THE UNIVERSITY OF
MELBOURNE

Department of
Computer Science and Software Engineering

Assignment 3 Report

for

COMP90024 (433678): Cluster and Grid Computing

Version: 1.0
June 4, 2012

Copyright notice

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria
AUSTRALIA
3010

ICT Building
111 Barry Street
Carlton

Tel: +613 8344 1300
Fax: +613 9348 1184
`http://www.csse.unimelb.edu.au/`

Version: 1.0
June 4, 2012.

Credits

This document was written by Terence Siganakis (134860), Scott Ritchie (330975), Andrew Vadnal (326558), and Adam Whiteside (327705).

CONTENTS

ONE

Introduction

This report outlines the architecture, design and implementation details of The Grid, a distributed grid computing software package. The Grid allows for the distribution and execution of self-contained executable files across a heterogeneous grid, allowing users of the software to optimise the running of their jobs based on their deadline and any cost restraints. This software package includes a number of different scheduling algorithms that can be used depending on the type of jobs The Grid will be used for.

TWO

Architecture

The architecture of The Grid involves 3 key components: The Master, The Client and The Node.

2.1 The Client

The Client is what the user interacts with to execute jobs on The Grid. The Client sends an executables and input files to The Master, and then also requests and downloads the output files from The Master when the job has completed. The Client can be installed in any location as it connects to The Master remotely. The Client can be reimplemented in any language as long as it follows the expected API to communicate with The Master and authenticates with a trusted username and password.

2.2 The Node

The Node is the workhorse of The Grid. Many computation nodes are setup with The Node, each of which connects to The Master. After specifying the number of cores available for use, and the cost to use the node, each instance is then ready to accept jobs delegated to it by The Master. The Node is responsible for the execution of delegated jobs, then informing The Master as jobs are finished, and sending the output and error files of the job back to The Master.

2.3 The Master

The Master is the central component of The Grid. All communication between The Client and The Nodes goes through The Master. The Master accepts job requests from The Client and then adds them to an internal queue. It then determines which jobs should be run and on which node in The Grid. It keeps track of which jobs are on which nodes and what the status of each job is.

