Team-based Project Work – Grid System

Andrew Banh, Scott Beca, Cheuk Man Mong

433-678: Cluster and Grid Computing, Semester 1, 2011
Department of Computer Science and Software Engineering
The University of Melbourne, Australia

Email:
a.banh@pgrad.unimelb.edu.au
s.beca@ugrad.unimelb.edu.au
c.mong2@ugrad.unimelb.edu.au

Abstract
Our team was tasked with designing and developing a Grid computing system. This system required a single Master which would control what a set of remote Workers were computing. After comparing different technologies, Java, Sockets and Swing were chosen as the preferred platform on which we would develop this system. The user controlling the Master is supplied with a number of commands that allow them to add Workers, add applications that are allowed to run on the Workers, add Jobs, get updated information of both Workers and Jobs, change the Job scheduling algorithm, cancel Jobs, reschedule Jobs and delete Jobs. A Worker waits for commands from a verified Master and then completes any Jobs it is asked to do. During our testing, our team was successfully able to schedule multiple Jobs to be run on multiple Workers using different scheduling algorithms.

1 Introduction

The aim of this project was to create a working grid system that consisted of a Master that controlled a set of Workers remotely. Our implementation consists of two separate programs – the Master responsible for managing and scheduling jobs to Workers and collecting the completed output files and a Worker program that executes a given application for a job.

The implementation is done in Java using Object Oriented programming principles, and relies on socket programming for communication between master and worker nodes. This system can be run on any networked computers with IP address and ports that are free to use.

1.1 Features

- Can load available Applications, Jobs and Workers from file
- Runs Jobs on Workers and retrieves output file.
- Connection with Worker nodes
    - Password verification for each connection
    - Monitors health status of workers
    - Can start and cancel Jobs
- Ability to reschedule jobs (including Failed Jobs)
- Job management system which schedules jobs
    - Supports job priorities and delayed starting times
- Scheduling Algorithms for assigning Jobs to Workers
    - Earliest Deadline First
      (Completes all jobs as fast as possible)
    - Earliest Deadline First with Budget Constraints
      (Completes all jobs as cheap and fast as possible)

2 Architecture

2.1 Language

Java was chosen as the language for the project as the majority of our group members felt comfortable with the language. Java provides Socket Programming that is an essential part of the project and is a familiar tool as it was taught earlier in the subject. Java also provides a simple GUI toolkit (Java.Swing), which proved to be very helpful in constructing a simple and easy to use interface.

2.2 Structure Breakdown

2.2.1 Master

The Master program is broken up into several classes, each with its own specific roles:

Object	*Description*
Master	The class that contains the main function and is at center of the whole program, it will call on other classes to help make the program work.
GUI	The graphical user interface that is shown when we initially run our program. It is here that you will find buttons and tables that give information to the user in a way that is easily understood and also provides an environment where the user can monitor the system.
AddJobFrame	The window that opens up when a request is made to add a new job. It requires the user to input application, application arguments, priority of the job, the maximum budget and also a deadline for the job to be completed. When all of these are filled out correctly, a new job will be created and given to the Job Manager to handle.
AlgorithmFrame	Also represents a user input request window. The user is required to select from radio buttons to choose one algorithm to be used for job scheduling. Actions in this frame may update the Job Manager's scheduling algorithm.
ApplicationManager	Keeps track of all the applications and input files that are currently allowed to be used by jobs. It also determines how and what data to display in the Applications/In files GUI table.
JobManager	Keeps track of all the jobs that are queued in the system. It provides methods to add and delete jobs and also query their statuses. The Job manager is also responsible for determining which job runs next (scheduling) based on the algorithm chosen and also the priorities of jobs. It also determines how and what data to display in the Job GUI table.
Job	An object representing a job that's defined at the master node and executed on the worker node. It contains information such as the application name, arguments, id, priority, budget, deadline and the current status of the job (queued, computing, complete, failed).
Connection	Provides the channel of communication between the master and the workers. It is based on socket programming and abstracts the low level details away for the other classes to use. It provides a few public methods to send and receive files, integers and strings. This class is common to both the worker and master since both have the same communication needs, and using the same class reduces the chance of de-

	synchronisation within calls.
WorkerManager	Keeps track of all the worker nodes that can be utilized by the master and also their state. Worker Manager will periodically check the status of its workers and also check if new workers are available. It will check if the worker is busy, free or if it has crashed. With this information the Worker Manager can determine where to send the next job. It also determines how and what data to display in the Worker GUI table.
WorkerHandler	The Worker Manager's abstraction on its worker. It contains information about the worker such as its IP address, which port communication is running on, identification and also its status (alive, dead, ready and busy).
WorkerCostComparator	Implements the Comparator Interface to allow WorkerHandlers to be sorted by (Cost * Time).
WorkerTimeComparator	Implements the Comparator Interface to allow WorkerHandlers to be sorted by Time.

