A Multi-platform Job Submission System for Epidemiological Simulation Models

Kunal R. Mudgal

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Madhav V. Marathe, Chair
Henning S. Mortveit
Keith R. Bisset

July 22, 2011
Blacksburg, Virginia

Keywords: Openmalaria, BOINC, Simulation job submission system

A Multi-platform Job Submission System for Epidemiological Simulation Models

Kunal R. Mudgal

(ABSTRACT)

In the current era of computing, the emergence of middle-ware software has resulted into a high level of software abstraction which has contributed to the diffusion of distributed computing. Distributed frameworks are systems comprised of many computing resources that are coupled together to perform one or more tasks. In a scientific grid-like environment, there are many different applications which require vast computing resources for simulation, execution, or analysis. Some of the applications require faster results, while other rely on more resources for execution. A configurable job submission system helps in distributing the tasks among the resources depending on their preference, priority and usage in a distributed environment.

We implemented a job submission system using JavaSpace which can be used for simulating jobs according to their priority. Two different types of brokers are used to monitor the JavaSpace. These brokers can either be used to compute results for tasks that require faster results using binaries installed on a stand-alone system or they can be used to submit the task to a BOINC system for computation. The flexibility to submit tasks as per their priority would help users to get faster results. Upon completion of the tasks, the brokers updates the jobs and transfers the results for further processing thus completing a cycle of retrieving a job, computing the data and transferring the results back to the user.

The generic nature of the framework makes it very simple to add new services which can perform a variety of tasks, making the system highly modular and easily extensible. Multiple brokers doing the same or different tasks can run on the same or different systems allowing the users to make efficient use of their resources. The brokers can be configured to detect the existing system and start monitoring jobs of different types. The framework can be used to transfer the results or detect any failures while execution and report it back to the user. The simple modular design and strong high-level JavaSpace API make it very easy to develop new systems for the framework.

Acknowledgments

I wish to thank my advisor Dr. Madhav Marathe for showing the confidence in me to work on this project and write my thesis. Without his encouragement and support this would not have been possible. I wish to thank Dr. Henning Mortveit and Dr. Keith Bisset for agreeing to be on thesis committee and their valuable inputs in this project. I will always be grateful for the assistanceship and help they have given to me, and the incredible kindness they have shown while I was working on the project and later when I was writing the thesis.

I want to thank Nicole Maire, Tomas Smith, and Diggory Hardy from Swiss Tropical and Public Health Institute (Swiss TPH) [27]. The OpenMalaria simulation model developed by the Swiss TPH have guided the design and implementation of several brokers in my thesis. The project was part of the work funded by the Swiss TPH and The Gates Foundation under grant no. 456334 for the project "A stochastic simulation platform for predicting the effects of different malaria intervention strategies".

I would like to thank the Computer Science Department at Virginia Tech for letting be a part of one of the best graduate school in the country. I have thoroughly enjoyed all the courses that I had taken during my graduate studies, all the participation in the seminars and all the interaction with the amazing faculty of the department. I am indebted to my department for all these wonderful things. I would also like to thank all the faculty and students in Virginia Bioinformatics Institute (VBI) for the enthusiastic participation. Working on my thesis at VBI and interacting with everyone here has been a truly fulfilling experience.

A special thanks to all my friends back in India and all the new friends that I made during my time here in Virginia Tech. They have definitely made my journey worthwhile. I want to thank them all for their hospitability and cohesiveness, you have always made me feel at home.

Finally I want to thank my parents and my sister for their continuous support throughout my studies. Without their love and belief in me, this all would have not been possible. I wish you extend my hearted gratitude towards my family and friends who have touched me with their patience and understanding during the course of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The spread of the Internet within the past decade and continual increase in the availability of faster and cheaper computer resources have changed the way that people manage information and information services. The low-cost hardware and the Internet have resulted into proliferation of peer-to-peer (P2P) [21], cluster [5], and distributed computing [14]. In traditional distributed computing, different resources are connected to form a cluster of resources. Today distributed computing is more complex than just a client and server communicating over the same protocol. In a truly distributed computing framework the resources are geographically distributed and accessed via an Internet or a dedicated network between the systems. These resources can be used to perform dedicated tasks such as computation, simulation or a mix of both along with variety of other tasks. Due to the coupling of machines over the Internet geographically distributed machines can now be used for solving large-scale, data-intensive problems very easily. Data can now be shared, computed, and aggregated using the Internet, which would help in solving many resource-intensive applications in the field of science, medicine, mathematics, and engineering. Today there are many projects on the internet which make use of the paradigm of distributed computing (i.e., BOINC [2], Seti@home [3], etc.) for computing vast amounts of data which require high computing power. These projects leverage the resources volunteered by users on the Internet to achieve a high amount of computing power.

A typical grid job submission framework would consist of four layers; namely the fabric, core middleware, user-level middleware, and applications [12]. Similar architecture can be used for

designing a job submission system for other types of resources. Figure 1.1 gives you a generic view of the layers in the distributed computing framework. A fabric consists of hardware resources, such as the computer itself; network; and other devices that are interconnected or managed by their resource management system. The core middleware consists of services which are used for storage, job submission, resource profiling, and their access management, along with their security. The user-level middleware consists of tools for executing the jobs and application development, along with the runtime environment. The information is passed from the core middleware to the user-level middleware via a communication system for computation. The last layer (i.e., applications) consists of libraries and binaries which are built using the user-level middleware tools for users to access.

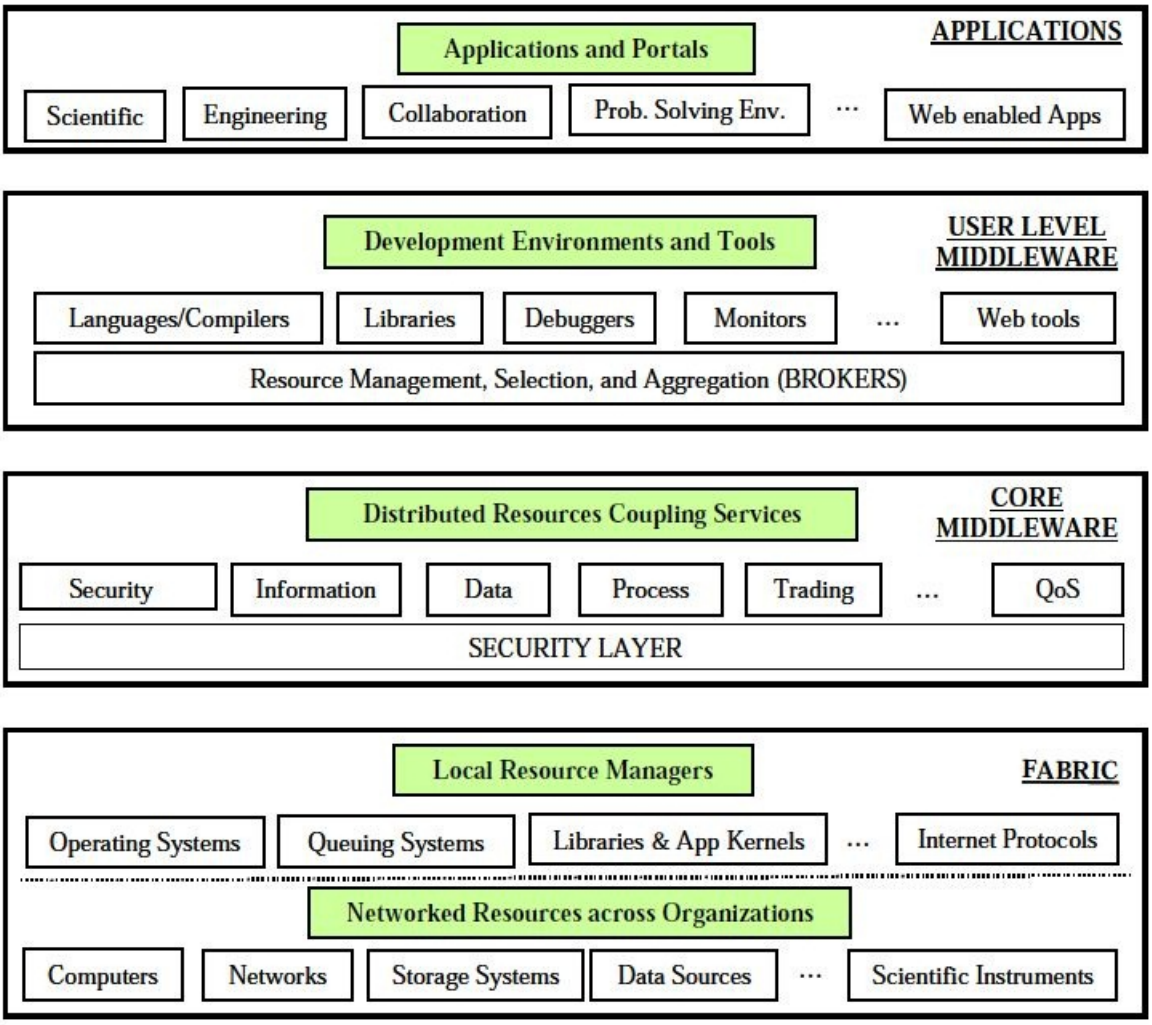

Figure 1.1: Layers of distributed framework [adapted from [10]]

The users interact with the user-level middleware layer through the applications. The tools in the user-level middleware layer communicate with the different layers within the framework, thus hiding the complexity of the framework from the users. A distributed job submission system can be thought of a system consisting of a producer (i.e., resources) and a consumer (i.e., end users). The main task of the job submission system is to enhance the performance of specific applications in a way that its end-user requirements are met. A consumer is responsible for creating a job which consists of data, instructions, and all of the information related to executing that job. The consumer then sends this information through the application layer to the back-end system for processing. The producer is the end resources which operate on the information that is transferred from the consumers.

A resource broker is a tool which communicates with the users and hides the complexity of the underlying distributed framework [11]. The broker can run on resources which the users can access to stage applications, process data, initiate job execution, and collect the results. It is developed to perform the task of job monitoring, task execution, and sending any information related to a job back to the system requesting the information. When the broker finds a job for execution, it reads in all of the data and instructions for executing the job and then performs the tasks by using the machines that are connected to the broker. Once the job execution is complete, it collects all of the results, notifies the job monitoring system, and transfers the results back to the user.

1.1 Motivation:

Over the past decade, the term distributed computing has evolved to encapsulate systems ranging from clusters, grids, desktops, and now even personal computers. As the notion of distributed computing has changed, the ability to efficiently connect such systems have got more importance. The idea of connecting systems varied in design and architecture have resulted into developing frameworks which can allow multiple systems to communicate with each other and perform collaborated or independent tasks. These frameworks help create a unified back-end system so that the application developer/designer do not have to worry about creating a new framework for each new type of application. There are many existing job submission systems that can be used to run a variety of applications [7, 15,

9]. Even though most of the systems are configurable, they are either designed to execute the applications independently on a separate machine or run a coordinated tasks between machines in a cluster [17, 9]. For tasks executing independently on a separate system, the job submission framework expects that the machine should have enough computational resources to run the application. If the application is designed to a run on a cluster, there is an extra overhead on the job submission framework to monitor the cluster regularly to check for any updates related to the task.

One of the challenging task in designing the job submission framework is the ability to run on different platforms and machine architectures. As newer system architectures are introduced and new technologies emerge, it becomes more difficult for the framework to manage the applications and the connected systems. JAVA as a programming language has emerged to solve this problem by executing all the applications on a separate JAVA Virtual Machine (JVM) and allowing the same applications to be executed on different platforms. Such a job submission framework developed using JAVA would ensure that all the systems communicate with the same uniform protocols and there is no need to manage multiple versions of the same framework for different platforms.

As a job submission framework is designed to handle multiple tasks, it becomes highly important to assign and execute applications which have higher priority over other applications in the framework. This is an important feature for a job submission framework because it would improve the overall efficiency of the framework if it can service higher priority jobs before other lower priority jobs.

One of the important feature of job submission system is the ability to plug different systems into the framework. The framework should not be limited only to desktop machines or clusters because newer technologies in distributed computing allow geographically separated system to communicate with each other and perform task execution. An example of such a distributed architecture is the BOINC framework for connecting volunteer computing [30]. If the job submission frameworks can handle a BOINC server the total computation power of the framework increases tremendously.

With all the above mentioned features and concepts, we planned to design a job submission system using JAVA programming language. This system can then be ported on any platform using JVM. The new job submission system can be designed to handle applications with

different priorities by executing higher priority tasks on dedicated systems ensuring faster results. The most important feature of the new job submission system would be the ability to execute tasks developed for volunteer computing. The framework would allow users to submit jobs to a BOINC server which would then carry out the execution over the internet and return the results back to the framework.

1.2 Contribution:

We have designed a job submission system which can schedule jobs for execution based on the priority and availability of resources in the system. Our job submission system consists of JavaSpace, which is a central repository for submitting jobs and exchanging related information, and two different types of brokers. The first type is a stand-alone broker that is used to execute jobs which require faster results. Based on the information posted on the JavaSpace by the users, the stand-alone broker retrieves the data, configurations, and information that is related to the job from the JavaSpace and runs the binaries installed on its own systems while passing the retrieved data. After the execution is complete, the users are notified by updating the information on the JavaSpace. The result data and related files are then sent back to the users, where they are transferred to a database and displayed for the user. By running the jobs locally, the stand-alone broker provides faster results for high-priority jobs. The stand-alone broker can be configured to monitor the current executing job and send real-time status updates for the job back to the user. Multiple systems running the same standalone brokers can be used together in the same framework to compute results for high-priority jobs which require immediate results.