THREE

Design

3.1 Class Diagram

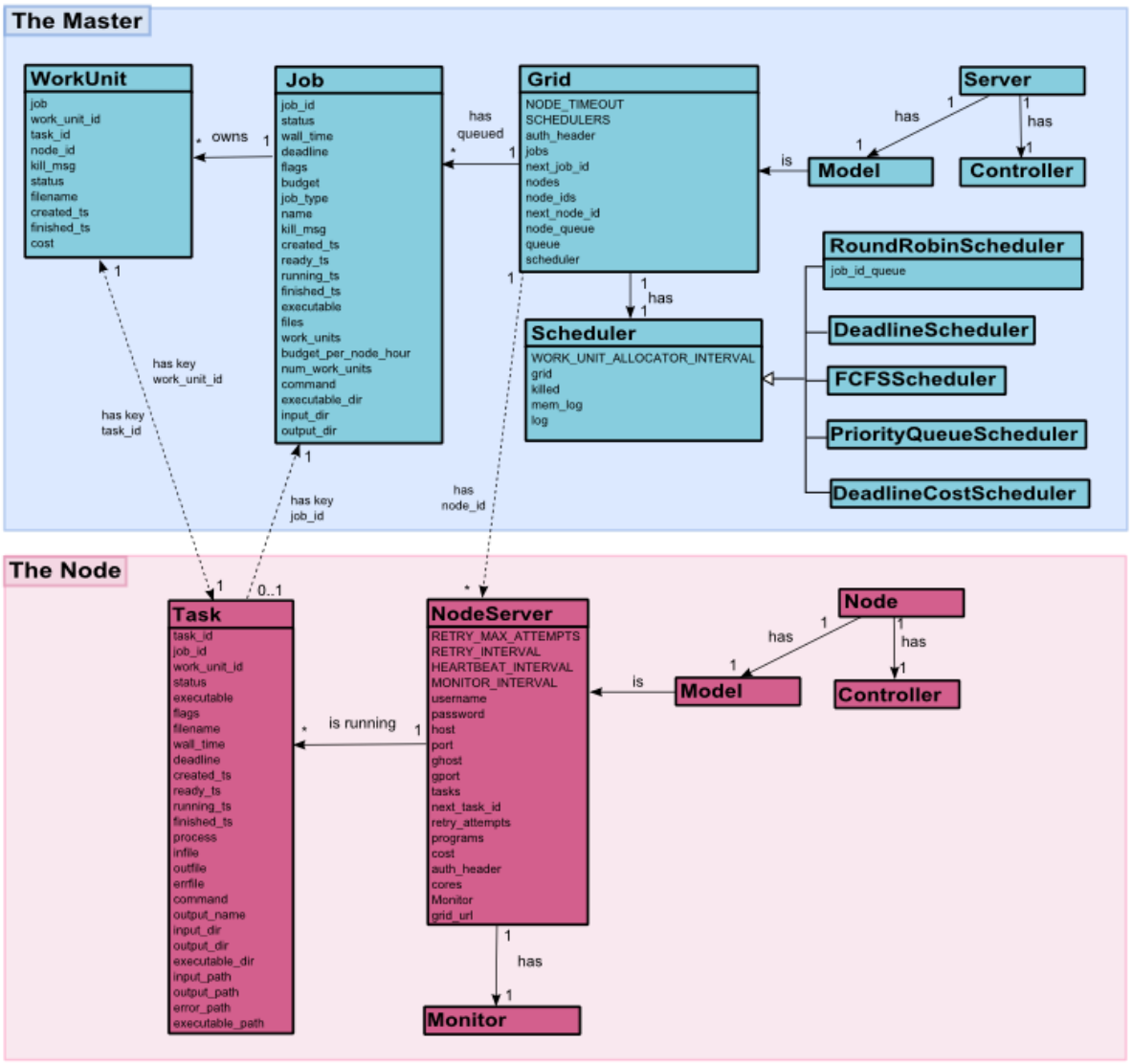

Figure 3.1: Class Diagram

3.2 Sequence Diagrams

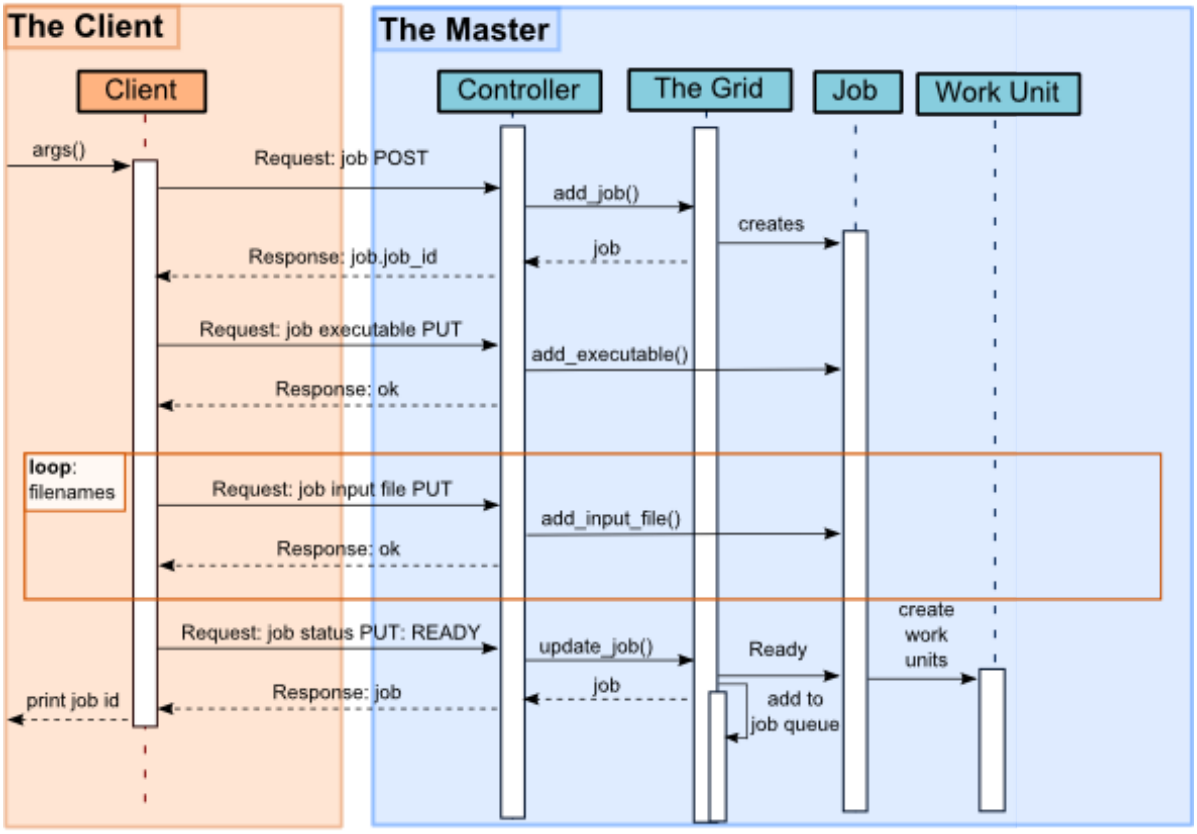

Figure 3.2: New Job Creation

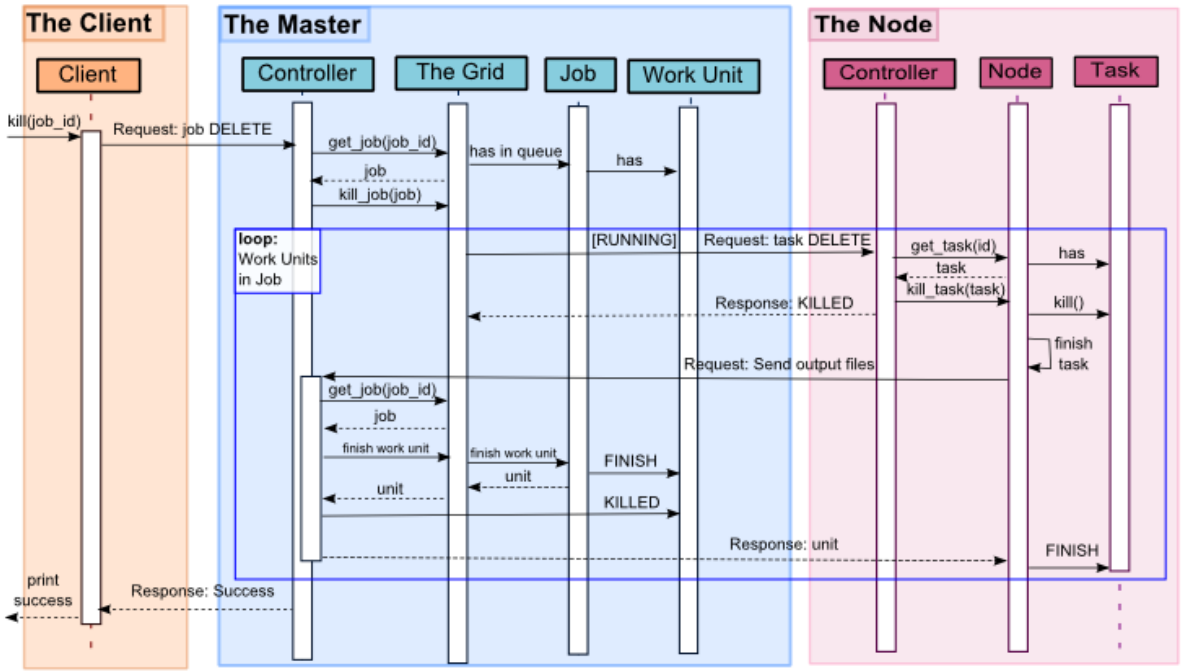

Figure 3.3: Kill a Job

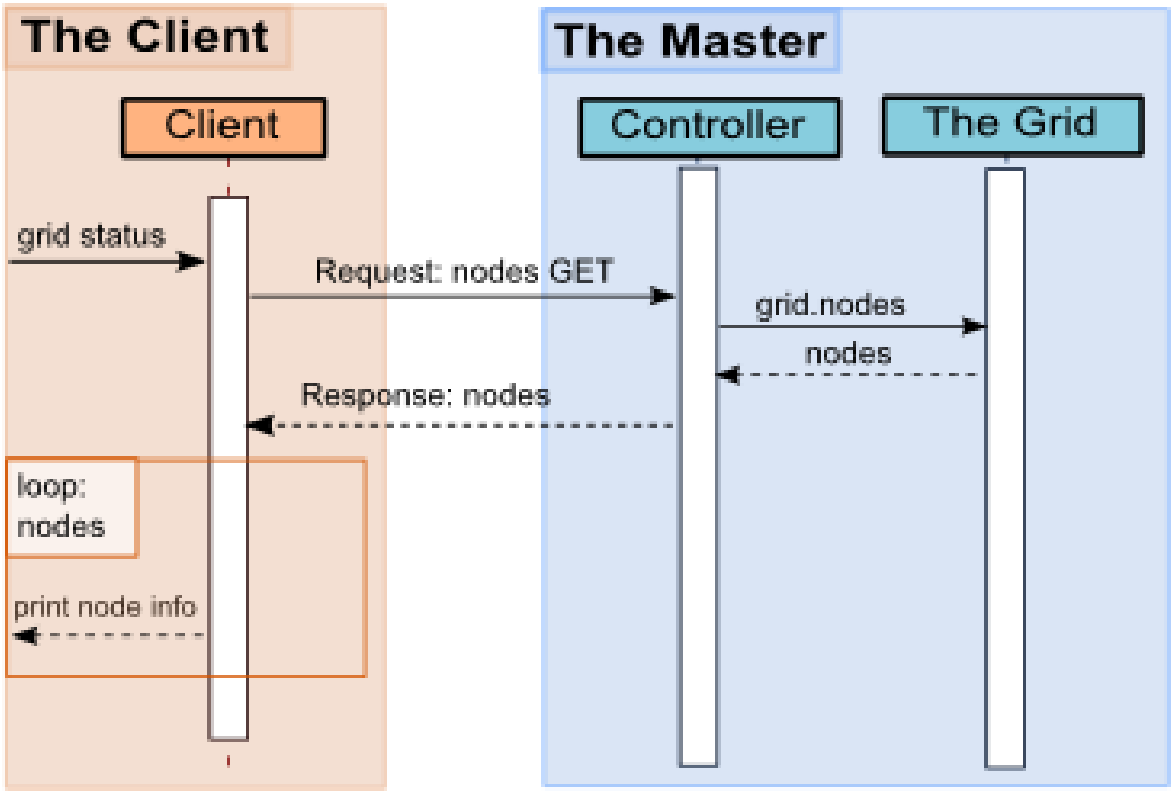

Figure 3.4: Get the status of The Grid

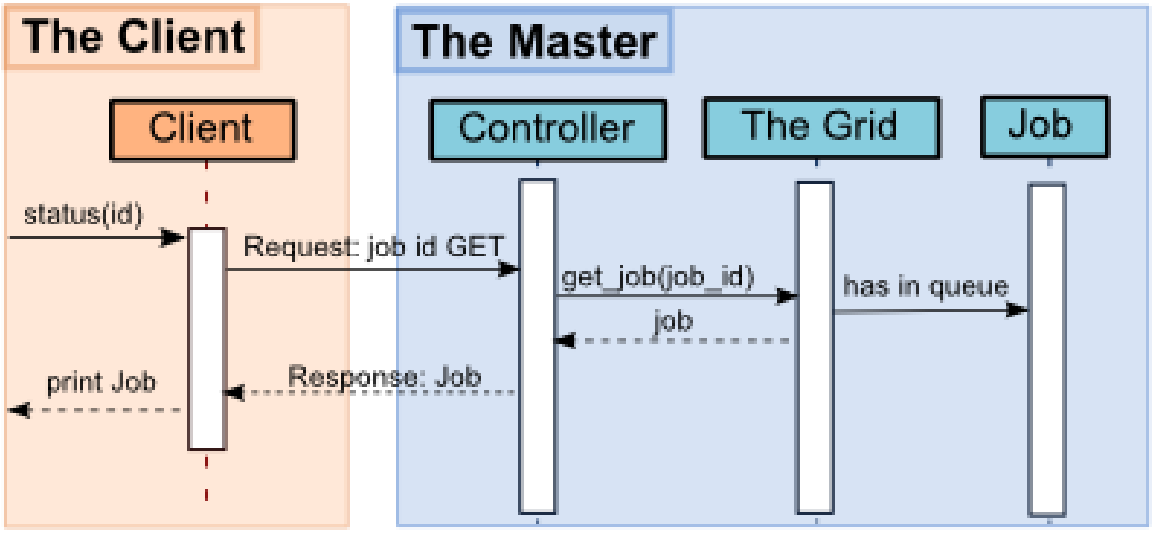

Figure 3.5: Get the status of a Job

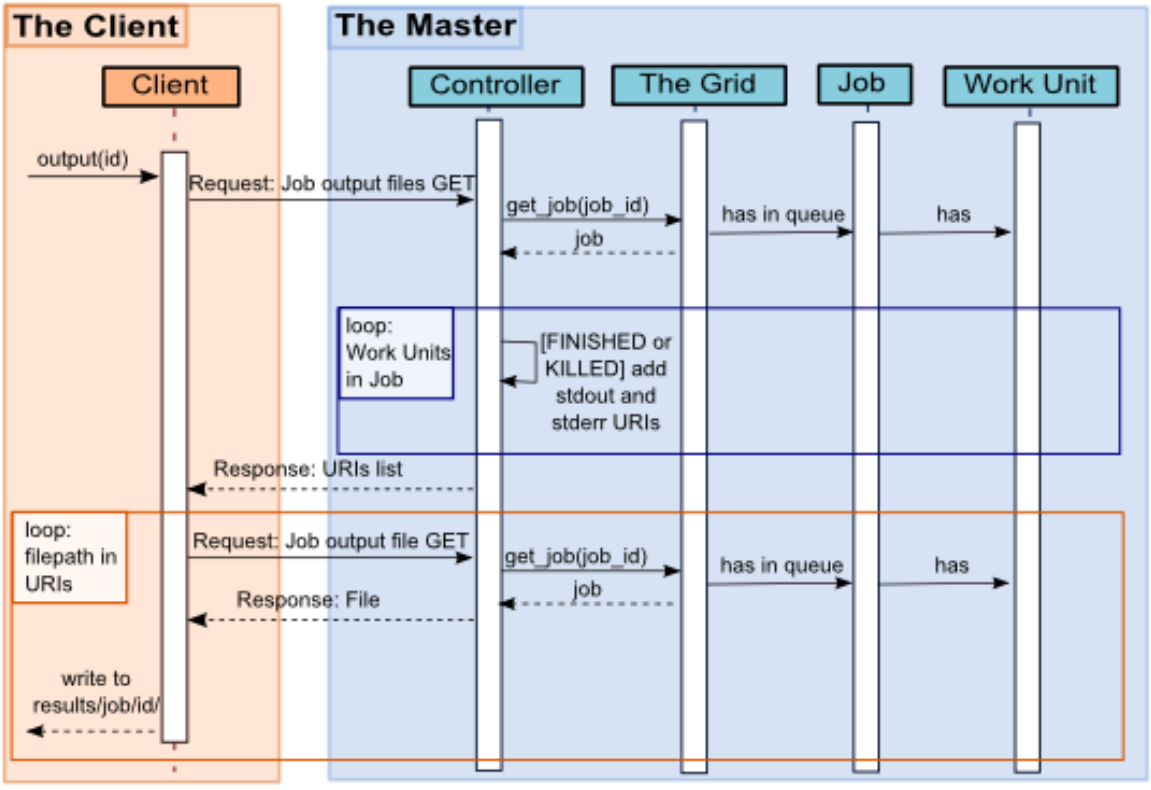

Figure 3.6: Get the output of a finished Job

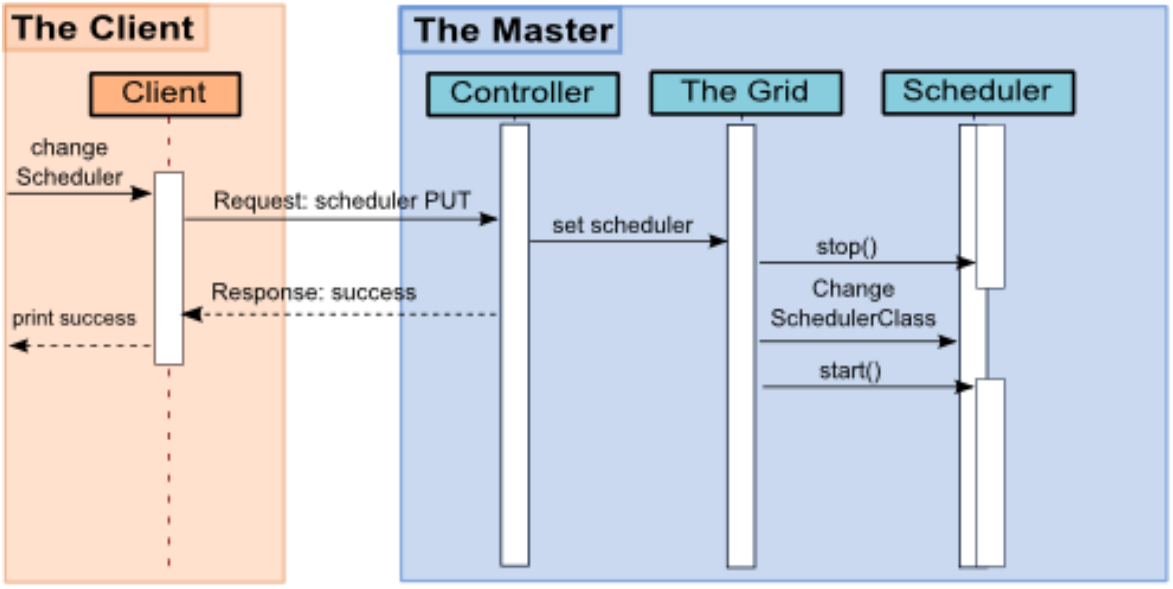

Figure 3.7: Change the scheduler dynamically

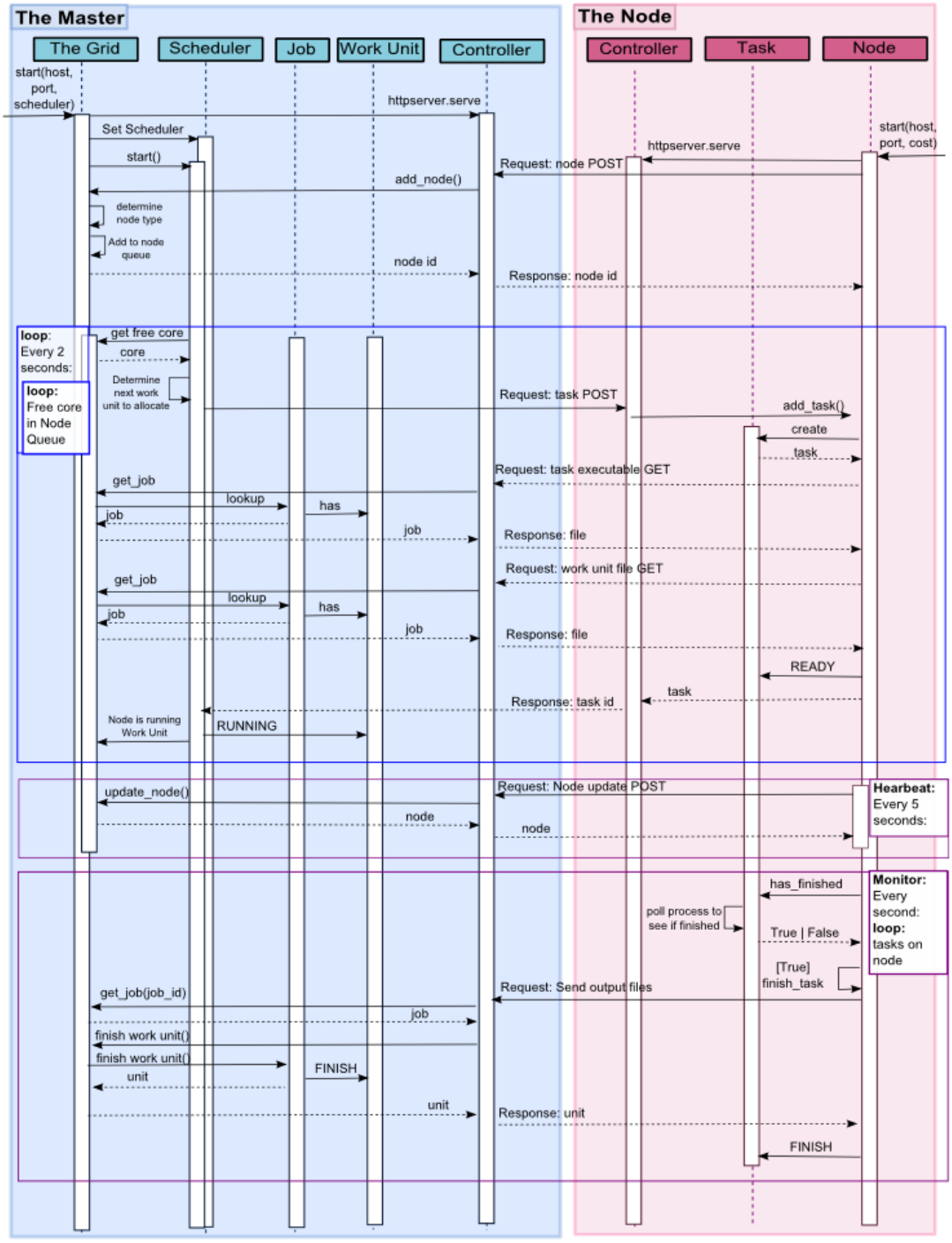

Figure 3.8: Interaction between The Master and The Nodes in The Grid

FOUR

Implementation

The Grid is implemented using Python 2.6, with a package dependency on Paste. Paste is a multithreaded HTTP server which is used as the underlying communication protocol in The Grid. Communication between The Client and The Master, and The Master and The Nodes is done via an agreed JSON based API.

Python 2.6 was chosen for both internal and external reasons. Internally, the development group all knew Python better than any other language which was suitable for the task. Secondly, development in Python is quite fast, though Python itself can be quite slow. As the main latency in The Grid is in the sending of input files and executables around, the computational speed loss due to language choice was deemed negligible. Externally, Python 2.6 was chosen due to its commonly installed nature which would make it easier to install The Node software on potential nodes.