2.2.2 Worker
The Worker program is similarly broken up into several classes with roles:

Object	*Description*
Worker	The bulk of the Worker program containing the main method for driving and all the support methods required for the Worker to run.
Connection	As above.
ServerConnection	ServerConnection is a subclass of Connection that extends Connection to work as a Server. While the underlying Connection remains the same as the Master version, the ServerConnection creates a ServerSocket and a method for opening new listening sockets on that Server.
Job	A Thread that represents the current assigned Job to the Worker. Starting the Thread runs the given application name with arguments in a process.
StreamRedirect	An object that is used to redirect the process output of the running Application in Job to a new output file.

3. Design

In order to create the program, several key design decisions had to be made each with benefits and drawbacks.

3.1 Network Protocol
During the planning stages we initially researched an API known as JMS (Java Messaging Service), part of the J2EE platform, which seemed useful as a form of communication between the Master and the Workers. It utilised Asynchronous Message Queues which, along with Topic Broadcast/Subscribe, could allow for more robust networking connections. However further research into this API revealed that there are many dependencies and additional libraries which made the program too bloated and difficult to work with.

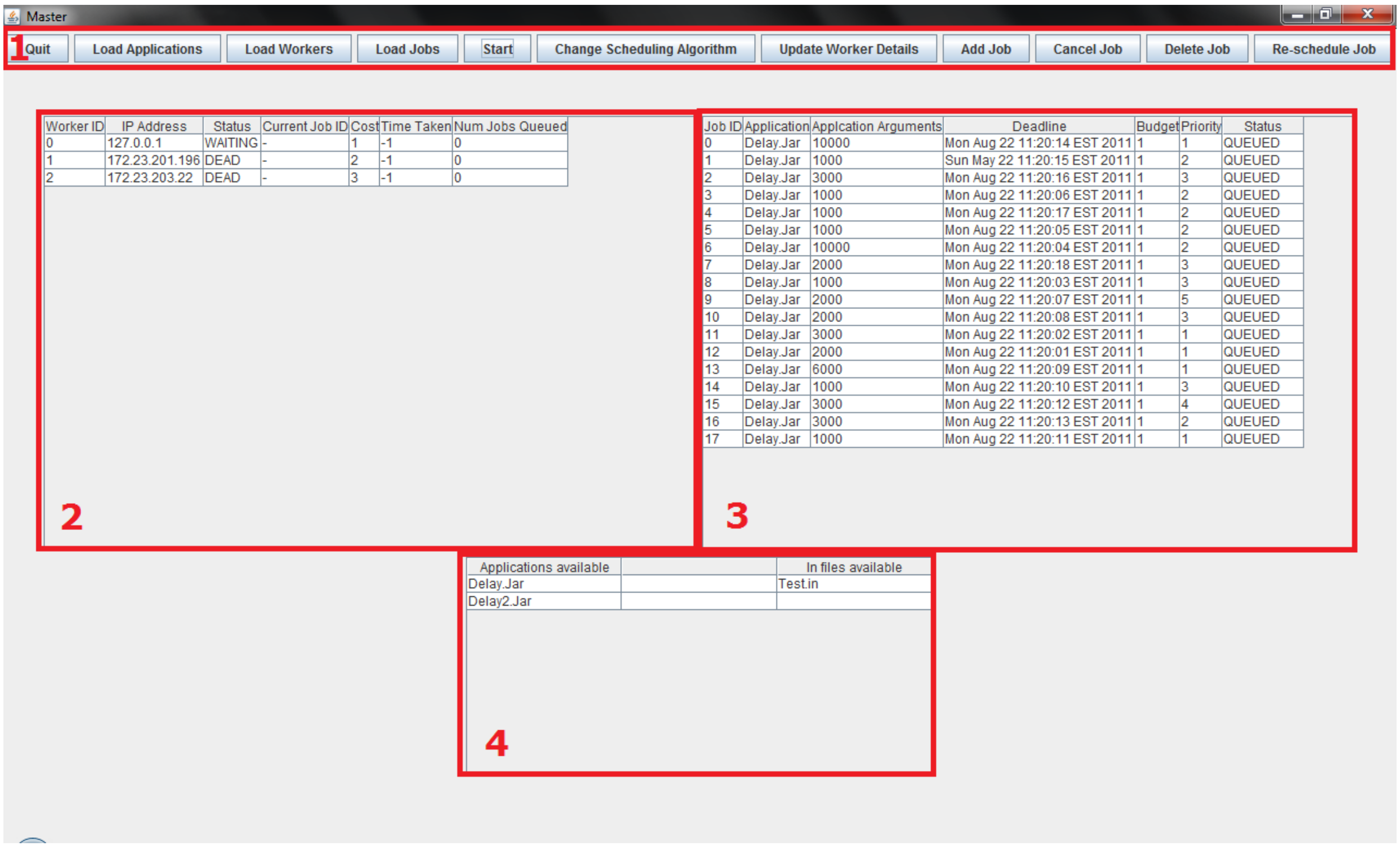
We decided not to use JMS because the cost and complexity outweighed the benefits mentioned above. Therefore we chose to go with socket programming which we were familiar with already and have had experience with it in Assignment 1.

Our networking is implemented by using short UDP messages because we found TCP, with features like ACK handshaking, unnecessary as the transferred message sizes were relatively small. The largest transfers that the program is responsible for are the sending of files, which are expected to be no more than 100kb. Connections are kept open only when they are active and being used. This keeps the Master and Workers loosely coupled and independent. The only downside is that regular reconnects require the connections to be verified every time.