The second type of broker that we developed in the job submission framework is used to execute jobs which are configured to run on a BOINC server. As BOINC distributes the jobs over the Internet and computes its results through resource volunteering, this broker is used to execute jobs which do not require immediate results. The tasks which require very high computational resources and take a long time for processing can be submitted to the BOINC system for calculation. The BOINC broker retrieves the BOINC job from the JavaSpace, reads in all the data, and writes all of the configuration files necessary for submitting the task to the BOINC server. Once the files are written, it calls the BOINC server and places

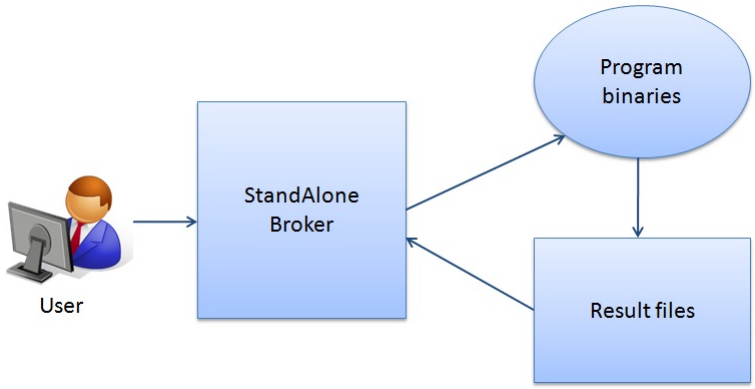

Figure 1.2: Standalone Broker

the job on its run queue for execution. It then continuously monitors the BOINC database to check for task status updates related to that particular job. It reports the status back to the user as to whether the task is in the running mode or has been completed. Once the computation is finished and the results are ready, they are then transferred to the database and displayed to the user.

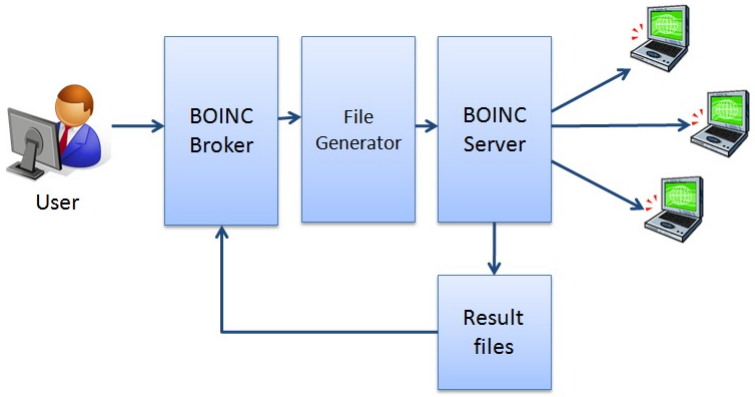

Figure 1.3: BOINC Broker

1.3 Outline:

In Chapter 2, the key technologies and concepts that will be referred in the later chapters are discussed. Chapter 3, explains the overall design of the system and important individual components. Chapter 4 will give more in-dept information about the system architecture and its implementation. The chapter covers the details of the two types of broker and also

discuss the libraries used in the implementation along with the configuration files used in the system. Chapter 5 provides a discussion of related work. Chapter 6 summarizes ther work and provide directions for future work.

Chapter 2

Background

The services within the job submission system communicate with each other through JavaSpaces. We discuss the complete model of JavaSpace and the data object used in transferring information between services in section 2.1. The job submission was designed to submit Openmalaria simulation jobs. Section 2.2 gives an overview of Openmalaria. BOINC is used to submit Openmalaria jobs which needs high computational resources. We discuss the BOINC architecture and its database schema in section 2.3 and section 2.4.

2.1 JavaSpace:

Writing an application for a distributed system introduces many network complexities which are often ignored on applications that run on a single machine. Perhaps the most obvious complexities are the different machine architectures and the software platforms over which the applications must commonly execute. The growth of distributed applications were stalled by the problem of developing an application and then porting it to every platform on which it could run and also the distribution of this platform-specific code to each machine in the distributed network. There are other issues, such as latency, synchronization, and partial failure, which also have a significant impact on designing and implementing distributed applications. Technologies such as low-level sockets, message passing, and remote method invocations [31] were previously used to design distributed application and the programmer needed to address each networking issue carefully in order to develop an efficient application.

The Java technology is platform independent and its use of virtual machines for running applications make the code write once and run anywhere. JavaSpace is a completely different programming model in which the application is viewed as a collection of processes that coordinate with each other via the flow of objects into and out of spaces. JavaSpace can be viewed as a shared, network accessible repository for objects. JavaSpace is different from other message-passing technologies in the sense that it provides persistent object exchange areas or spaces through which other Java processes coordinate actions and exchange data. The advantage of using this approach is that it simplifies the design and implementation of sophisticated distributed applications.

The JavaSpaces technology does not require a very complex programming interface. It consists of very simple operations. These small sets of operations can be used to build a large, complex class of distributed applications with very few lines of codes. JavaSpace uncouples the senders and receivers which helps in the composition of large applications in which one can easily add new components without redesigning the entire application, seperately examine local computation and remote procedures, and also easily replace existing components. It also handles the problem of concurrent access and persistent storage and transactions, due to the fact that it becomes very easy to write client/server applications. The most important factor of using the JavaSpaces technology is that it is quite powerful and very easy to understand. By using JavaSpaces, one is able to reduce the coding and debugging time which can result in applications that are more robust, conveniently maintained, and easier to integrate in existing applications.

As you can see in the Figure 2.1, a client can interact with multiple JavaSpaces. The server and the client use the JavaSpaces as a repository for persistent object storage and exchange mechanism instead of for direct communication; they coordinate by exchanging objects through spaces. Objects on the JavaSpaces consist of passive data so they cannot be modified directly. In order to operate on the methods and variables in the object, the object itself has to be removed explicitly, changed, or operated upon and then should be placed back on the JavaSpaces. Processes that interact with each other in JavaSpace use the following simple set of operations:

- write(): This function write a new object in the JavaSpace
- take(): This function fetches the object and removes it from the JavaSpace

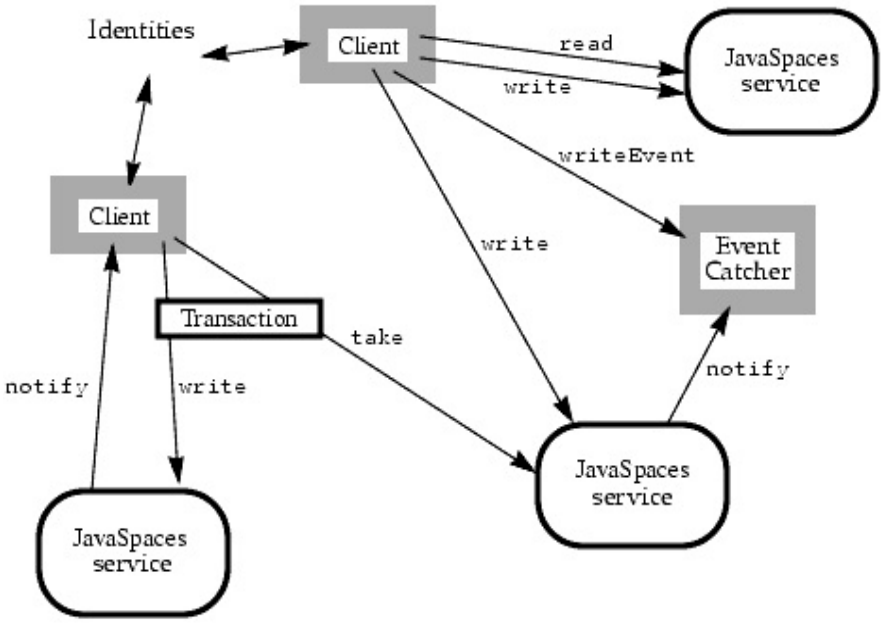

Figure 2.1: JavaSpaces (adapted from [20])

- read(): This function reads the object from the JavaSpace
- notify(): This function notifies a specified object when an entry matching its template is written in the JavaSpace

JavaSpace provides a transactional model which ensures that an operation on the space is atomic. Either all or none of the modifications for the transaction are applied. The transactions can span from one or more operations or spaces. Since the space which shares the objects provides a coordination mechanism, the programmer does not have to worry about scenarios such as synchronization issues, network protocol challenges, or other multithreaded programming situations. Writing a client server application with JavaSpace is very simple.

The Listing 2.2 is a Java code for an application in which one system reads the JavaSpace for updates and the other system updates the entries [16]. The Java code listed in 2.1 is a simple implementation of a message class which will be read by the clients.

```
1 public class Message implements Entry {
2   public String content;
3   public Integer counter;
4
```

```
5    public Message() {
6    }
7
8    public Message( String content, int initVal) {
9     this.content = content;
10    counter = new Integer(initVal);
11   }
12
13   public String toString() {
14     return content + " read " + counter + " times.";
15   }
16
17   public void increment() {
18     counter = new Integer(counter.intValue() + 1);
19   }
20 }
```

Listing 2.1: Base message class (adapted from [16])

The class Message is a general class with contains a message and a counter to check th number of times that the class has been accessed. We have an increment() method to increase the counter and a toString() method to print the counter.

```
1  public class HelloWorld {
2     public static void main(String[] args) {
3       try {
4         Message msg = new Message("Hello World", 0);
5         JavaSpace space = SpaceAccessor.getSpace();
6         space.write(msg, null, Lease.FOREVER);
7
8         Message template = new Message();
9         for (;;) {
10          Message result = (Message)space.read(template, null,
11                                Long.MAX_VALUE);
12          System.out.println(result);
13          Thread.sleep(1000);
14        }
15      } catch (Exception e) {
16        e.printStackTrace();
17      }
```

```
18      }
19    }
```

Listing 2.2: Code to add a new entry and monitor it on JavaSpace (adapted from [16])

The HelloWorld class in the Figure 2.2 reads from the JavaSpace to see how many times the Message entry has been read by other processes. It starts off by initializing the Message object and then, after searching for a JavaSpace it writes the Message object to that space. It then continuously reads the space to see whether other processes access the Message entry. If we run this code, we get a continuous output of counter zero because there is no client to modify the object yet.

```
1   public class HelloWorldClient {
2   public static void main(String[] args) {
3      try {
4       JavaSpace space = SpaceAccessor.getSpace();
5
6       Message template = new Message();
7       for (;;) {
8         Message result = (Message)space.take(template,
9           null, Long.MAX_VALUE);
10        result.increment();
11        space.write(result, null, Lease.FOREVER);
12        Thread.sleep(1000);
13      }
14    } catch (Exception e) {
15      e.printStackTrace();
16      }
17    }
18  }
```

Listing 2.3: JavaSpace client code (adapted from [16])

The HelloWorldClient shown in 2.3 reads the message entry from the space and updates its counter. The client creates a template Message object and looks up the space to find an object that matches the template. Since we want to update the entry, we will retrieve the message entry from the space, update the Message object (i.e., call the increment() method on the object) and write the modified entry back to the space. When we start running the

client, the output from the HelloWorld class would then be noted as follows:

Hello World read 0 times.
Hello World read 1 times.
Hello World read 5 times.

Even though the example is simple, it helps to understand most of the important concepts in JavaSpace. Since the system is asynchronous, the client accesses the message entry on the space in unpredictable manner. The code is very simple and expressive; just 4 lines of code are needed for the JavaSpace operations. The code is loosely coupled because the HelloWorld class can update the entry without knowing anything about the clients (i.e., how many there are), or when they are added. Similarly the HelloWorldClient does not care about who added the entry to the space. Since entries in JavaSpace are persistent, the client can access it much later after the entry is added to the blackboard. This feature is useful when we develop a chat client that can read all of the previous messages sent by a sender to the client (e.g., offline messages in Gtalk or Yahoo). Since JavaSpace provides a high-level coordination mechanism, the programmer does not need to worry about multithreaded servers or synchronization issues. They simply need to focus on creating a data structure to be shared and distributed through spaces. It ensures that only one client has exclusive access to the entry at any given time. Thus, in comparison to other distributed programming tools, JavaSpaces will ease design and reduce coding and debugging time.

2.2 Openmalaria

Openmalaria is a software program that is designed to model individual-based stochastic simulations of malaria epidemiology, health systems, and malaria interventions. Mathematical models have been traditionally used to study the behavior and spread of diseases. Over the last few decades, the work of Ross, MacDonald and Struchiner et.al [23, 18, 26] have shown how the mathematical model can be used to study malaria transmission. More recent predictive models use effectiveness of intervention, studies on disease burden, and the costs of morbidity and mortality [24]. The complexity of the malaria models and limitation of computing resources for calculation limit the applicability of a conventional deterministic and stochastic malaria model. Openmalaria provides an individual-based stochastic simula-

tion of malaria epidemiology to predict the impact of intervention on infection, morbidity, mortality, health services use, and costs [24]. These stochastic models are designed in C++ programming language and Fortran 95 and they are quite computationally intensive. The definitions and specification for the model are defined in a markup file (XML) and the software tool generates a plain text results file that contain predictions of age and time-specific epidemiological quantities. For complex simulations which involve multiple scenarios and objectives, the models are sent to a BOINC system to compute results in parallel for each scenario and objective which helps simultaneous multi-objective optimization [24]. Even with the availability of high computing resources, each optimization takes several weeks because it is limited by server capacity, the capacity of the research team to process the results and the turnaround time on the BOINC workunits.