The dependency on an external library Paste was determined necessary, though external libraries were avoided where possible. This was done to both avoid licensing issues, as well as making the software easier to install as it needed to be portable and installed on many different environments due to the heterogeneous nature of The Grid. Neither of the WSGI servers available in the standard library of Python were suitable as neither is multi-threaded, which is unworkable due to having to send possibly large files around that would otherwise block the server from accepting new requests.

The choice of HTTP to communicate over was done so because of the familiarity of the development team with HTTP based communication protocols and the development of RESTful APIs with which to communicate via JSON over HTTP. It also allowed for the easy development of a web based GUI which both had the benefit of playing to the development teams strengths as well as allowing for a interface to The Grid that is accessible anywhere in the world via the internet. HTTP also has built in authentication which was utilised in order to provided a level of security for who can access and use The Grid. HTTP is also extendible to use HTTPS which would allow for the easy securing of communication within The Grid as a possible future extension of the software.

JSON specifically was chosen as it is light-weight and easier to parse than XML, while also being easily used from within JavaScript which allowed the web service on The Master to utilise the same RESTful API as the Client and Nodes do.

The code was developed following the MVC style architecture, though as the majority of the views are actually JSON, the majority of the code is organised into Models and Controllers. Each of the web servers on The Master and The Nodes listen to a provided list of Routes which map URIs and request type (GET, POST, PUT, DELETE) to a controller which calls any specific functions from the model and returns the relevant output JSON.

4.1 The Master

The Master instantiates a HTTP Server on a given hostname and port, which it listens on for both communications from instances of The Client as well as web requests to the Graphical User Interface provided. The Master also uses this HTTP Server to accept messages from each of The Nodes in The Grid.

4.1.1 The Scheduler

The Master contains a Scheduler, which runs inside its own thread. It periodically polls the internal node list to see if any nodes have become available. If a node is available, it will check to see if there are any Work Units available that can be run on that node. If there is a job available, then The Scheduler will allocate the Work Unit to that node. It will then continue looking at available nodes and attempting to find available Work Units for them.