3.2 Graphical User Interface

The Swing toolkit is the natural option to choose when working in Java. It is the most commonly used GUI toolkit for Java and is simple to use.

To keep the interface simple, all the buttons were put onto a panel and in turn put into the NORTH section of the main window in order to keep all the buttons at the top (section 1 of Figure 1). The tables are all kept in the CENTER section to ensure that the buttons and tables are kept apart (sections 2, 3 and 4 of Figure 1). If this was not done then buttons would appear on the same row as a table which could cause some tables to be pushed under each other and even out of view depending on the size of the frame which can be very confusing for the user.



Worker ID	IP Address	Status	Current Job ID	Cost	Time Taken	Num Jobs Queued
0	127.0.0.1	WAITING	-	1	-1	0
1	172.23.201.196	DEAD	-	2	-1	0
2	172.23.203.22	DEAD	-	3	-1	0

Job ID	Application	Applcation Arguments	Deadline	Budget	Priority	Status
0	Delay.Jar	10000	Mon Aug 22 11:20:14 EST 2011	1	1	QUEUED
1	Delay.Jar	1000	Sun May 22 11:20:15 EST 2011	1	2	QUEUED
2	Delay.Jar	3000	Mon Aug 22 11:20:16 EST 2011	1	3	QUEUED
3	Delay.Jar	1000	Mon Aug 22 11:20:06 EST 2011	1	2	QUEUED
4	Delay.Jar	1000	Mon Aug 22 11:20:17 EST 2011	1	2	QUEUED
5	Delay.Jar	1000	Mon Aug 22 11:20:05 EST 2011	1	2	QUEUED
6	Delay.Jar	10000	Mon Aug 22 11:20:04 EST 2011	1	2	QUEUED
7	Delay.Jar	2000	Mon Aug 22 11:20:18 EST 2011	1	3	QUEUED
8	Delay.Jar	1000	Mon Aug 22 11:20:03 EST 2011	1	3	QUEUED
9	Delay.Jar	2000	Mon Aug 22 11:20:07 EST 2011	1	5	QUEUED
10	Delay.Jar	2000	Mon Aug 22 11:20:08 EST 2011	1	3	QUEUED
11	Delay.Jar	3000	Mon Aug 22 11:20:02 EST 2011	1	1	QUEUED
12	Delay.Jar	2000	Mon Aug 22 11:20:01 EST 2011	1	1	QUEUED
13	Delay.Jar	6000	Mon Aug 22 11:20:09 EST 2011	1	1	QUEUED
14	Delay.Jar	1000	Mon Aug 22 11:20:10 EST 2011	1	3	QUEUED
15	Delay.Jar	3000	Mon Aug 22 11:20:12 EST 2011	1	4	QUEUED
16	Delay.Jar	3000	Mon Aug 22 11:20:13 EST 2011	1	2	QUEUED
17	Delay.Jar	1000	Mon Aug 22 11:20:11 EST 2011	1	1	QUEUED

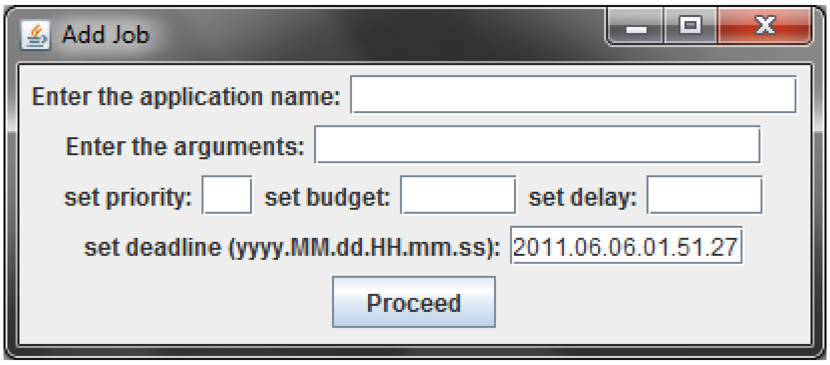
Applications available		In files available
Delay.Jar		Test.in
Delay2.Jar		

Figure 1**: Master GUI Window:**

1. Program control buttons – These buttons allow the user to control the Master program and get it to perform various functions. The functions of these buttons are explained in more detail in section 4.1 of this report.
2. Worker information table – stores and displays relevant information about each worker including their IP, status, current Job and various stats used by the different scheduling algorithms.
3. Job information table – stores and displays relevant information about each job including their Application details, status and various other details used by the different scheduling algorithms.

4. Application and input file table – stores and displays the applications and input files that have been loaded into the system, distributed to the workers and are available for use by jobs.

When a user clicks on the "Add Job" button, the Add Job window shown in Figure 2 is displayed. All the fields except the arguments and delay are required for a job to run and this window allows a user to enter that information.


Add Job

Enter the application name:	
Enter the arguments:	
set priority:	
set budget:	
set delay:	
set deadline (yyyy.MM.dd.HH.mm.ss):	2011.06.06.01.51.27

Figure 2: Add Job GUI Window.