2.3 Berkeley Open Infrastructure for Network Computing (BOINC):

The biggest pool of computing resources are the desktop, laptop , tablet and other devices that are owned by consumers all over the world. These computing resources are much bigger than the existing distributed computing systems such as clusters and grids. Volunteer computing is a mechanism by which consumer resources can be used for computational research projects. The difference between clusters and volunteer computing is cost and ownership. Volunteer computing resources are free to use but the owner does not get exclusive access to the resources as in clusters. The compute jobs run in the background and the data files are stored on the local disks in volunteer computing [8].

The Berkeley Open Infrastructure for Network Computing (BOINC) is a non-commercial middleware system for volunteer and grid computing [30]. It was developed by the same software team which developed the SETI@home [3] project when they realized that they had more computing resources available than they required for computation. BOINC provides server software that lets researchers create volunteer computing projects which need high computational power and client software that lets volunteers participate in any combination of these projects. BOINC is currently being used by 20 different projects, including 600,000 volunteers and 1,000,000 computers which is equivalent to 350 TeraFlops of process-

ing power [1]. BOINC follows the client-server paradigm in which each project has a server which manages all the communication and work distribution to the clients. Clients on the other side download all of the application files and software, compute the results, and send the results back to the server. Like SETI@home, BOINC users are awarded with credits when they volunteer their computer to do work for the project. BOINC software includes server side components such as the scheduler, programs which manage the distribution of task and data collections [4] and also a Web-based interface for clients and project administrators. BOINC provides the following features, which makes it a widely used middleware system for volunteer computing.

- Security
- Multiple servers and fault tolerance
- System monitoring tools
- Support for diverse applications
- Rewards participants by using a credit system
- Tunes work distribution based on host parameters
- Locality scheduling
- Open and extensible architecture.

A complete BOINC project consists of a server and client components as shown in Figure 2.3. All BOINC applications are linked with a runtime system whose functions include checkpoint control, process management and graphic displays [1]. Since the BOINC application runs at zero priority, it can be preempted by any other application when needed. The BOINC framework is very robust and it provides support for redundant computing by rejecting erroneous results and is fault tolerant because it uses and exponential backoff scheme.

The BOINC client initiates the communication by sending a request over HTTP to the server. The request is an XML file which describes the information about the hardware and the availability of the machine on which it is running. After the client establishes the communication, the server sends a reply consisting of an XML that describe the new

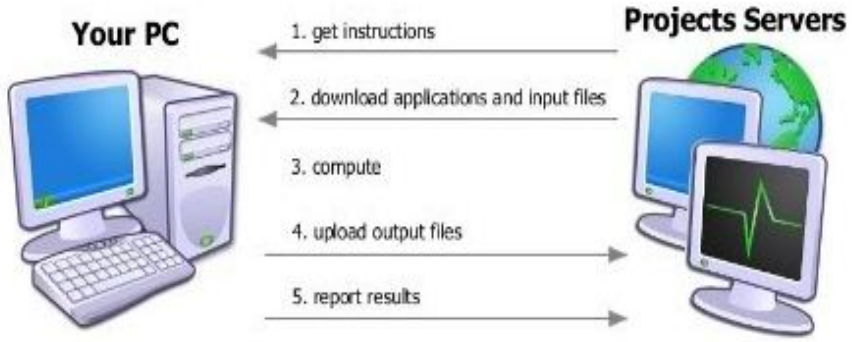

Figure 2.2: Typical BOINC operations (adapted from [32])

tasks and files related to the application. The client then downloads all of the binaries and files necessary for running the application and, after completion, uploads the results file. Figure 2.2 shows the steps in communication between a client and a BOINC server.

BOINC provides tools for creating new applications on any platform, creating workunits, monitoring server performance, and controlling projects. Since BOINC projects are mostly designed by scientists, they are very well documented; hence, complete projects can be created in a few hours [2]. Figure 2.3 gives you the overall architecture of BOINC. The details of the client server attributes are explained in the following sections.

2.3.1 Client:

Users who wish to contribute to the BOINC project, download and install the BOINC client software on their machine. The client software is developed for Windows, Mac and linux operating systems. In order to attach the client to a project, the user must obtain an unique identification number (ID) by registering themselves on the project Web site. Once the user enters the user ID and the project URL, it becomes attached to that particular project. A single user can attach to as many projects as is desired. The user's computer cycle would then be split between these different projects. The client software performs CPU scheduling of jobs between the projects and maximizes concurrency by using multiple CPUs when possible and overlapping both communication and computation. The user has a significant amount of control over the client. Using the preferences in the client software, a user can change the amount of work they want their computer to do for the project. Once the computation is

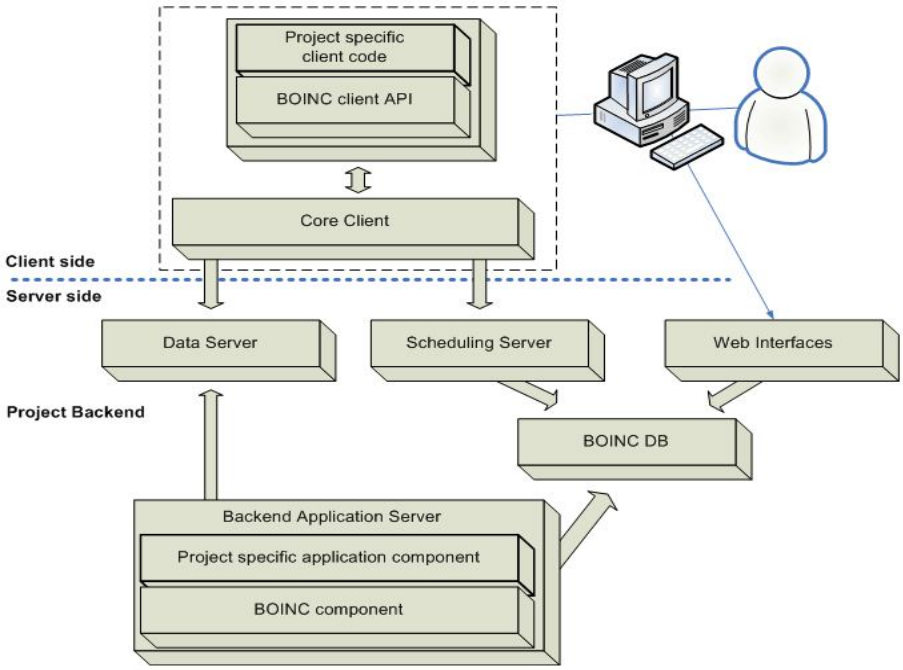

Figure 2.3: Architecture of BOINC (adapted from [])

completed, the client software notified the server during the next scheduling that the results are ready for transfer. The tasks are scheduled on a FIFO basis on the client, but if there are multiple projects, then an internal scheduler will distribute the work cycle equally amongst the jobs. The software also reports the amount of time spent on each project. The users are given credits for volunteering their computer which can redeemed to buy things online.

Security is one of the most important issues that arises while using the client software. All of the project related data and programs is hashed and signed by a private key. This ensures that the code the users are getting is from the right project and has not been tampered with a malicious code. The client software also provides solutions to a lot of project security-related issues where in the user can try to return wrong results or try to get more credit than the work they have done. This is achieved by comparing or distributing the same work to atleast 5 people and then comparing the results of each in order to allocate credits only to users who have the right results. There are other features, such as the ability to limit the size of the output file that will be uploaded to the server and distributing the credits equally between users to avoid other project related security problems.

2.3.2 Server

Every BOINC server consist of a Web server which handles the connection between the users and the project. It is also responsible for issuing work and collecting results from the clients, keeping a track of all the workunits and their associated results. Apart from the Web service there are other daemons which are designed to perform specific tasks and update the database with the information and results. These individual daemons can run on the same computer or be distributed as well. The server handles all of the remote procedure calls (RPC) with the clients. The daemons that run on the server include the following:

- Transitioner: The transitioner implements the redundant computing logic and handles the state transition of work units and results. The transitioner "moves" the results between phases as the results change state. It keeps a track of the results in progress and make sure that they move properly down the pipeline.
- Validater: The validator examines a set of results and selects those that are canonical. It is also responsible for granting credits to the clients.
- Assimilator: The assimilator daemon parses the results and puts them into the correct project database.
- File delete: This daemon removes the workunit and results data files when they are no longer needed for the project. This task helps to keep disks as clean as possible.
- Feeder: The feeder is responsible for loading undispatched results from the database into a shared memory in order to improve the performance of the BOINC system. The scheduler would have faster access to data through the shared memory segment and does not need to query the database for each Workunit.
- Scheduler: The scheduler programs run when a client connects to the system and requests WorkUnits.

2.4 BOINC Database

All of the information that are related to the BOINC projects are stored in the database. BOINC uses a MySQL database for storage. The BOINC database is based on the relational database model. This database is a collection of tables and indexes that hold information about workunit's, results, users, and almost all of the information about a specific project. BOINC provides the flexibility to run the database on a machine that is similar to one's Web server as the project daemons operate on it or a separate system. The framework even allows the user to miigrate the database from one system to another when required. The database schema for BOINC is very complicated and 2.1 offers a high-level view of the table structure.

Table 2.1: BOINC Database schema (adapted from [13])

Table Name	Description
app	This table stores information about Applications.
app_version	Contains different version of applications and includes a URL for executables and the MD5 Checksum of the executables.
host	Contains information about the clients computer connected to the project.
result	This tables gives the information about the state of the workunit. It also has other information about the results and exit status for the job.
workunit	Describes information about workunit including results and errors or failures in execution of the workunit.
user	Descibes the users name, password, email address and Account key.
platform	This tables consists information of the compilation platform of the target CPU and Operating System.

The workunit tables handle most of the information related to the tasks assigned to the system. The states of the workunits can be described in terms of the state variables in this table. This state variable can be monitored to find out whether the specific job in the system is running or completed. The results table gives even more details about the state of the jobs, but this information can be used to predict the estimated time of availability

for a batch as it contains details about when the workunit replicates will time out [13]. The workunit table consists of following important state variables:

id: The workunit ID of the job

file_delete_state: Indicates whether or not the input file should be deleted

assimilate_state: Indicates whether or not the workunit has completed its execution

error_mask: Indicates a bit mask for the error conditions

There are other variables in the table, such as transition_time, canonical_resultid, etc., which provide more information about the workunits. The assimilate_state variable can be used to monitor the execution state of the workunit. The validator sets the variable when the canonical results for the corresponding workunit are available or if there was an error condition during execution [25]. The different states of the assimilate_state variable include

- 0 = INIT
- 1 = Set to READY by transitioner if wu.assimilate_state = INIT and WU has error condition
- 1 = Set to READY by validator when it finds a canonical result and wu.assimilate_state = INIT
- 2 = Set to DONE by assimilator when execution finished

Changing the assimilate_state's value to DONE (2) indicates the WorkUnit has finished computation with or without errors and signifies that the result files are ready for collection.

Chapter 3

Design

The complete middleware of job submission system consists of many individual components such as JavaSpace, Brokers, Runners, etc. All of the components are either designed or extended from an existing infrastructure which is used to submit jobs to a batch queuing system. In this chapter, we will discuss the overall middleware architecture of the system in section 3.1. The main components of the framework, the broker and runner, are described in section 3.2 and 3.3, respectively.