The Scheduler itself is itself contained within The Master and can be dynamically changed. In the case that a Scheduler is swapped however, the internal state unique to one Scheduler type will be lost and all jobs currently in the queue will be allocated using the new scheduling algorithm. The Scheduling algorithms available are: FCFS (First come, first served), Round Robin, Earliest Deadline First, Cost Constrained Earliest Deadline First, and Mutli-level Priority Queues.

4.1.1.1 First Come First Serve

FCFS completes each job sequentially as they arrived. Jobs are ordered by the time they were created, and all of the Work Units in a job will be completed before Work Units from another job are executed. If a job is created but is not ready (not all files have arrived on The Master) then a job that has been created later but is ready will be executed. When the earlier job becomes ready, all Work Units from the earlier job will be executed before resuming the job that was ready first but created later. This is done as to not unfairly punish jobs with large input files by making them wait until they are ready to be considered in the queue.

4.1.1.2 Round Robin

Round Robin is similar to FCFS however rather than executing all of the Work Units from a job before moving to the next one, it iterates over the scheduled jobs, allocating them one Work Unit at a time. This is achieved by storing a persistent queue of Job IDs between polls checking the internal node list. The Work Unit from the front of this queue is allocated to a free node, and then this job ID is placed at the back of the queue. New jobs which are sent to the grid are placed at the back of the queue.

4.1.1.3 Earliest Deadline First

Earliest Deadline First takes into account the deadline of the job and prioritises jobs by earliest deadline first. That is, if a job needs to be finished by tomorrow and there is

another job that can be finished in a week, then the job that needs to be finsihed by tomorrow will take preference. As Jobs are allocated by schedulers as Work Units, a job may execute some work units before a Job is created that has an earlier deadline. In this event any remaining Work Units from the already running job will be placed behind the Work Units of the new scheduler. As a jobs deadline is a fixed amount of time in the past, indefinate starvation of a job cannot occur as eventually the deadline of the job will make it the highest priority to be completed. This deadline is calculated as the job's specified deadline minus its wall time, as this is the latest date the job can start. This prevents long jobs from being starved out until they can't possibly finish by shorted jobs.