When a user clicks on the "Change Scheduling Algorithm" button contained in section 1 of Figure 1, the Choose Algorithm window show in Figure 3 is displayed. This window allows the user to choose one scheduling algorithm from a selection of three. After clicking the "Proceed" button, the new algorithm is set to use straight away by sorting the worker and job tables into an order defined by the algorithm and is continued to be used for each new job.

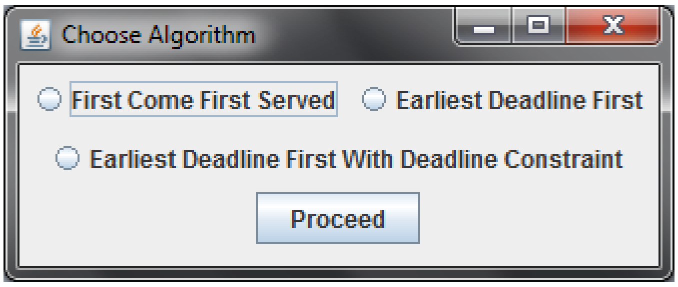


Figure 3: Scheduling Algorithm Selection GUI Window.

3.3. Advanced Scheduling Techniques

Scheduling Algorithms for time and cost were implemented in both Worker and Job choice. In addition, for Job selection there is the priority queue and scheduled delay.

For Jobs:

- When Jobs are created they are given a Deadline, Priority, Budget and Scheduled Delay.
- Jobs are placed into an ArrayList which is sorted based on their priority (lower numbers equals a higher priority) then deadline.

- Any Job with a delay that hasn't elapsed yet has the lowest priority and is sorted at the end of the list.
- Sorting based on priority ensures a priority queue system.
- Secondary sorting of deadline ensures that the earliest deadline comes first.
- For budget considerations, when Jobs are assigned, if they do not meet the cost of the Worker, then they are placed in a temporary queue, and re-added (and re-sorted) in the main queue after an assignable Job is found.

For Workers:

- Each Worker is allocated Jobs (until there are no Jobs left) based on the number of possible Jobs they could execute between the current time and longest 'time taken' by a Worker.
- Jobs are then assigned to Workers with a possible Job number higher than zero in order of lowest time (WorkerTimeComparator Comparator object) or lowest cost by time (WorkerCostComparator Comparator object).

4. Implementation

4.1 Master

The Master can accept one optional argument at runtime (“nogui”). If the argument is defined then the application will run with only command-line input and output. If the argument is not given at runtime then a GUI will be displayed to the user which they will use to interact with the program. To launch the Master, usage is as follows:

```
Usage:          java Master [nogui]
```

The user is supplied with a number of command-line commands or GUI buttons to interact with the Master. The commands available to the user are:

Command	*Description*
Exit	Commands the Master to safely shutdown. The Master cycles through every Worker and Job and calls a shutdown() method on each of them to allow them to do any clean-up they require. Any currently working Workers are sent a Cancel Job command.
Load Applications	From two predefined files, “application.txt” and “ins.txt”, loads a list of available applications that can be scheduled to run on the Workers and any necessary input files.
Load Workers	From a predefined file, “workers.txt”, the Master creates a number of new WorkerHandler objects used to interact with the Worker clients. On creation, each WorkerHandler tries to send a Setup Worker command to its Worker.
Load Jobs	From a predefined file, “jobs.txt”, the Master creates a number of new Job objects.
Start	Starts the Master assigning Jobs to Workers depending on the Scheduling Algorithm selected.
Change Scheduling Algorithm	Allows the user to select one of three scheduling algorithms: • First Come First Serve • Earliest Deadline First • Earliest Deadline Constrained by Budget
Update Worker	The Master sends each Worker a Check Status command and refreshes the

Details	Worker GUI Table with the updated details.
Add Job	Brings up a new window that allows the user to enter in details for a new Job to be added to the list of Jobs.
Cancel Job	Cancels the currently selected Job in the Job GUI Table. The user is only allowed to cancel a queued or currently computing Job. If the Job is currently computing, a Cancel Job command is sent to the Worker currently working on it. If no Job is selected in the Job GUI Table, a message box is shown to the user informing them of this.
Delete Job	Deletes the currently selected Job in the Job GUI Table. If the Job is currently computing, a Cancel Job command is sent to the Worker currently working on it. If no Job is selected in the Job GUI Table, a message box is shown to the user informing them of this.
Re-schedule Job	Re-schedules a failed/cancelled Job by setting its status to “queued”. If the current Scheduling Algorithm is set to FCFS, the re-scheduled Job is sent to the end of the queue, otherwise, it’s put into the queue at the position the Scheduling Algorithm defines.

4.2 Worker

Workers can accept several settings as arguments at runtime (Worker name, Port and Password), if the arguments are not defined, then default settings will be used. The Worker constantly listens as a server on the given port and recreates a listening socket when the previous one is closed. To launch the Worker, usage is as follows:

```
Usage:          java Worker <Worker Name> <Port> <Password>
```

Workers will wait for an incoming connection from the master and upon connection the Master will send a password to the Worker. If this password matches the Worker's predefined password the Worker will continue serving the Master, if the password doesn't match the Worker will exit. Once the password is verified, the Worker will receive a signal from the Master indicating the type of command that is given; hence the Worker will know what to do with subsequent messages.