3.1 Overall Architecture:

The Figure 3.1 shows the overall architecture of the middleware system that is primarily comprised of brokers, blackboards and runners. Several services are created that are available to process requests from the blackboard which is currently implemented as a JavaSpace. These services are nothing but the brokers that need to be executed on different systems. Some of the services that are currently implemented in this system are:

- Simulation Broker – BOINC broker and standalone broker
- Execution Broker – BOINC execution requests

The system can be extended to include other services (e.g., the data transfer broker to transfer results from one system to another or an analysis broker to perform a review of the

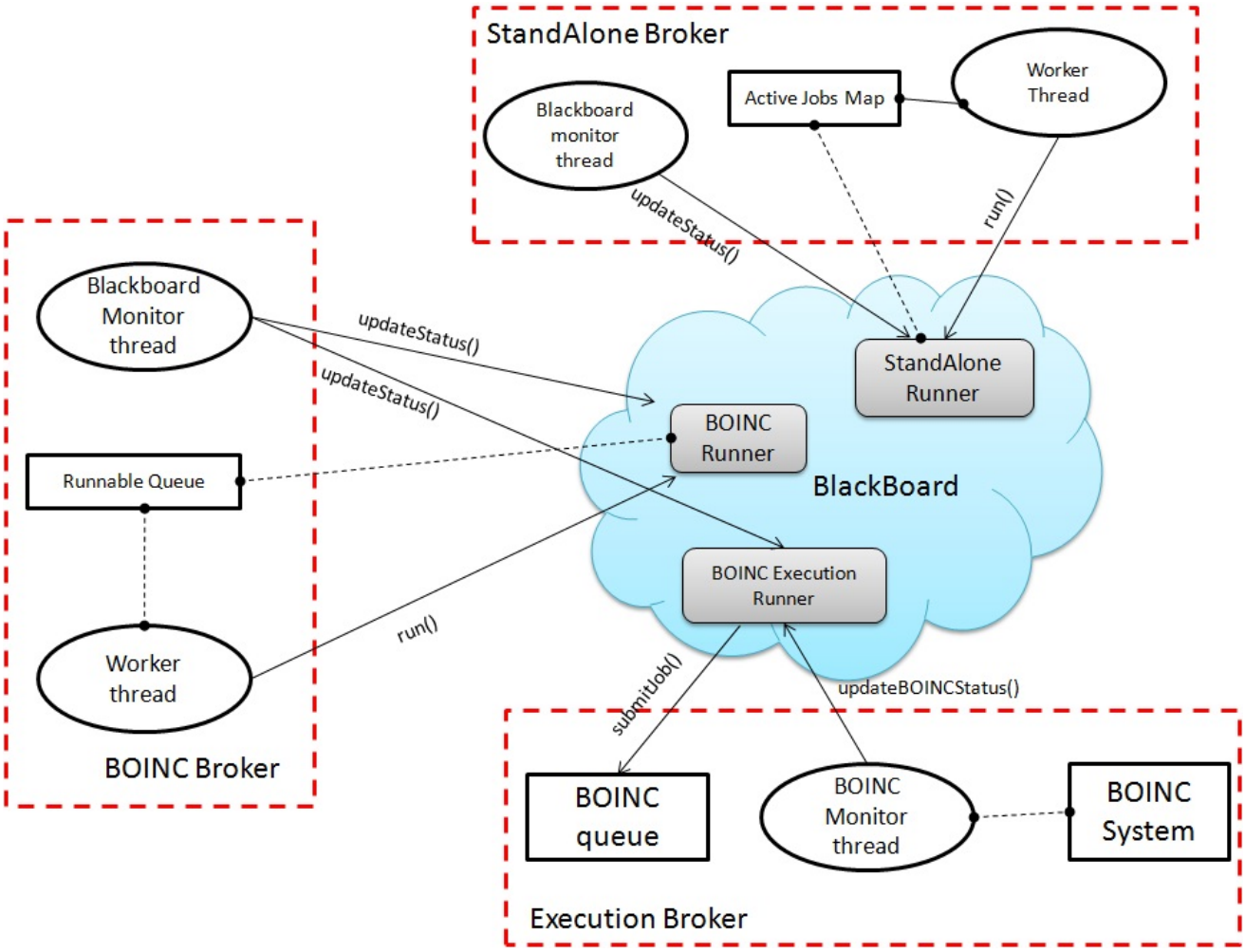

Figure 3.1: Middleware Architecture of broker, blackboard and runners

results obtained from different systems). Every service in the framework runs on a separate Java Virtual Machine (JVM). All of the services communicate with the frontend or with each other only through the blackboard. The jobs to be executed are placed on the blackboard as JAVA objects. The brokers are designed to identify the job requests and correctly execute them. All of the services in the system are configurable in terms of the number of threads used to monitor and the type of jobs executed to achieve optimum performance. Since each of the services run on a separate JVM, either some or all of the services can be started from the same machine. In addition, there is an option to start each service from different machines and they can connect to the blackboard through the JavaSpaces API. The complete

system is extensible, so it is possible to run many different services as is desired and they can connect to the same or different blackboard (i.e JavaSpace).

A blackboard entry object will consists of a runners which implement the SimfraservRunner interface. A thread in the broker monitors the blackboard(BlackboardMonitor class) to find new entries of definite types. When it finds a new entry, it is read by the monitor thread and the following sets of actions are taken:

- The SimfraservRunner object is retrieved from the entry
- The runner logger is set to the brokers logger object
- A pointer to the broker is placed in the runner object
- The status of the entry is updated to starting on the blackboard
- The entry is added to the brokers active request map
- The runner object is added to the runnable queue of the broker

The flowchart in Figure 3.2 shows the sequence of operation while processing a new request in the BlackboardMonitor thread.

Worker threads within the broker continuously monitor the active request map to find any new entries for execution. When they find a new entry, the worker thread takes the following actions:

- Sets the running flag in the SimfraserveRunner to indicate that processing has begun
- Removes the runner object from the queue
- Calls the run method of the object and waits until its execution completes
- Once execution is completed, it sets the running flag to false to indicate the end of processing

The runner maintains the state information while executing the run method. Depending on the status of the job on the blackboard, the runner makes subsequent calls to the run method

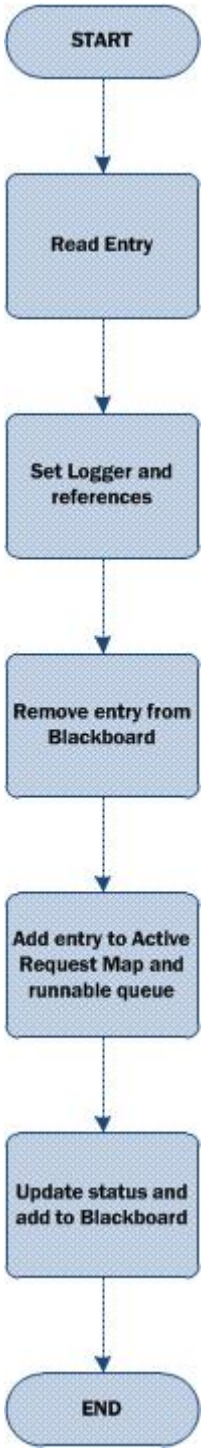

Figure 3.2: Flowchart for operation on a new entry in BlackMonitor thread

in the runner object. The entry is updated by the broker or by other execution requests on the blackboard.

Talk about simfraserv runner classes and RunManager

3.2 Brokers:

A broker is used to manage the workflow for the broker-specific requests from the blackboard. The blackboard in the system is a JavaSpace which is used for communication between the systems via Java objects. The broker continuously communicates with the blackboard to find any new objects that are added to the blackboard (i.e., one dedicated thread in the broker system continuously communicates with the blackboard to check new requests that are placed on it). The broker has a queue for the SimfraservRunner [] objects, so when a new job request is placed on the blackboard, the broker can retrieve that object from blackboard and place it in this queue for execution. The broker also maintains a map of active requests of the SimfraserveRunner objects that it is executing. Every broker maintains a pool of worker threads so that multiple execution requests can be handled by the broker simultaneously. The number of worker threads that a broker can maintain is configurable. The brokers implements the RunManager interface for the basic behavior (e.g., communication and job retrieval), but new methods can be easily added to alter the behavior of the broker.

As mentioned earlier, one thread in the broker monitors the blackboard for new requests. The type of requests that a broker monitors can be changed and so the same broker can be used to monitor multiple types of job requests on the same system. When a new request arrives on the blackboard, the blackboard monitor thread checks the type of the request and retrieves it from the blackboard if it matches the type of request that the broker is designed to handle. The broker then creates a new entry in the active request map for the runner, places an object containing the local property information and a reference to itself (RunManager) in the runner, and then adds the runner object to the runnable queue. Worker threads in the broker monitor the runnable queue to find any new requests to execute. When a new entry is added, the worker threads remove the runner object from the runnable queue and executes them. For execution, the worker thread calls the run() method of the runners and waits for the execution to be complete. After the execution is completed, the status of the

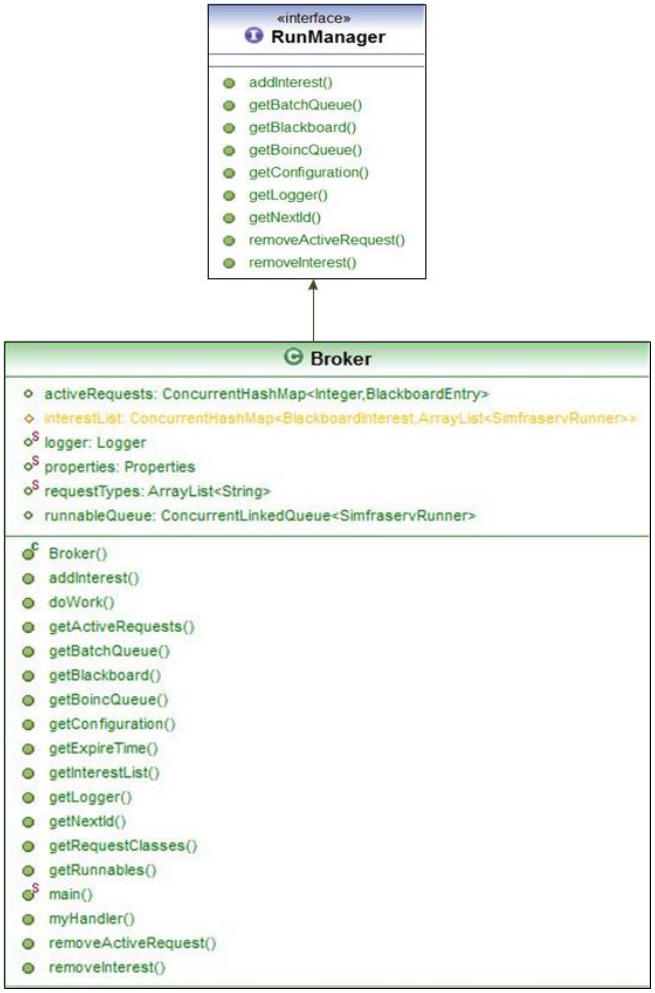

Figure 3.3: Broker class UML diagram

job is updated in the active requests map to indicate that it has completed execution.

When the blackboard monitor thread finds an entry on the blackboard which is already present in the active request map of the broker, it checks the status of the entry on the blackboard and updates the entry in the active request map to indicate the new status. The broker then calls the runner's updateStatus() method to reflect the new change in status for the runner object in the entry.

When the broker is started, it reads all of the information from a configuration file. Some of the configurable options are the number of worker threads, logging properties for the broker, sleeptime for the worker, blackboard monitor thread and the type of requests to monitor, etc. As shown in the class diagram, the broker creates the blackboard monitor thread and the

worker threads. The BlackboardMonitor class updates the status of the entry to indicate the start of processing. It is also responsible for setting the run manager in the request, adding or updating the requests on the activerequest map, and placing the runner object in the runnable queue for execution. Similarly, the worker class polls the runnable queue for new runner objects and calls the runners run method. Once the run method is called, the thread will wait for the runner to complete its execution. It also handles any exceptions that are thrown by the runner.

3.3 Runners:

A runner class contains the information necessary to initialize a program or series of programs to execute a job request. The execution request is processed by the job execution service to take the steps necessary to run a program on a system. The runner implements the SimfraserveRunner interface. The SimfraservRunner class handles the communication of the runner with the broker. The runner class is called by the worker thread in the broker. When a new object is added to the runnable queue, the worker thread retrieves the object and calls the run() method in that object. The main goal of a runner class is to simplify the process of adding a new simulation or analysis to the system by implementing a simple run function which would handle the complete execution of the program. Runners are instantiated by a specific runner factory and enclosed within a job request for a broker. Examples of runners include the standalone runner, BOINC runner and BOINC execution runner for executing openmalaria tasks.

The SimfraservRunner interface defines the basic methods that the runner classes must implement. The runner class, which implements the interface, contains all of the data structures that are common to all of the classes that are implementing the runner and which are expected to function in the system architecture. The SimfraservRunner allows the runner classes to receive asynchronous status updates from the entries on the blackboard via the blackboard monitor thread by calling the updatestatus method. It also allows the runner to take actions based on the updates from the run method. The interface contains methods for batch and BOINC job submissions, which would ensure that the runner receives and handles updates from these two different execution requests. Since the worker thread in the broker

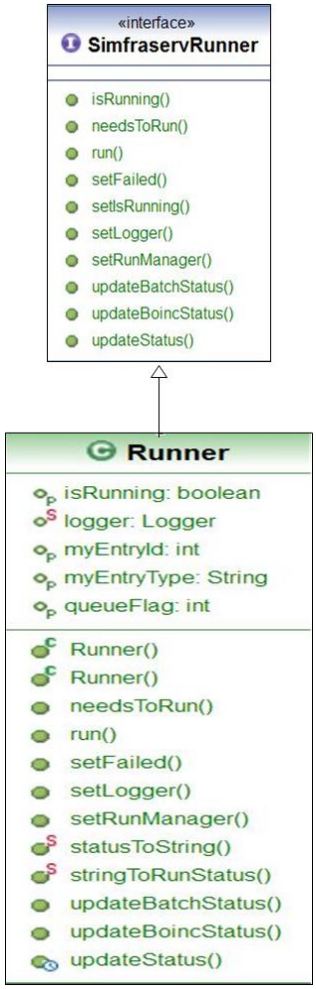

Figure 3.4: Runner class UML diagram

calls the runner, the simfraservRunner interface enables it to be connected to the broker log. The runner can even change the properties of the broker at runtime to adjust to the current execution request and update the status of the job.

The base runner class, which implements the SimfraservRunner, contains information to create and maintain a thread-safe data structure for ordered status updates of the entries on the blackboard. The runner contains the isRunning flag, which helps the broker to know whether the current request is executing or has completed execution. The queueFlag is monitored by the broker to determine whether the current execution request should be placed back in the running queue. The queueFlag is primarily set when there is a status update for the current execution, which directs the broker to put the request back in the running queue

for execution with the new status. The runner class also has the definitions for the logger, properties, runners blackboard entry, ID, and the broker object. It implements some of the important methods, such as the updateBlackboardStatus which updates the status of the runner entry on the blackboard. The registerInterest and removeInterest functions are used to modify the broker properties to search for specific types of entries on the blackboard. The Finish method is called by the runner when all of the tasks have completed their execution. The method decrements the reference count for the runners blackboard entry and removes the runner's blackboard entry from the broker's active requests.

