed in various ways, marking them as a key concept in linear algebra and matrix theory.

- The Maximum Flow Problem - Finding a feasible flow through a single-source, single-sink flow network that is maximum? . Sometimes it is defined as finding the value of such a flow. The maximum flow problem can be seen as special case of more complex network flow problems. For example, it is the multi-commodity flow problem with only one commodity, and it is the minimum-cost flow problem with all costs set to zero except for an additional arc from the sink to the source, which has a cost of negative one and no capacity. The maximum s-t flow in a network is equal to the minimum cardinality s-t cut in the network, as stated in the Max-flow min-cut theorem.

- Recursion Relation - In mathematics, a recurrence relation, is an equation which defines a sequence recursively: each term of the sequence is defined as a function of the preceding terms. A difference equation is a specific type of recurrence relation. Some simply defined recurrence relations can have very complex (chaotic) behaviours and are sometimes studied by physicists and mathematicians in a field of mathematics known as nonlinear analysis.

- Stochastic - A stochastic process is one whose behavior is non-deterministic in that the next state of the environment is partially but not fully determined by the previous state of the environment.

- Vector Space - In mathematics, a vector space (or linear space) is a collection of objects (called vectors) that, informally speaking, may be scaled and added. More formally, a vector space is a set on which two operations, called (vector) addition and (scalar) multiplication, are defined and satisfy certain natural axioms which are listed below. Vector spaces are the basic objects of study in linear algebra, and are used throughout mathematics, the sciences, and engineering.

Appendix B

Frequencies and Rates

Often in the paper I use the terms “frequency” and “rate” with a lackadaisical omission of specificity. I would like to clarify the use of these terms here.

“Frequency” is a quantized (discrete) measure of some integer number over a positive, finite measure of time. This implies that a measure of frequency is always reducible to the ratio of an integer to a real number (of course, this is true for all real numbers, as all real numbers have an inverse that is also real).

$$\text{Frequency} = \frac{\{y \mid y \in \mathbb{Z}^+\}}{\{x \mid x \in \mathbb{R}^+\}} \tag{B.1}$$

A similar measure, know as “rate”, is also measured over a positive, finite interval of time, but the numerator can take on any real value.

$$\text{Rate} = \frac{\{y \mid y \in \mathbb{R}\}}{\{x \mid x \in \mathbb{R}^+\}} \tag{B.2}$$

The difference really exists in the quantity that is to be measured. If the quantity comes in integer values only (quantized) it is referred to as a frequency. If a quantity can come in any amount, including fractional, irrational, and possibly negative quantities, it is referred to as a rate. For example, “7 bullets fired from a gun each second” is a frequency, while “3.5 liters of water from a hose each second” is a rate.

The point of this hairsplitting discussion is this: the methods of differential and integral Calculus are used to describe the relationships between the rate of change of a variable quantity to other rates of change or other variables. The assumption of the Calculus is that these quantities are continuously defined and can take on any real value. The Calculus is used, however, to make accurate predictions about systems that are, in reality, quantized. An amount of water, for example, was earlier mentioned as a quantity that is not quantized and is adequately measured using a rate. However, water is actually composed of many tiny molecules and the so called “rate” of water coming from a hose is actually a frequency – a discrete number of water molecules per unit time. “3.5 liters of water coming from a hose each second” is an approximation to an actual integer number of water molecules that came from the hose during one second. The Calculus works very well to describe the relationship between a frequency and other variable quantities, as long as that frequency can be adequately approximated by a rate.

Appendix C

Why Avoid Java Serializable and Externalizable?

Many Java programmers will be familiar with the concept of a Serializable or Externalizable class and the question naturally arises: Why does the Rings system seem to avoid usage of these interfaces provided by Java? When a Job or JobFactory object is transmitted across the internet, the state of the object is restored using the set method with key/value pairs that are of the String data type. The set method required by these interfaces is an alternative to the methods of restoring the state of an object provided by Java. The reason for using the set method instead of just requiring that a Job implementation be Serializable/Externalizable is because we wish to require that all properties of a Job have some String representation.

It is perfectly appropriate for a Job or JobFactory implementation to use some unintelligible method of encoding information in the form of a String, but the String requirement allows us to create columns in a relational database that represent the properties of any Job or JobFactory implementation. Also, if the properties of a Job or JobFactory are written directly to an output stream onto the file system, the resulting file will most likely be human readable, which is another convenient feature of requiring String fields only. From a design perspective, we believe that it is a worthy trade off to add some extra processing time for the conversion between a String and other data types stored by a Job because this places the responsibility for data integrity and validation on the Job instead of on the relational database or the Rings system itself.

Again, this is something that was given much thought and could perhaps be changed in the future if performance requires it. For example, we considered representing the properties of a Job with integer or long data types for key/value pairs. However, this proved to be too much of a hassle for the development of new Job implementations, as the Job implementations we have written so far seem to most often use fields that are of the String type. Also, this creates a pretty strict requirement on the size of the key/value data (32 or 64 bits). Overall, I believe the decision that was made is the right one.

Appendix D

Configuring and Running the Rings Client

Appendix E

Relational Database Tables for DefaultOutputHandler