The type of command given can be one of the following:

Command	*Description*
Setup Worker	Warns the Worker that the Master is ready to send executable applications (.Jar) and input files. The Worker receives the number of executable/input files and the filename/file for each file.
Check Status	A request for the status of the Worker. The worker responds with its own Integer corresponding to its status (“error”, “waiting”, “running” and “finished”). If the Worker has finished, it will then send the Master the completed job’s output file.
New Job	A Job assignment by the Master. The Worker will receive the name and arguments to execute and the Worker will respond with a message indicating the success of starting the Job.
Cancel Job	A commandment by the Master to destroy the process of any currently running Job. The Worker will then revert to a “Waiting” state and respond to the Master with its state.

5. Testing and Result Evaluation

In preparation for the live demonstration and to verify the working status of the program, a set of testing data was created.

5.1 File Lists

A number of files were created containing the data for Application, Worker and Jobs for the Master to load in as input and parse into objects.

- application.txt - Contains the file names for all test executable applications.
- ins.txt – Contains the file names of all test input files.
- worker.txt - Contains the address, port, cost and password for each Worker that it will connect to.
- jobs.txt - Contains a set of arbitrarily created jobs which each contain the data to run an Application as well as some extra scheduling parameters (priority, deadline and scheduled delay)

5.2 Test Applications

The test application for the test set of data is 'Delay.Jar', a java application compiled as an executable Jar file that takes an input as an argument and sleeps for that input (milliseconds). The purpose of this application was to simulate the time that a job could be expected to take as any other simple application would not have a significant runtime without being overly complex or unpredictable. To launch the test application, usage is as follows:

```
Usage:                    java –jar Delay.Jar <sleep_duration>
```

5.3 Focus

The test data was created in order to simulate a regular instance of the system. There are three main areas that required testing in order to verify their working conditions.

5.3.1 Graphical User Interface

In order to run the test, the tester must utilise the Graphic User Interface and interact with the buttons in order to initialize the instance by loading the various lists and observing the results through the interface.

5.3.2 Networking / Connections

The backbone of the system lies in the networking and communication between the Master and its Workers. In order to ensure that this is working, the Workers display a verbose amount of feedback to the console output by relaying each command it receives and action it takes along with errors.

5.3.3 Scheduling Algorithms / Job Queue

Through the Graphical User Interface, it is possible to track each job and Worker and their current statuses and relations. This allows the tester to ensure that the implementation of the Scheduling Algorithms and Priority Queue are working correctly and that jobs are being assigned correctly to the appropriate Workers.

6. Team Member Contributions

Project Sections	*Team Members*		
Master Client	**Andrew**	**Scott**	**Cheuk**
Master Client		X	
Job / Worker Management		X	X
Job Queue (Priority / Scheduling)	X	X	X
Scheduling Algorithms	X	X	X
Networking / Connections	X	X	
Graphical User Interface - Display		X	X
Graphical User Interface – Interaction / Configuration			X
Worker Client	**Andrew**	**Scott**	**Cheuk**
Worker Client	X		
Networking / Connections	X	X	
Running Applications	X		
Testing	**Andrew**	**Scott**	**Cheuk**
Test Applications	X		
Networking / Connections	X	X	
Graphical User Interface		X	X
Scheduling / Job Management		X	
Report	**Andrew**	**Scott**	**Cheuk**
Introduction			X
Architecture	X		X
Design	X	X	X
Implementation		X	X
Testing	X		X