The incoming status updates to a runner are stored in a queue in the runner class. It is the responsibility of the runner to process the status before calling the run method, so that proper action is taken in the run method depending on the current state of the request. Each runner is also responsible for maintaining its state information between the invocation of the run method; updating the status of the blackboard entry, depending on its current state; and calling the finish method of the runner class when the processing is complete.

Chapter 4

Implementation

The implementation of the complete job submission system is based on the extension of the broker and runners that are described in the section 3. This chapter will give details about how the overall framework is implemented. The section 4.1 gives you an overview of the front-end interface that the users employ to submit jobs and back-end operations in the framework. Section 4.2 will describe the implementation of the standalone broker and its main components. The BOINC broker consists of two separate but interdependent brokers. The architecture and implementation of the BOINC broker and the execution broker is described in section 4.3 and section 4.4.

4.1 Overview:

The job submission system allows users to submit jobs which execute on different systems connected to the framework. Jobs with different types and priorities can be submitted to the system for execution. This framework is designed to handle Openmalaria jobs which can run on either standalone binaries or submit the jobs to a BOINC server for execution. The front-end system allow users to create or modify the Openmalaria experiments and studies. Figure 4.1 shows the interface that the users employ in order to submit the Openmalaria jobs. The front-end interface for the Web application is still under development and can change over the course of time, but the general functionality would still remain the same. After logging into the Web application, users can create new scenarios or load an existing

scenario for execution. All of the new jobs submitted by the user are sent to the JavaSpace for execution. Users can check the result of the jobs that have completed execution or monitor a running Openmalaria job. The clean tabular interface allows the users to switch between different operations in the application. Each tab on the top of the Web page have a specific task as listed below:

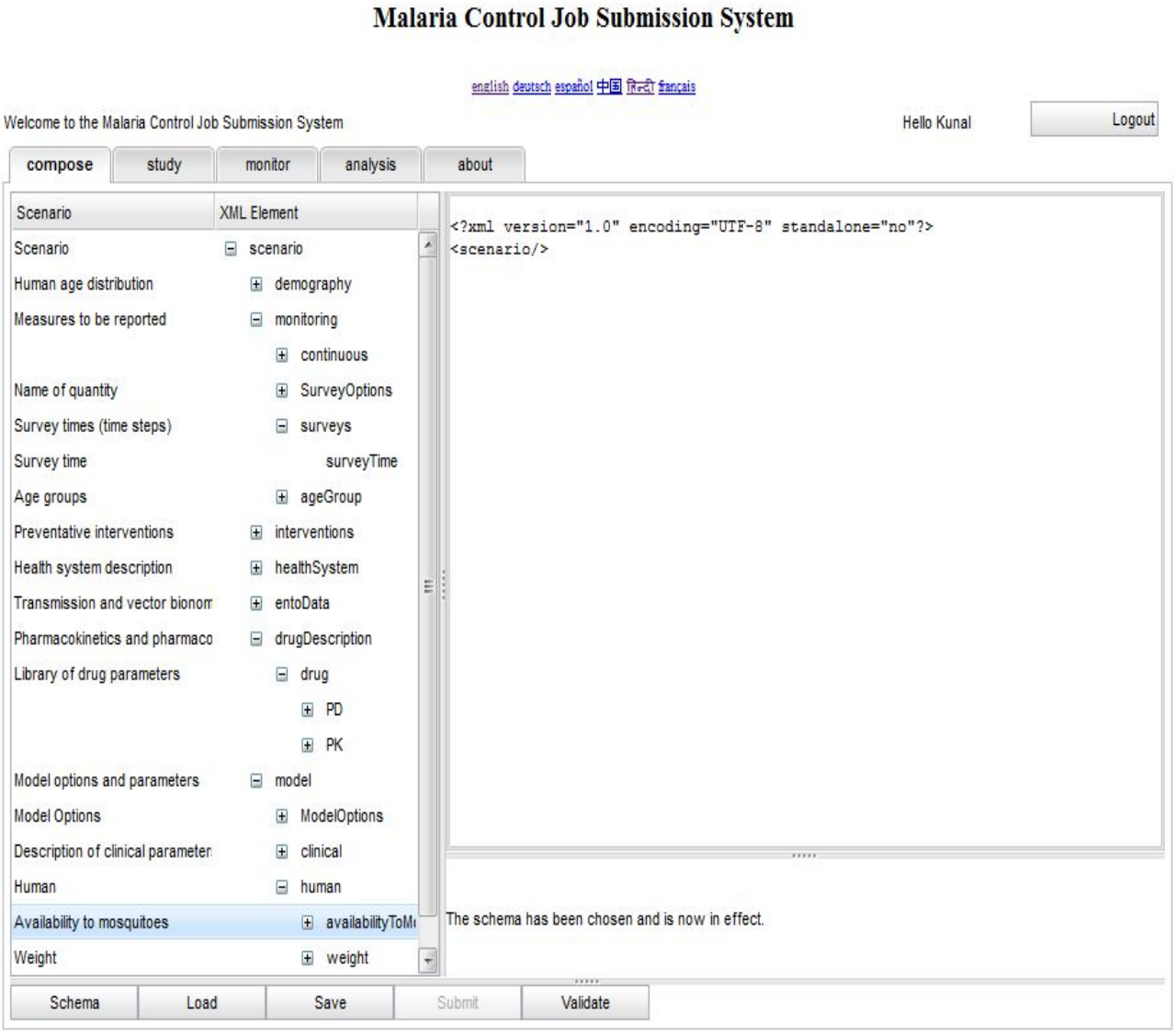

Figure 4.1: Malaria control frontend

- Compose: This tab allows for the creation of a new scenario or lets an existing scenario

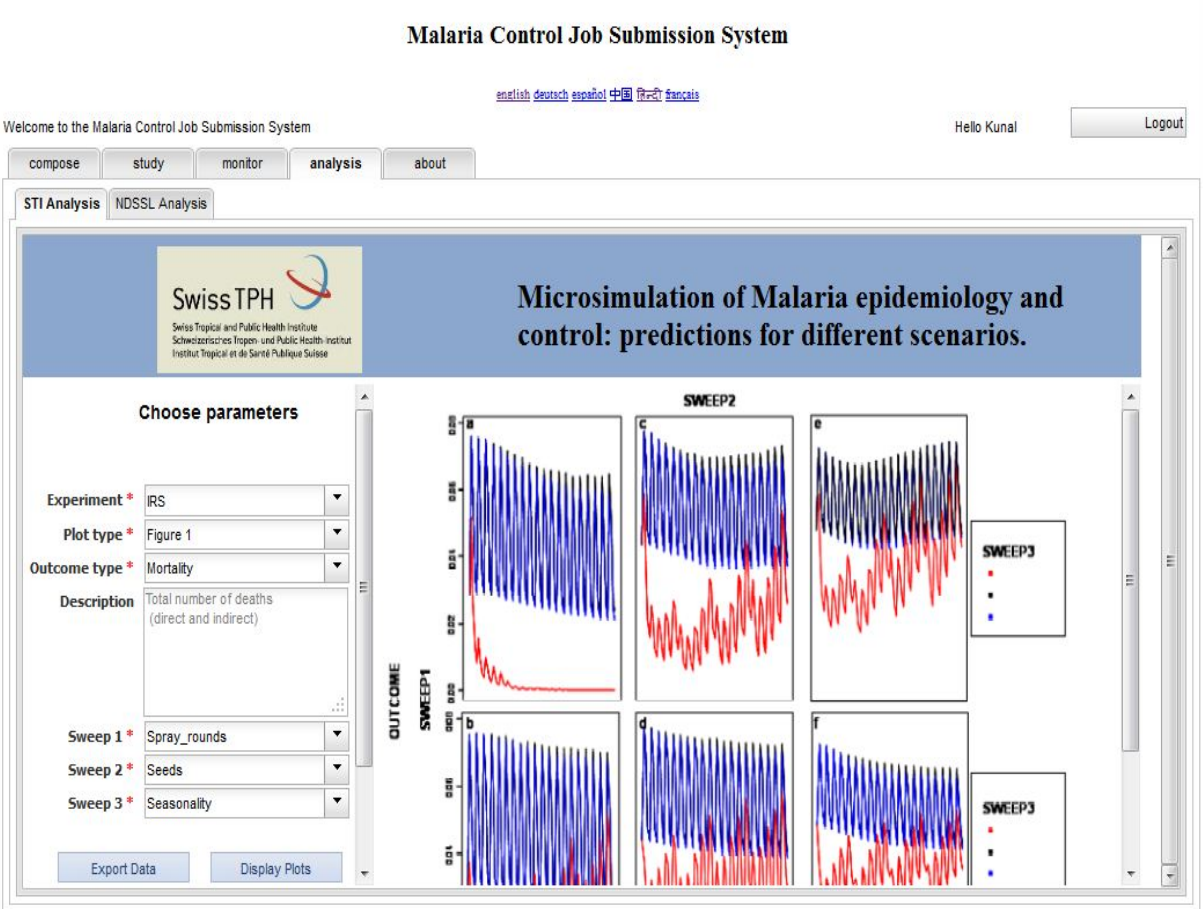

Figure 4.2: Malaria control analysis

to be loaded from the system. It allows users to save the new schema and the validate option can be used to verify whether or not all of the fields in the newly created scenario are correct and have valid values. After the schema is verified for correctness the user can then submit the study scenario for execution.

- Study: The study tabs list all of the studies that are created by a user. All of the experiments within the study can be seen by selecting an existing study. The users can then edit or add experiments in the study, remove existing experiments, and even resubmit the study for execution.

- Monitor: This tab lists the status of the jobs that are submitted by the users. It is used to monitor the status of the experiments that are currently executing on the back-end job submission framework. Users have the option to cancel the execution of study from

the system.

- Analysis: The analysis tab gives a graphical representation of the results in a study. Various graphical plots can be drawn for the experiments and the user can study the results by varying different parameters in the experiments. It also allows the users to export the graphs.

- About: This tab gives information about the complete Openmalaria project and all of the institutions and contributors for the project.

An additional tab is presented to the administrator of the application which allows it to add or delete users from the system, create new groups, and control the Openmalaria Web application. For a user, an Openmalaria job can consists of zero or more studies of which each can have, again, zero or more experiments. Furthermore, each experiment can create multiple sweeps. For every new sweep, the attribute value or element value of the base scenario is changed to create new scenario. These scenarios are then passed to the back-end system for processing.

The system generates an Extensive Markup Language (XML) file for each experiment in the study. The XML file contains the definitions and values for all the parameters in the experiment. It is validated against an XML schema definition (XSD) before a user can submit the study for execution. The schema description file is used to check whether all the elements and their corresponding values in the XML are correct and within a valid range. Once this XML file is validated, each study and experiment is assigned a unique ID which is later used in the job submission system for execution and monitoring.

As mentioned earlier the Openmalaria jobs can be executed on two different types of systems, a standalone or BOINC server depending on the priority of the job. The process of initialization and execution of jobs are totally different on both the systems. The XML generated for a study by the Web application is similar for both types of system. The framework does not need to create any new supporting files while running the job for a standalone system. For a BOINC server, supporting XML files need to be generated before the job can be successfully submitted to the server for execution. A resource broker which is currently under development will monitor the resources connected to the job submission system and select the best available option for executing the jobs if the priority of the job is undefined.

For each experiment submitted in a study, a Java object consisting of a study number, an experiment number, and a scenario XMLl file for the experiment is added to the JavaSpace for execution. The entry on the JavaSpace will also specify the execution type of the job (i.e., standalone or BOINC server) so that the correct broker can retrieve and execute it on the correct system. It also consists of other details, such as the lifetime of the object on the JavaSpace and the status of the job when it is submitted. Any change in the status of the entry on the JavaSpace can be monitored on the front-end interfaces by the user. Every job entry on the JavaSpace expires after a finite time, even if the execution is not completed, which ensures that none of the jobs remain on the JavaSpace forever. After the execution is over, all the results or error files for the study are moved to a correct location on the system from which they are added to a digital library for analysis.

4.2 Standalone Broker:

High-priority Openmalaria jobs are executed on a dedicated machines thath have standalone Openmalaria binaries installed on them. The execution time for these jobs is much less in comparison to the execution time of jobs on a BOINC system. The standalone broker retrieves jobs from the JavaSpaces and executes them on the Openmalaria binaries in the system. The standalone broker can run on multiple systems which have the Openmalaria binaries installed on them. The standalone broker performs all of the operations of a generic broker described in Chapter 3. The runner for the Standalone broker is designed to execute the tasks on the Openmalaria binaries, monitor the progress, and return the result. Execution jobs run quicker on systems that are operating the Standalone broker because they have dedicated hardware resources for computation unlike the BOINC broker, which relies on volunteering resources for computation. The flowchart of steps taken by a standalone broker while executing a job is shown in Figure 4.3.