4.1.1.4 Cost Constrained Earliest Deadline First

Deadline/Cost First takes into account not only the deadline of each Work Unit as in Deadline First but also takes into account the budget of the Job. It first ensures that a job only runs on nodes that are within the Jobs budget. It secondly preferentially places jobs with higher budgets about jobs with lower budgets.

4.1.1.5 Multi-level Priority Queues

This Multi-Level Priority Queue implementation splits The Nodes on The Grid into seperate queues. There can be any number of different queues, with a different proportion of The Grid allocated to them. The default queues are Default, Batch and Fast. Half of The Nodes on The Grid are allocated to Default, which is for any generic job between 1 hour and a few days. Batch is allocated 30% of the nodes and is for jobs that will take a week or more. Fast is allocated 20% of the nodes and is for jobs that are less than an hour. These priorities reserve nodes for specific job types without indefinately starving any one type of job if there is an excess of another.

This scheduler also stops large jobs from taking up most of the queue during offpeak periods, and then causing a large back log of other jobs. This is an unfortunate side effect of not being able to pause and resume a Work Unit once it has begun. While Deadline First and such can interupt a low priority job in terms of stopping further execution of additional work units, if a job is already running a number of work units which each may take a long time, there is no way to interrupt these running processes.

Different scheduling algorithms are used for each of the three different types of queues. These were all cost constrained to prevent jobs from running on nodes they couldnt afford. Where two jobs were equal in the algorithm, the one with the highest budget took preference. For the Batch queue, FCFS was used because it provides high throughput. The Fast queue uses Round Robin for its good response time, and the Default queue uses Earliest Deadline First.

4.2 The Node

Each Node, like The Master, also instansiates a HTTP Server on a given hostname and port. It listens on this for communications from The Master. The Master sends job

information to The Node, which in turn uses this information to request from The Master any files that The Node requires in order to complete the job it has been assigned. The Node will then execute a Work Unit once it is told by The Master that the Work Unit is READY. The Node will then preiodically check whether the Work Unit has finished executing. Once it has finished, The Node will report back to The Server that the Work Unit is complete and will then send back any information written to stdout or stderr during the Work Unit's execution.

4.2.1 The Heartbeat

The Heartbeat is a thread spawned from The Node which controls the sending of the Node's heartbeat to The Master. This heartbeat lets The Master know that The Node is still there, if this heartbeat is not received by The Master, it will be assumed to be offline. This heartbeat also detects for a loss of connection to The Master. If connection to The Master is lost, The Node will attempt to reregister itself to The Master. The Heartbeat has the additional task of monitoring the health of The Node. It reports information such as the CPU usage to The Server so that overall statistics of The Grid can be monitored at a Grid-wide level.

4.2.2 The Monitor

The Monitor is a thread spawned from The Node which monitors the state of all running processes and reports back to The Master when a Work Unit has finished executing. The Monitor will check for Jobs that may have exceeded their allotted wall time and kill them, returning their current progress.

4.3 The Client

The Client is built on top of the available API in Python and allows the creation of new Jobs via a command line interface. It additionally allows for the monitoring of a running Job's status. As well as retreiving the output files created during a Jobs execution. The Client also allows a user to kill a running job early and retrieve any output that had been generated until that point. It is also possible to dynamically modify the Scheduler being used by The Grid, however this requires the client to be logged in with an Administrator level account. The Client must be run with a provided username and password that is valid for use with The Grid. These username and password combinations can be either Client level, or Administration level. The Client level allows the user to create, view and kill jobs. The Administration level is the same as Client level, however with the additional functionality of being able to change The Scheduler.

4.4 The Web Interface

The Web Interface is built on top of the same API as The Client, however it is written with a combination of HTML and JavaScript to run from The Browser. The Web

Interface is served by The Master and is accessible to any computer that can see The Master. The Web Interface has the additional feature of also being able to easily view the output log of The Scheduler, for debugging or monitoring purposes, as well as being able to see which Nodes are available, and what Work Units have been assigned to them.

FIVE

The Grid: API

Both the Web based User Interface and the Client application (Client.py) communicate with the master node of The Grid through a REST API. REST stands for Representational State Transfer and is becoming the de-facto standard for web based API's, such as those used by Twitter and Facebook. The key benefits to using this technology are that many tools exist for consuming these services; it works over HTTP enabling consumption of services on mobile devices and provides a consistent model to develop with.

Rather than using XML, we have chosen to use the JSON format. The primary driver for selecting JSON is its simplicity. XML can easily become cumbersome for development due to the challenges of maintaining XML schemas in a rapidly evolving development process. JSON is also extremely well supported within Python and Javascript which greatly simplified development.

The REST API is secured using based HTTP Authentication, preventing unauthorized access to The Grid.