7. UML Diagrams

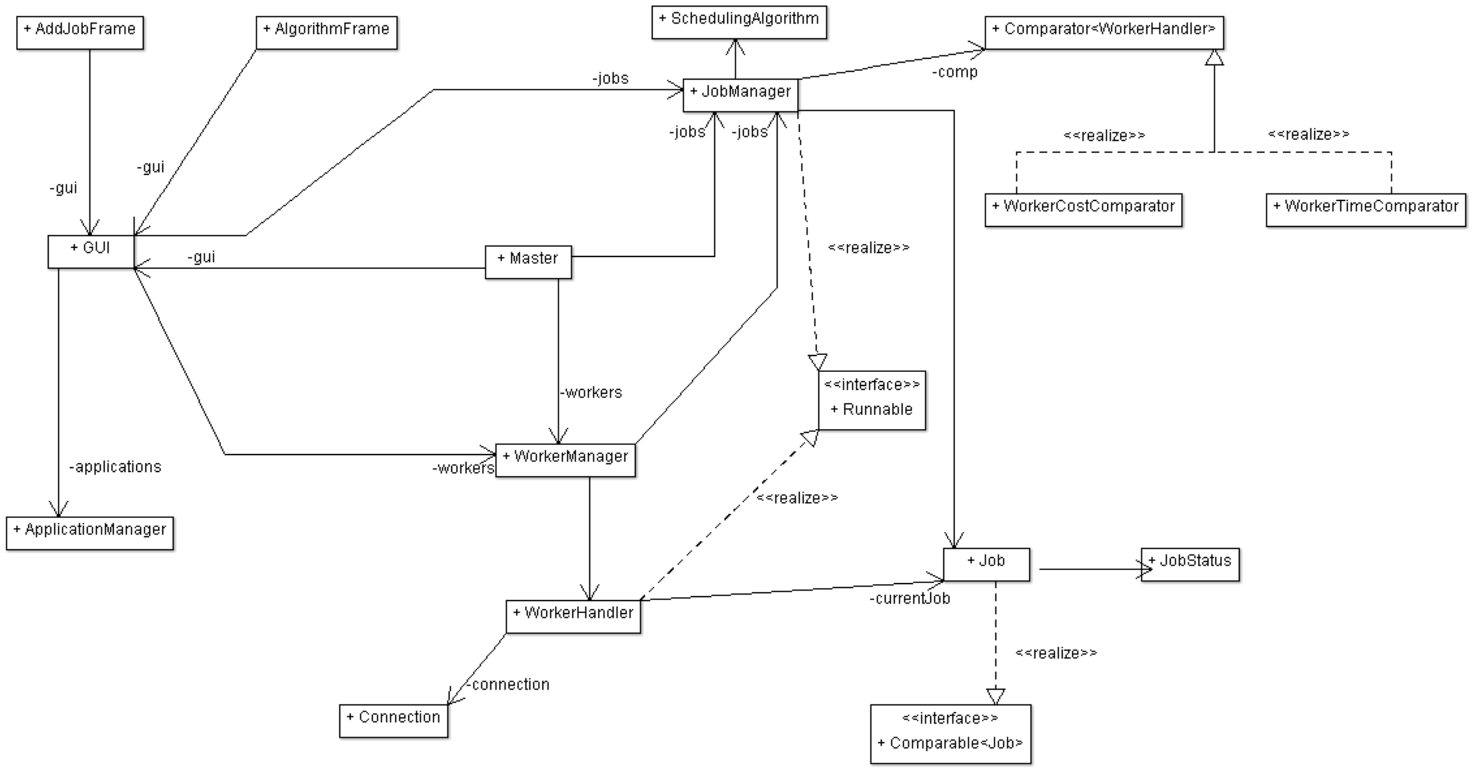


Figure 4: UML Class Diagram for the Master program.
Please note that because the Master is so large, this report will not show every detail for every class.