The jobs in a standalone broker follow a sequential path because the execution time is short and so that a single thread is sufficient to start the job, monitor its execution, wait untill the job is completed and send the results back to the system. The sequential model also helps to ensure that only the defined number of worker threads in the broker can run the Openmalaria binary at any given point of time. The number of Openmalaria jobs that can

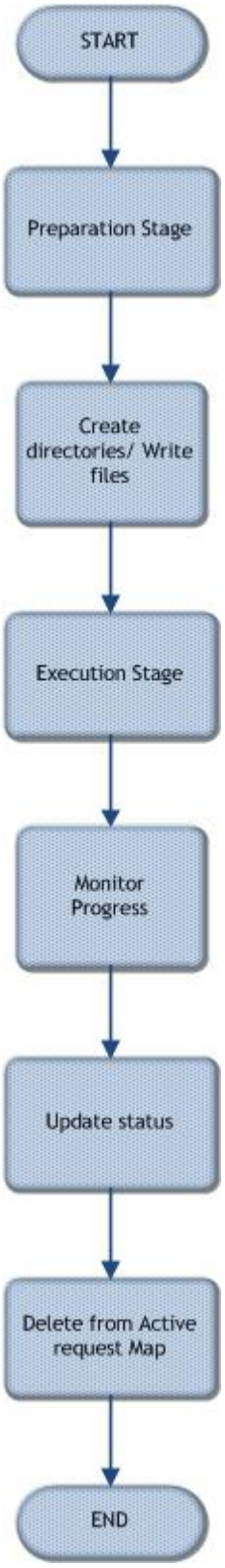

Figure 4.3: Flowchart for standalone broker

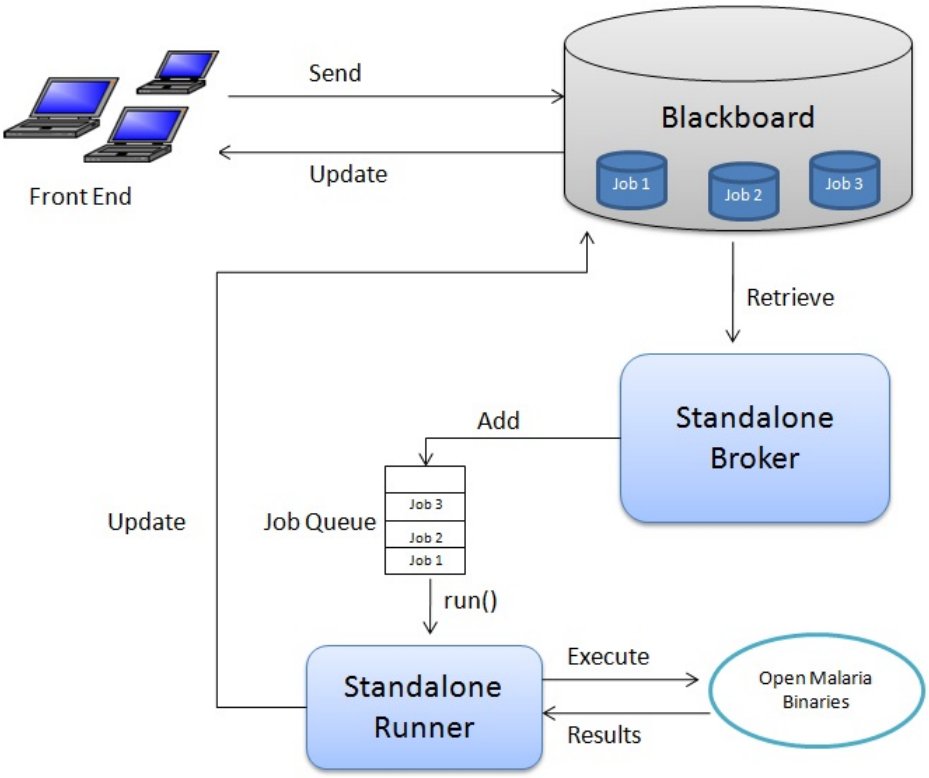

Figure 4.4: Standalone broker architecture

run concurrently depends on the number of worker threads the broker creates for execution. The worker thread calls the run() method in the Openmalaria_Standalone_Runner class, which updates the status of the job, copies the files to the correct directory for execution, runs the binary with the necessary parameters, and then waits for the Openmalaria binaries to complete execution so that it can pass the results or errors back. Figure 4.4 shows the overall operation of the standalone broker.

The Openmalaria.properties file defined by the users contain information and directory locations of the binaries for the standalone broker. Before execution, the broker reads this properties file to find the information about the Openmalaria binaries. The information in the Openmalaria properties file is user-defined and can be changed as per the system. For a standalone broker, the properties file defines the absolute path of the directory location for the Openmalaria binaries *(edu.vt.ndssl.Openmalaria.bin_directory)* and the name of the binary which will be executed *(edu.vt.ndssl.Openmalaria.binary)* . The properties file also contains the location for the output directory *(edu.vt.ndssl.openmalaria.outputDirectory)*

where the result files should be moved after execution.

Before placing any job on the blackboard, the front-end system converts all of the supporting files and information into a stream of bytes. The OpenmalariaRunData class handles the task of reading the input files and converting them into a byte stream. The OpenmalariaRunData object is then wrapped within a Openmalaria_SA_Runner object and a OpenmalariaSAJobRequest object is formed by using this runner object. The OpenmalariaSAJobRequest object is then added to the blackboard as an execution job. The blackboard monitor thread, when looking for a new OpenmalariaSAJobRequest on the blackboard, retrieves this object from the blackboard and places the runner object in the runnable queue for the worker threads to execute. When the blackboard monitor thread retrieves the job from the blackboard, it updates the status of the job from new to starting on the blackboard to notify other systems that the job is currently under execution. When a worker thread retrieves a job from the runnable queue, it starts its execution by calling the run method of the standalone runner object. Before calling the Openmalaria binaries, all of the files should be placed in the current working directory. In the initial phase of execution, the broker's properties are parsed to identify the directory where the input file and results files would be placed on the system. The runner creates a new directory for each new study and/or new experiment within the study.

The next phase is preparation in which the supporting files are written in the correct experiment directory for execution. The OpenmalariaConfigBuilder converts the OpenmalariaRunData object from the runner object into a series of XML and XSD files which are used by the Openmalaria binaries for execution. At the end of the preparation phase, all of the files required for the execution of the Openmalaria binaries are present in the correct directory.

The Openmalaria binaries are then executed by making a system call and passing in all the required files as arguments from the experiments working directory. When the execution completes successfully a results file is created in the same working directory as other supporting file written for execution. The name of the result file is passed as one of the arguments to the Openmalaria binary. It is important to call the Openmalaria binaries from a correct execution directory, which contains all the supporting files. Using a JAVA runtime process management class, new processes can be started from any specified direc-

```
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:42,334 - printing stdo
ut
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:42,334 - BoincWrapper:
 not using BOINC
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:42,447 -
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:42,692 - [  1%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:43,445 - [  2%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:44,137 - [  3%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:44,654 - [  4%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:45,164 - [  5%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:45,665 - [  6%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:46,185 - [  7%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:46,633 - [  8%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:47,091 - [  9%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:47,572 - [ 10%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:48,097 - [ 11%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:48,552 - [ 12%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:49,001 - [ 13%]
INFO pool-1-thread-1 edu.vt.ndssl.broker 2011-07-07 01:32:49,423 - [ 14%]
```

Figure 4.5: Execution of standalone Openmalaria job

tory. The Openmalaria binaries are executed as sub processes that use the JAVA runtime process management class. The sub process can be monitored to check for outputs or errors generated while running the binaries. The Openmalaria binary prints the percent execution that is completed while running. The runner records this information in the broker logs in the current implementation. Any errors generated while execution occurs is also recorded in the same broker log. The Figure 4.5 below show the output from the log file, which describes the execution of the Openmalaria binaries.

The result files are not generated if there were any errors during execution. This could also be a check condition that would ensure that no errors occurred during execution if a result file is generated for that scenario. Multiple worker threads running different Openmalaria jobs call the same Openmalaria binary on the system. Since the binaries are executed in a separate sub process within the runners, data corruption or the generation of wrong output is unlikely. When the execution is completed, the entry on the blackboard should be updated with the new job status. The runner thread calls an updateBlackboardStatus method, which updates the entry of the job on the blackboard (i.e., JavaSpace). This status can either be a RunComplete to indicate successful job completion or a RunFail to indicate errors during execution. Changing the state of the job is necessary to ensure that the broker's worker threads do not rerun the same job after its execution is complete.

If the execution is successfully completed, then the result files generated needs to be moved to output directory as specified in the standalone properties file. The *edu.vt.ndssl.openmalaria.outputDirectory* property in the Openmalaria properties file indicate the directory location where the result files should be copied. New directories are created according to the study and experiment number to copy the result files. The results then get added to the digital library for further analysis. In the final phase of execution, the broker queue needs to be updated to indicate job completion. The runner directs its base class, simfraserveRunner, to notify that the execution of the job has finished therefore the active requests map and the blackboard interest queue should be updated to remove the brokers' interest in that particular job.

The Openmalaria standalone broker executes the results faster; therefore it should be used for tasks which require faster results. The general architecture of the standalone broker can be easily replicated to execute jobs over different binaries by changing the configuration files and a few internal structures within the runner class. All the runner classes and the supporting libraries make it very simple to develop and debug applications designed using the standalone broker

4.3 BOINC Broker:

BOINC brokers are used for executing Openmalaria jobs designed to run on a BOINC server. BOINC operates on the principle of volunteer computing in which the users volunteer to share their computing resources for executing jobs from a BOINC project. Since access to the user resources is not consistent in BOINC, jobs run slower as compared to standalone binaries. The design and architecture of a BOINC broker is completely different from a standalone broker. The BOINC broker is designed to operate only on systems which have a BOINC server running on them. A complete execution of the Openmalaria job on a BOINC server involves two different brokers, the BOINC job broker and BOINC execution broker. The BOINC job broker handles the creation of all of the metadata before executing any BOINC job and the BOINC execution broker executes and monitors the job while it is running on a BOINC server.

The separation of the execution broker from the BOINC job broker has several advantages.

The design of the system is simplified since two independent brokers handle totally different tasks. The job broker no longer has to worry about managing resources to monitor updates since the execution broker can notify the job broker when the status of the job has changed. Since the operation of the execution broker is very generic, the same execution broker can be used with other BOINC job brokers to monitor different jobs. The BOINC execution broker is explained in the details in section 4.4. The flowchart of steps taken by a BOINC broker while executing a job is shown in Figure 4.6.

The initial execution flow of the BOINC broker is similar to the standalone broker. The BOINC broker retrieves the job and the worker thread begins an execution of the runner. Similar to the standalone broker, the runner thread creates files that are necessary for execution. Then, instead of running the job and waiting for its execution to end, the runner creates a new execution job, notifies the blackboard monitor thread to monitor it, and exits execution. The workers threads in the BOINC broker do not have wait for the runner to finish execution of the complete job. Once the runner exits from submitting an execution job, the workers can then handle any other outstanding BOINC requests. After the completed job returns from the execution broker, the job runner is again executed to complete the final phase of the job execution, such as moving the results or notifying any errors during its operation. The architecture of the BOINC broker is shown in Figure 4.7.

The BOINC broker's properties are defined in the openmalaria_BOINC.properties file. The BOINC job runner reads the properties files to extract information about the BOINC server and supporting files. The variable *edu.vt.ndssl.openmalaria.boinc_directory* defines the directory of the BOINC server. All of the supporting files and executables required to submit the job to a BOINC server are present in this directory. The *edu.vt.ndssl.openmalaria.boinc_wu_template* and *edu.vt.ndssl.openmalaria.boinc_result_template* define the directory location within the BOINC server directory where the work unit and the results file for each job should be placed. There are other defined variables which are used by the execution broker to read information about the BOINC databases.