5.1 API End Points

5.1.1 Submit Job

The meta-data of a job must be submitted before files can be submitted and before the job can be started.

```
URL:    /job
Method: POST
Parameters:
    name            label for identifying the job
    wall_time       time allowance for the job (HH:MM:SS)
    deadline        date and time that the job must be completed by
    budget          amount of money this job may cost
    job_type        type of job (BATCH, FAST, DEFAULT)
Returns:
    job_id
```

5.1.2 Submit Executable File

Only a single executable file may be uploaded to the server for a particular job. If multiple files are uploaded, then latter uploads will replace the initial file.

```
URL:    /job/<job_id>/executable/<file_name>
Method: PUT
Parameters:
    job_id          the id of the job to submit the file to
    file_name       name of the executable file
    file_content    content of the file in the content body
Returns:
    job_id, file_name
```

5.1.3 Submit Input File

Many files may be associated with a single job, with the executable being executed with a single input file in each invocation which may occur on different servers.

```
URL:    /job/<job_id>/files/<file_name>
Method: PUT
Parameters:
    job_id          the id of the job to submit the file to
    file_name       name of the executable file
    file_content    content of the file in the content body
Returns:
    job_id, file_name
```

5.1.4 Start Job

After all the required files have been uploaded, the job may be started using the above mentioned url.

```
URL:    /job/<job_id>/status
Method: POST
Parameters:
    job_id      the id of the job to start
    status      must always be "READY"
Returns:
```

5.1.5 Kill Job

A job may be killed when it is Queued, Ready, Pending or Running, in which case all work items that have been queued or are running will be halted. The output for jobs that have already been run will still be available.

```
URL:    /job/<job_id>
Method: DELETE
Parameters:
    job_id      the id of the job to kill
Returns:
```

5.1.6 List Available Nodes

A list of all currently connected nodes is provided at this url along with the work items that each node is currently working on.

```
URL:    /node
Method: GET
Parameters:
Returns:
    JSON encoded string listing available nodes.
```

5.1.7 Node Information

Get all the information The Grid stores on a specific Job

```
URL:    /node/<node_id>
Method: GET
Parameters:
    node_id     the id of the node
Returns:
    JSON encoded string of the node
```

5.1.8 List Jobs

A list of all jobs that are in the pending, ready, queued, running, finished or killed state along with each jobs work items and the output files associated with those work items.

```
URL:    /job
```

```
Method: GET
Parameters:
Returns:
    JSON encoded string listing jobs that have been submitted.
```

5.1.9 Job Information

Get all the information The Grid stores on a specific Job

```
URL:    /job/<job_id>
Method: GET
Parameters:
    job_id      the id of the job
Returns:
    JSON encoded string of the job
```

SIX

Evaluation

The Grid allows for the sending of self-contained executables across a heterogeneous grid. It suffers from a few key weaknesses. The first of these is the insecure method of communication between The Master, The Client, and The Nodes. In a distributed computing environment, a lot of uses are confidential and as such the transfer of files to remote computers needs to be secured from outside interference and interception. One way in which this could be resolved is to use SSL in order to take advantage of the HTTP based transfer and authentication layer used in The Grid.

It also has the weakness that executables must be entirely self contained. Without being entirely self contained, a executable may look for linked system libraries which will not be present on the distributed nodes. Any compiler that uses architecture specific compilations may also suffer as these executables may not operate on all Nodes in The Grid. It is our recommendation that users of The Grid ensure that each Node has minimum versions of common languages such Python and Java installed on all machines. Users of The Grid can then be assured that if they write their software to work with these languages, that their software will execute.

There is also an inherent weakness in allowing users access to a remotely distributed network. Doing so puts a large amount of trust in the user, who through malicious or simply accidental means, could do quite a bit of damage to The Grid. This is always going to be a factor if you are allowing users to execute remote code in a distributed environment. To minimise this, a more complex virtual file system could be used to better sandbox and limit the functionality the submitted executables have over the system.

The Grid also only accepts very limited software in terms of the way it operates. The only form of parallelism available is for problems that are “Embarrassingly Parallel”. That is, an executable must be supplied with the input files, pre-split, which it can then work on independently without inter-node communication. The weakness of this method is still that the files must be pre-split, and the output files re-joined manually or by another program in order to be used by The Grid.

While The Grid is quite simple in its current state, with a few improvements as outlined in the Future Improvements section, and outlined in this section, it could potentially become a fairly powerful tool for easily deploying distributed heterogeneous computer grids for the delivery of computation power as a resource. The basic requirements of Python in order to get started mean that it is quite simple to get running on a small internal network of desktop machines and would allow the use of their processing power to work on distributed tasks.

SEVEN

Future Improvements

There are a number of key improvments that could make The Grid much better.

- The use of SSL in order to secure the communication between The Master and The Nodes and The Clients.
- The use of a proper job specification format that would allow the sending of non-self-contained executables around the heterogenous grid.
- The use of a virtual file system that would allow better sandboxing of executables on The Grid.
- The allowance of more complex input/output execution than stdin and stdout.
- The allowance of executables that run on more than one processor
- The Determination of computational power at runtime to factor into Node-type load balancing. This would prevent load balancing from allocating improperly Nodes with an unusually large number of cores, RAM or processing power.

EIGHT

Contributions

Each team member of the group has contributed approximately 25% of the work to this project. Specific leading contributions of each team member are outlined below, however all team members contributed to all aspects of the developement and design of The Grid.

Adam Whiteside & Scott Richie - Backend Communication and Framework

Terence Siganakis - Graphical User Interface

Andrew Vadnal - Scheduling Algorithms