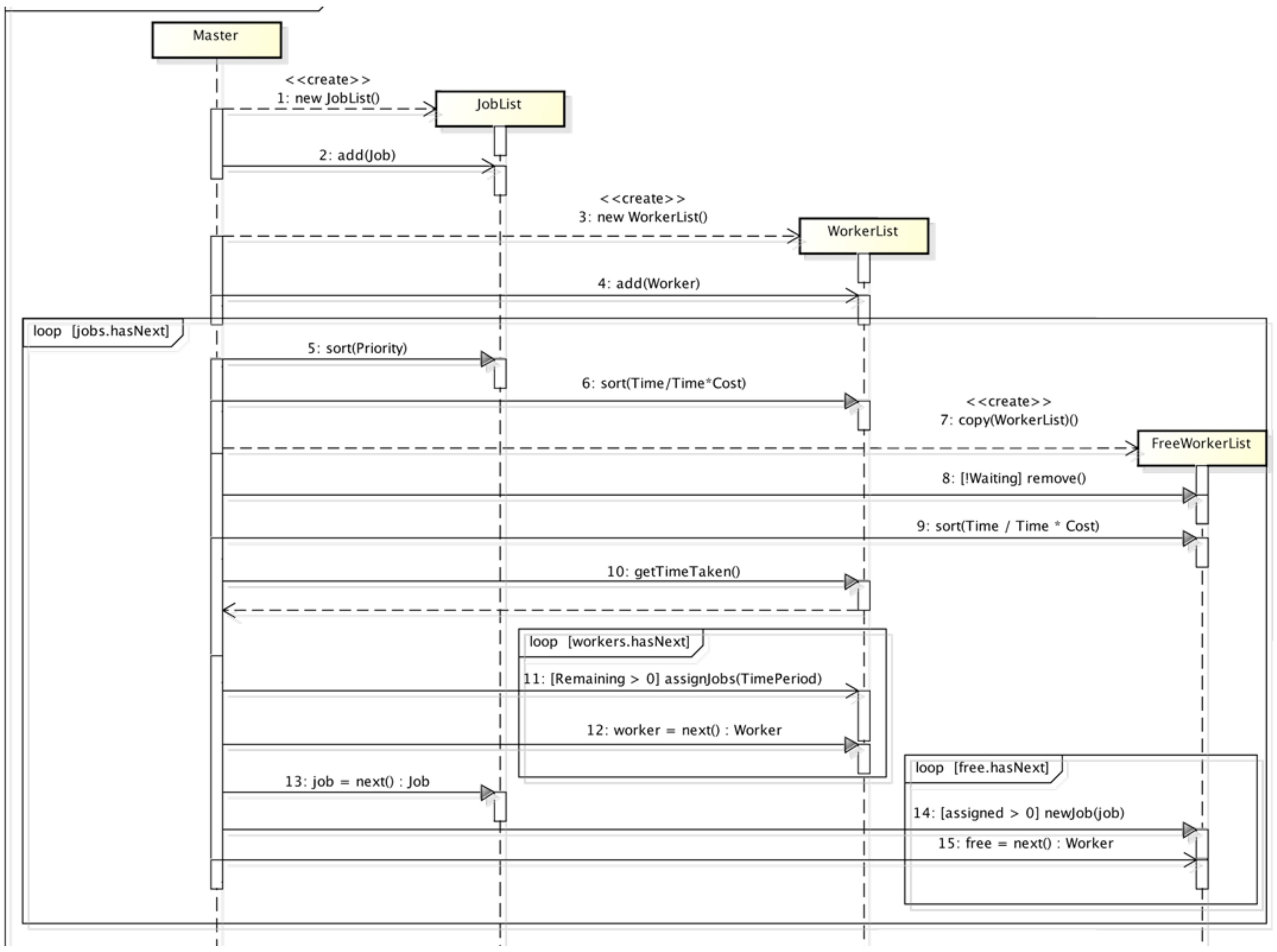


Figure 5: UML Sequence Diagram for the Master program.