The job request is made of an OpenmalariaBOINCJobRequest object which encapsulates the Openmalaria_BOINC_Runner object. The Openmalaria_BOINC_Runner contains the OpenmalariaRunData object which converts the XML files that are generated by the front end for every experiment in the study into a stream of bytes. The blackboard monitor thread

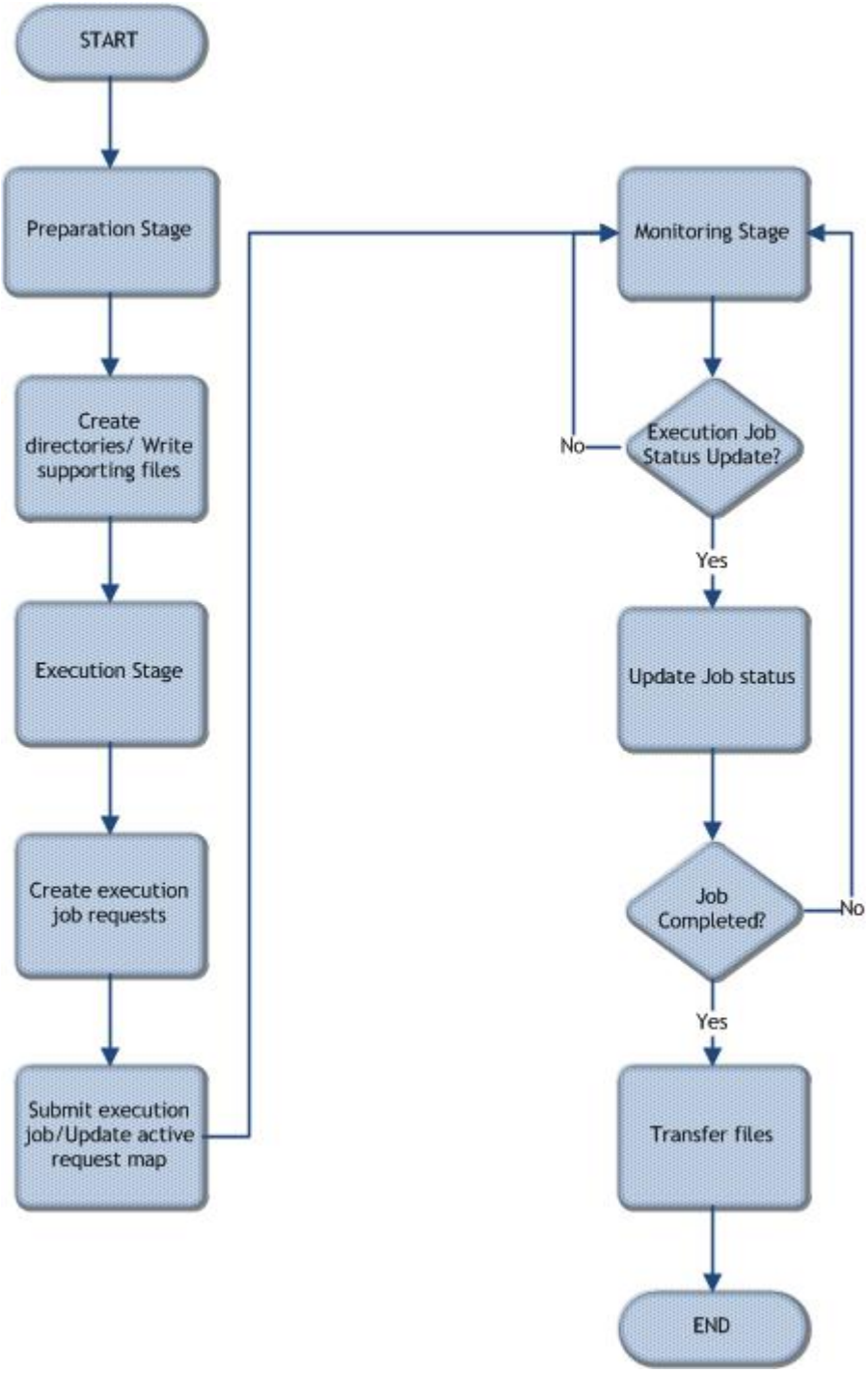

Figure 4.6: Flowchart for BOINC job broker

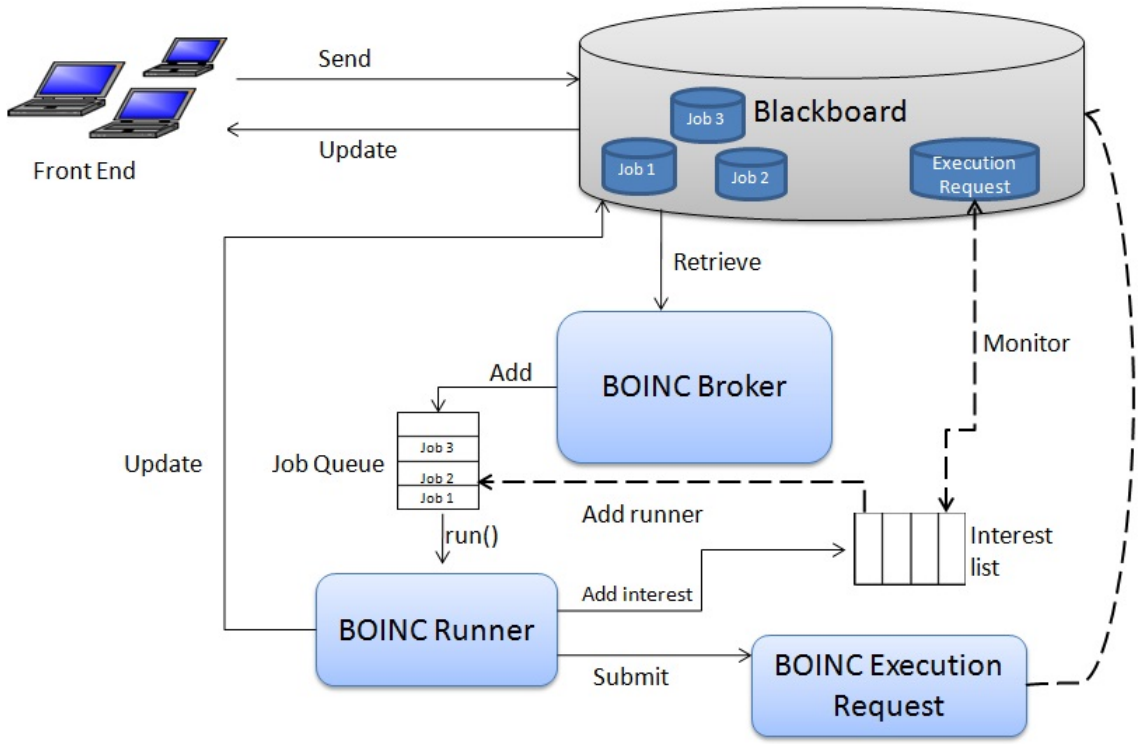

Figure 4.7: BOINC broker architecture

watching for a new job of type OpenmalariaBOINCJobRequest on the blackboard retrieves this object from the blackboard and adds it to the execution queue. The blackboard monitor thread also updates the status of the job from new to starting on the blackboard to notify other systems that the job is currently under execution. The worker threads within the broker read the execution queue (i.e., runnable queue) to find any new jobs for execution. The broker gets the runner object from the job request and calls the run method to begin execution. It passes all of the information required by the runner object for execution. Unlike the worker threads in the standalone broker, the BOINC broker thread does not wait for the complete execution to finish by the runner. The runner hands off the execution to an execution runner and returns quickly so the worker thread can process other requests. When the execution job status changes, the worker thread reruns the job runner object with the new job status. The BOINC job can only be in one of the four states at any given time during its execution (FAILED,DONE,RUN_SIM,PREPARE).

In the preparation phase of execution, the runner prepares all the files required by the BOINC server for running the job. The BOINC server requires all the files to be placed in specific

directories for correct execution. As mentioned earlier, the properties file is specific to the directory locations and the name of the files for the workunit and results file on the BOINC server. The workunit and results file contains definitions for parameters which describe the way that the task is executed, URL's for the supporting files, experiment files, etc which are required by the BOINC server and the BOINC clients. A sample workunit and result XML file can be found in Appendix A.

When a job is submitted to a BOINC server, a utility program that gives information about the location of the supporting files to the server is executed. In order to run the job correctly, the supporting files should be placed in the right directories. BOINC saves all the files related to a project in hierarchical upload/download directories. Each directory has a set of 1024 sub directories. Each file is hashed into these directories. This hierarchy is used only for the input and output files. Before creating BOINC work for the current job, the files should be placed in the right folder in the hierarchy. The BOINC installation provides a utility program (dir_hier_path) which prints the full path name of the given file in the hierarchy. So, after the experiment, XML files are created in the preparation phase and the runner copies the files in the right path printed by the utility program. The BOINC runner copies the scenario XML and XSD file along with the densities.csv file which is needed while running the Openmalaria binaries on any machine. The OpenmalariaConfigBuilder class handles creation of all the supporting files and copying them over to the correct directory. At the end of the preparation phase, all of the files required for the execution of the Openmalaria binaries are present in the correct directory.

In the execution phase, the BOINC runner makes a new execution job request for executing and monitoring the task on the BOINC server. The BOINC runner forms a command which the execution runner needs to run in order to submit jobs on the BOINC server. The query command contains the file names as arguments along with complete relative path for each file. The BOINC job runner creates a new BoincExecutionRequest object that contains the query command and other information related for execution and places the job request object on the blackboard. In order to check whether the execution for the job request is complete, the runner needs to monitor for any status changes for the execution job request on the blackboard. The runner then adds its own execution ID to the active request map for the broker's worker threads to re-execute. The state of the BOINC job request on the blackboard is then changed from PREPARE to RUN_SIM, indicating that the simulation

is now running on the execution runner. The execution ID of the new execution job request is then added to the broker's interest list. This list is updated by runners to express interest in particular entries on the blackboard. The blackboard monitor thread monitors the status of the entries on the blackboard and notifies any change of status to the interested runners.

The blackboard monitor thread calls the updateStatus() method of every runner whose execution request has changed status on the blackboard. The updateStatus() method compares the old and new status of the execution job request and then updates the status of the BOINC job request on the blackboard. It then sets a flag to indicate that the runner for the job entry needs to be pushed back on the runnable queue in the broker to process the update. When the runner is added to the runnable queue, the worker thread retrieves it and calls the run method of the runner object. This time before processing anything, the BOINC job runner checks to see the new status of the BOINC execution runner and then updates the status of itself to indicate whether the execution completed successfully or if it failed.

If the status of the execution request is RunFail ,it indicates that the execution has failed on the BOINC server. At that point the current status of the BOINC job broker is changed to FAILED to indicate execution failure. If the status of the execution request is RunComplete, it indicates that the execution is complete and the output is ready from the BOINC server. The execution runner can also update its status to RunStart, RunQueued, or RunExecuting, indicating that the execution is underway or that the job is currently under execution. During this stage, the BOINC job broker does not update its state and remains in the RUN_SIM phase. The states are used for monitoring purpose and to check whether the execution request started correctly.

In the final stage after the execution is complete, the job runner needs to copy the output files to the results directory. The *edu.vt.ndssl.openmalaria.outputDirectory* property defined in the openmalaria properties file indicate the directory location to copy the result files. When the results files are copied, they are added to the digital library so that they can later be accessible to the users.

The current status of the BOINC job can be found by monitoring the status of its entry on the blackboard. A major advantage of BOINC job broker is that a single broker can handle multiple job requests as it does not have to wait for the runner to complete execution. Once the job is handed to the execution broker, it only needs to monitor the status of the execution

job and update its own status on the blackboard.

4.4 BOINC Execution Runner:

The BOINC execution runner handles the execution and monitoring of Openmalaria BOINC jobs. When a new job is submitted to a BOINC job broker, it prepares the files for execution, forms a new execution request, and places the execution job request on the blackboard. The brokers monitoring for jobs of the *edu.vt.ndssl.runner.BoincExecutionRequest* type are designed to handle BOINC execution requests. Similar to the other brokers that have been discussed, when the execution broker finds an execution request on the blackboard, it retrieves it and places it in the runnable queue so that the worker threads may execute it. The worker thread retrieves the runner object from the runnable queue and calls the run method to start execution. The flowchart for the operation of the BOINC execution runner is shown in Figure 4.8.

In the initial phase, the execution runner submits the job to the BOINC server for execution and retrieves the ID for the job from the BOINC database. The execution runner object that is formed by the BOINC job runner contains the query which submits the job to the BOINC server. When the brokers' worker thread calls the run method of the BOINC's execution runner, the runner thread executes the query and submits a new job to the BOINC server. In order to monitor the job in the later stages it is important to find the ID of the job from the BOINC database. During the initialization phase itself, the runner make a request to the BOINC database to retrieve the workunit ID for the new job that was submitted. The database returns the ID which the runner saves for future monitoring. This is a unique ID for the job on the BOINC server. A job in the execution runner can only be in one of the following states at any point during the execution: FAILED, DONE, DELETED, COMPLETED, EXECUTING, or INITIAL. Figure 4.9 shows the overall architecture of the BOINC Execution Broker.

The query that the runner executes contains the path of the results file and workunit file as arguments. To run the execution command for the BOINC server, other supporting information such as XML, XSD, and the densities.csv are also passed as arguments. The execution command should be executed from the BOINC server's working directory. Once