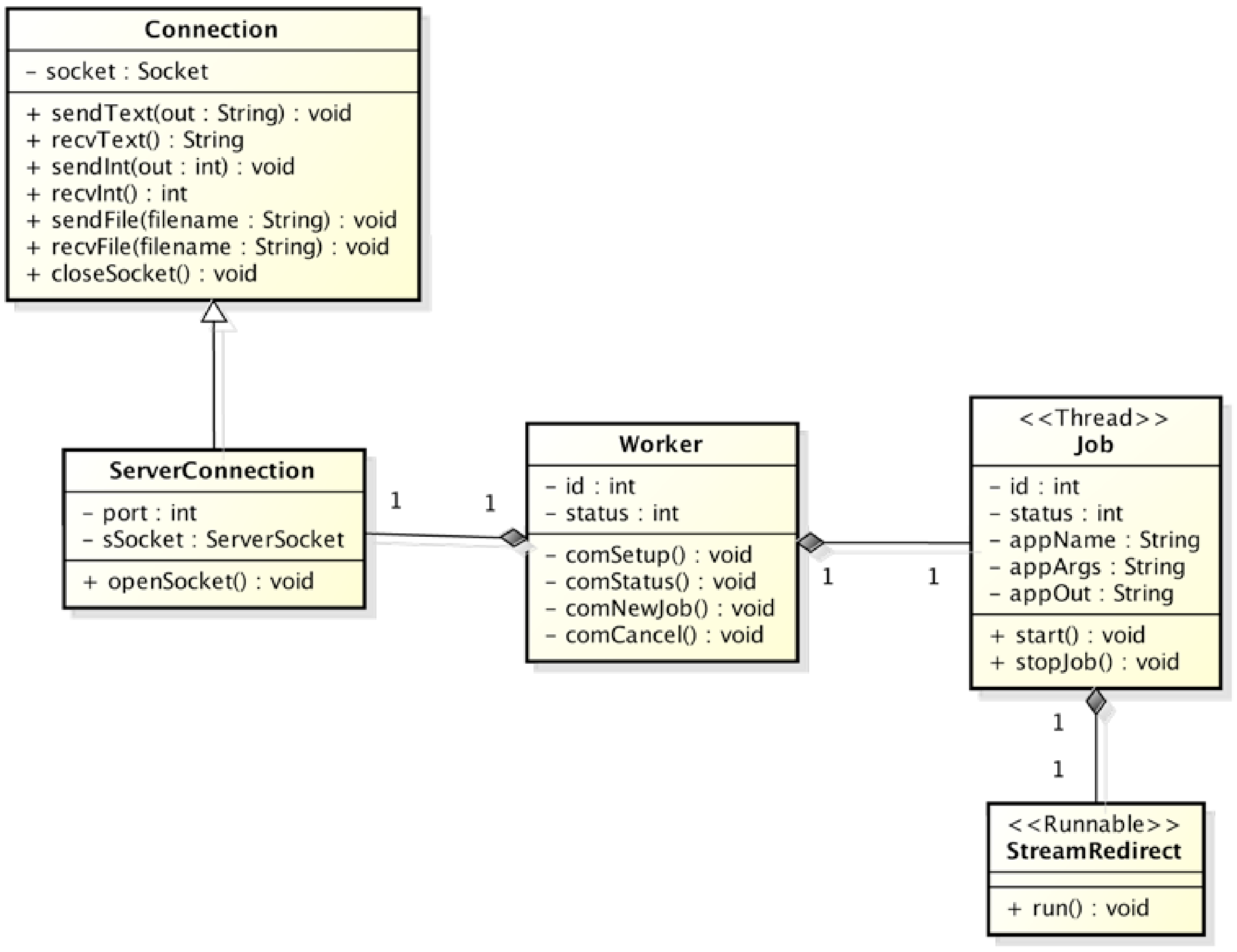
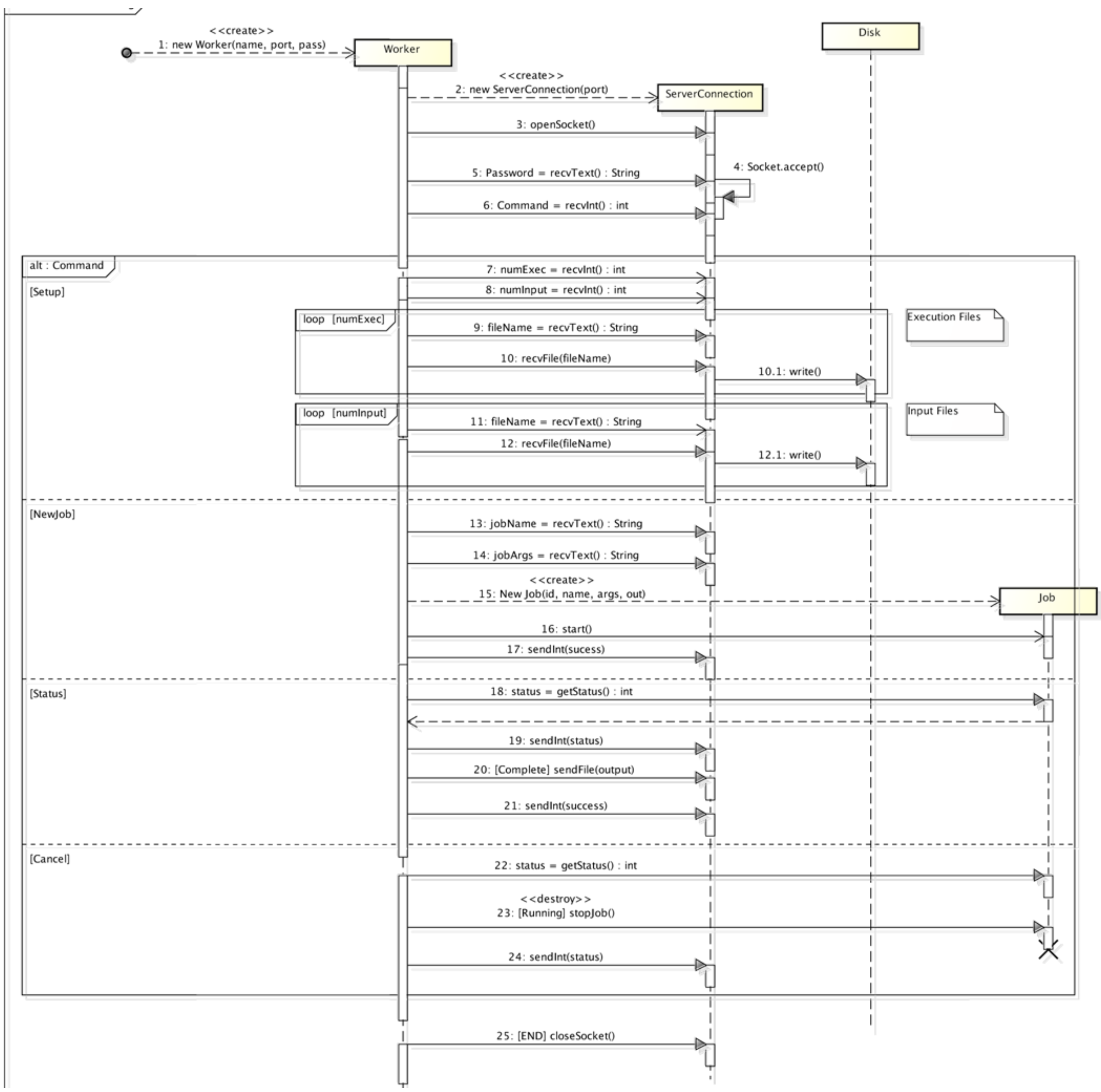


Figure 6:UML Class Diagram for the Worker program.



<<create>>
1: new Worker(name, port, pass)
Worker
Disk
<<create>>
2: new ServerConnection(port)
ServerConnection
3: openSocket()
4: Socket.accept()
5: Password = recvText() : String
6: Command = recvInt() : int
alt : Command
[Setup]
7: numExec = recvInt() : int
8: numInput = recvInt() : int
loop [numExec]
Execution Files
9: fileName = recvText() : String
10: recvFile(fileName)
10.1: write()
loop [numInput]
Input Files
11: fileName = recvText() : String
12: recvFile(fileName)
12.1: write()
[NewJob]
13: jobName = recvText() : String
14: jobArgs = recvText() : String
<<create>>
15: New Job(id, name, args, out)
Job
16: start()
17: sendInt(sucess)
[Status]
18: status = getStatus() : int
19: sendInt(status)
20: [Complete] sendFile(output)
21: sendInt(success)
[Cancel]
22: status = getStatus() : int
<<destroy>>
23: [Running] stopJob()
24: sendInt(status)
25: [END] closeSocket()

Figure 7: UML Sequence Diagram for the Worker program.