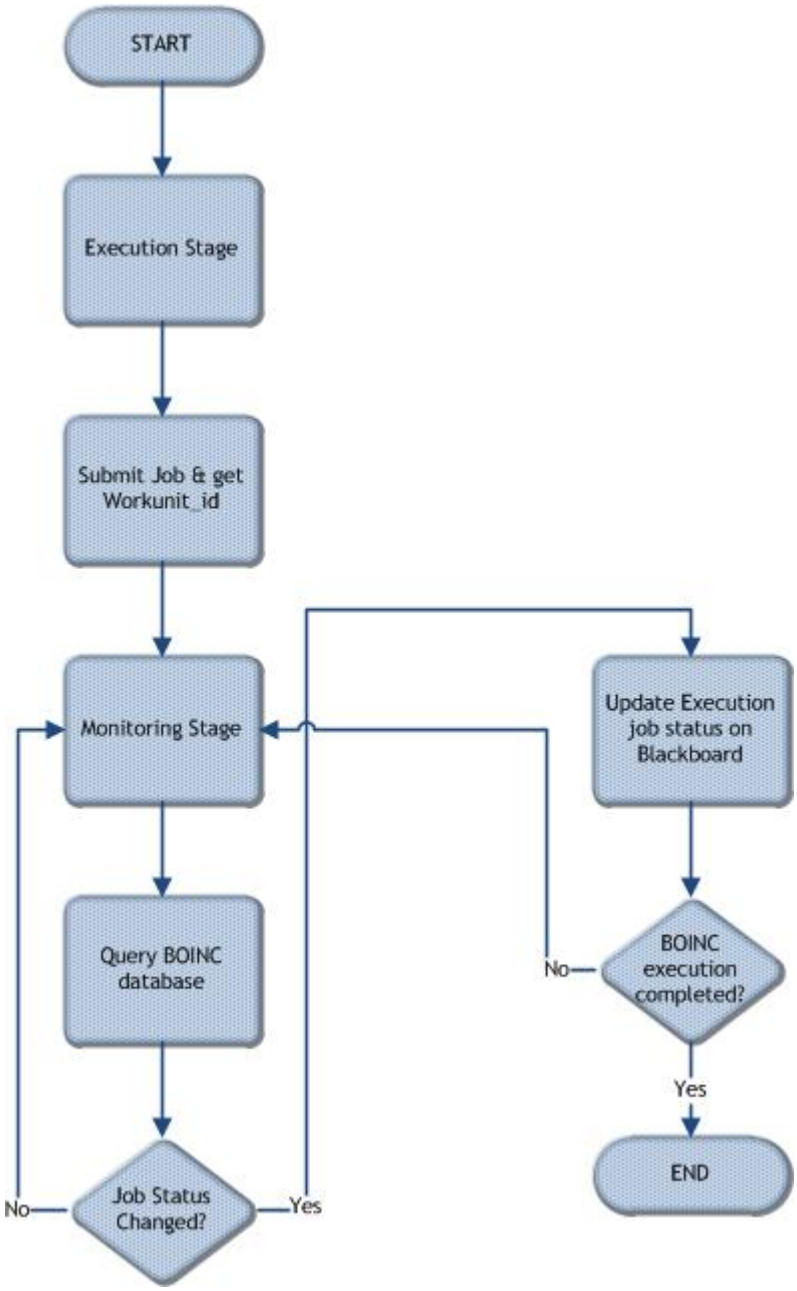

Figure 4.8: Flowchart for BOINC execution broker

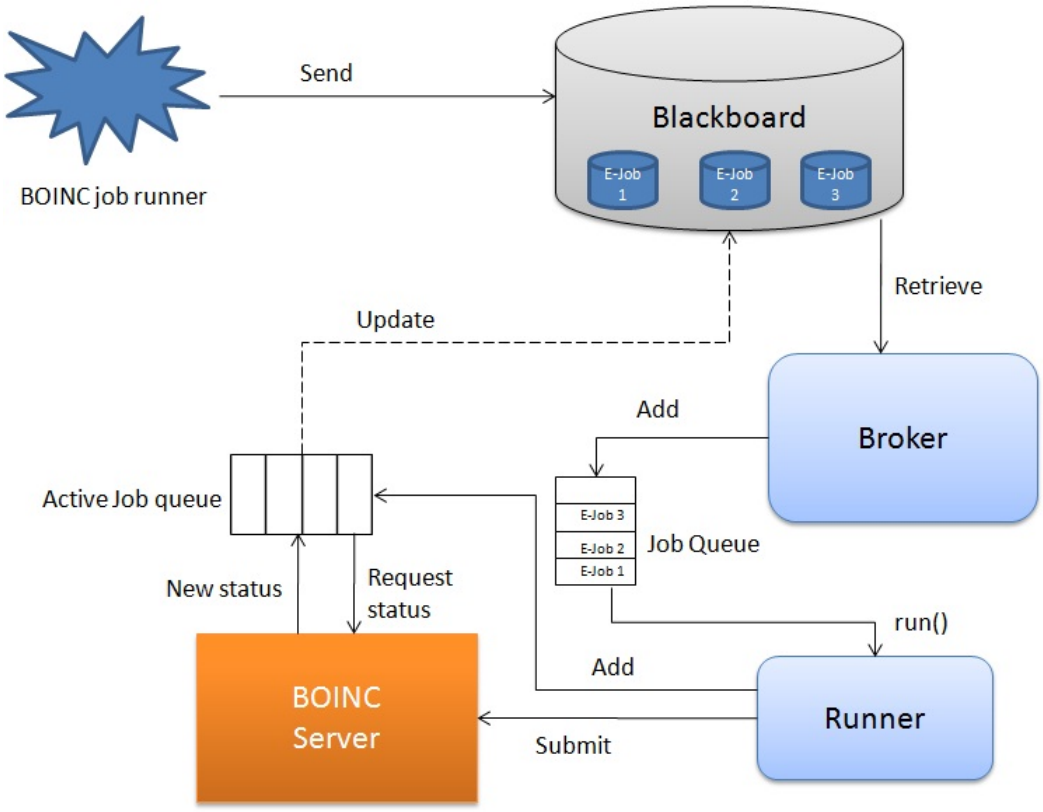

Figure 4.9: BOINC execution broker Architecture

the command is executed correctly, without errors, the job is added to the BOINC server queue for the clients to execute. The workunit ID in the workunit table is the unique ID to represent the information about the job in the database. BOINC uses a MySQL database and the runner uses a JAVA-MySQL connector (JDBC) for connecting to the MySQL database.

The BOINC server restricts the access of the database to any registered user on the system. This is done in order to protect the information in the database from any unauthorized access. To allow the execution runner to access the database, the super user for the BOINC server needs to changes the restriction. Instead of allowing access to any user and creating a security loophole, a dedicated user for the execution runner client could only be given access to the database. The runner can then use the credentials of the dedicated user to access the database. The new user can only have the access rights to read the information in the database since the execution runner does not make any changes to it. Further, restricting the access to a particular workunit table or database would ensure more security to the BOINC server.

The credential for the new user should be passed to the execution runner to run queries on the database. The Openmalaria properties file contains the credentials for the user which is used to access the database. The username and password are mentioned in the *edu.vt.ndssl.Openmalaria.db_username* and *edu.vt.ndssl.Openmalaria.db_password* respectively. For an empty password field, the runner uses a blank password to establish a connection to the database. The *edu.vt.ndssl.Openmalaria.db_hostname* parameter in the properties file specifies the hostname of the system on which the BOINC server is running. The *edu.vt.ndssl.Openmalaria.db_name* parameter specifies the database name to which the user has access. The name of this database is usually the project name for the application on the BOINC server.

When an execution broker is initially loaded, it creates a new BOINC monitor thread which checks the status of all the BOINC jobs that are currently running. When a new job is submitted to BOINC, its unique workunit ID is retrieved and added to the execution queue to check the status of the job. After a job is submitted to a BOINC server, the state of the execution job changes from starting to execution. It remains in the execution phase until the processing of jobs is completed. The BOINC monitor thread polls the running queue after a specific time period to determine whether or not the jobs that are

currently running have changed states. As explained in section 2.4, the workunit table in the BOINC database gives the information about the current job running on BOINC server. The assimilate_state value in the workunit table gives the most current state of the job on the BOINC server. The BOINC monitor uses the same database credentials as the execution runner to retrieve the status of the job from the database. The workunit table returns an integer value for the status of the job which gets mapped to an execution state according to the documentation for the workunit table. The integer is mapped to any one of the following states: QUEUED,RUNNING,ERROR, COMPLETED, DELETED, UNKNOWN, NEW.

For every current job running in the BOINC queue, the BOINC monitor thread queries the database using the workunit ID from the queue and retrieves the current state of execution of the job from the BOINC database. The execution runner then compares the current status of the job with the new status from the BOINC monitor thread. If the statuses are different, a thread-safe synchronized function updateBoincStatus is called by the execution runner to update the status of the job on the blackboard. The new status causes the BOINC job broker to rerun and update the status of the BOINC job on the blackboard. In the final phase after the execution is completed, the execution runner updates the active request jobs queue to indicate that the job has completed. It then removes any references for the job from the execution broker and also updates the blackboard with the necessary information (i.e., whether or not the job completed or failed) for the BOINC job broker to process. Since the execution broker is a totally different entity, the BOINC job broker cannot modify any properties of the execution broker. It can only register an interest in an execution job and monitor the job on the blackboard.

Chapter 5

Related Work

The traditional means of supplying computing power to projects has been through "Grids". Most of the existing research has focused on designing and developing a job submission system for grid computing. A grid is designed of multiple computing resources of heterogeneous types that are coupled together. Applications that are running in a grid environment contain not only varying resource requirements, but also other constraints such as data locality and specific execution deadlines. The use of centralized policies is not suitable for resource scheduling on a grid configuration since performance metrics for the grid are not available and the complete state information about the resources cannot be determined for scheduling [17].

Even though grids and public resource computing share the same goals of executing jobs on computing resources, there are many differences between a public resource computing platform (i.e., volunteer computing) and a grid network. There are various challenges in connecting the grid network and public resources computing together under a centralized job submission system. The existing grid submission framework cannot be modified directly to submit jobs to volunteer computing project like BOINC. Even then it is important to understand the operations of the existing job submission system for grids in order to design a sustainable job submission system for BOINC. Some existing research has guided us in the directions of designing a well defined job submission system for jobs of varying priority. In the following section, we describe two types of grid job submission systems. In the later part, we explain the lattice project- a grid computing framework that has the functionality

to assign jobs to a BOINC server.

5.1 Nimrod-G:

The Nimrod-G grid resource broker is a grid system which uses computational economy driven architecture for scheduling task farming applications and managing resources on a large distributed network of resources [9]. The Nimrod-G resource broker provides a framework for running applications within the quality of service-driven resources allocation and regulating supply and demand for resources [10]. The Nimrod-G resource broker operates in the User-level middleware architecture described in Chapter 1. The key components of a Nimrod-G broker consist of [9]:

- A task farming engine (TFE) that performs the task of creating jobs, managing jobs status, and interaction between clients.
- A scheduler that performs resource discovery, trading, and scheduling.
- A dispatcher and actuator to deploy the agents and assign jobs for execution.
- Agents for setting up the execution environment and managing the execution of jobs on resources.

The Figure 5.1 gives the flow of actions in the Nimrod-G runtime environment. Nimrod-G utilizes its own Nimrod clients on the resources for communication with the broker. The complete framework closely resembles the architecture of BOINC, which does the task of resource scheduling and management. The difference between Nimrod and BOINC is that BOINC is used for executing applications on a non-clustered system which operates on the principle of stealing computer cycles while the system is idle. Nimrod-G, on the other hand, is helpful in managing resources which are distributed globally.

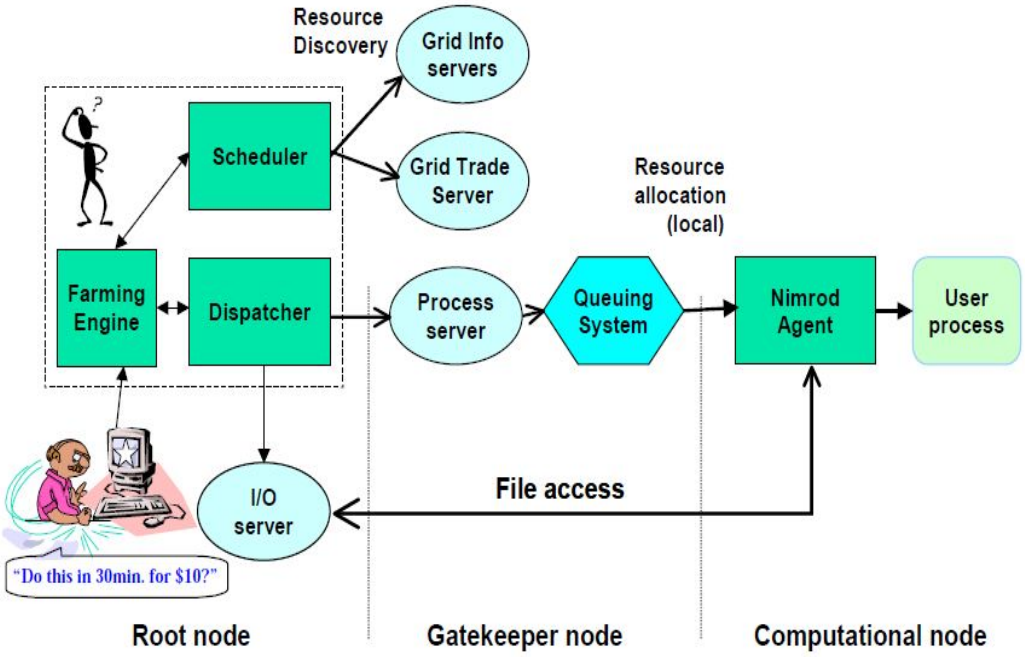

Figure 5.1: Nimrod G flow of action [adapted from [10]]

5.2 Condor-G:

The Condor-G is the job submission broker for the Condor Globus interface. The Condor-G acts like a window to the grid service and lets users submit and monitor their jobs [15]. Condor-G provides services like job monitoring, notification, logging, fault tolerance for jobs, user credential management, and can also handle complex job interdependencies [15]. Condor is a framework to manage a collection of computers owned by individual. Condor uses the process of matchmaking to identify available resources and match user demands [28]. Condor assign jobs to the resources based on the priority of the job and this priority is assigned by Condor itself to the job using the up-down algorithm. Condor used the mechanism of checkpointing which allows it to return the resources to the user when preempted. Unlike BOINC, Condor does not steal the computer cycle; instead, it waits for the computer to become free by detecting the mouse and keyboard activity and then starting to run the jobs [17].

There are many other job submission system for the grid which operate in different layers of the job submission framework. The taxonomy of some of the available systems is described

in Appendix B.

5.3 The Lattice Project (TLP):

The lattice project is a grid computing research project and production system developed at the University of Maryland [6]. The idea of the project is very similar to the job submission system we designed. The lattice project is used to develop a unified grid submission system which would allow heterogeneous computing system to be uniformly used and addressed [19]. The architecture of the Lattice project is shown in Figure 5.2.

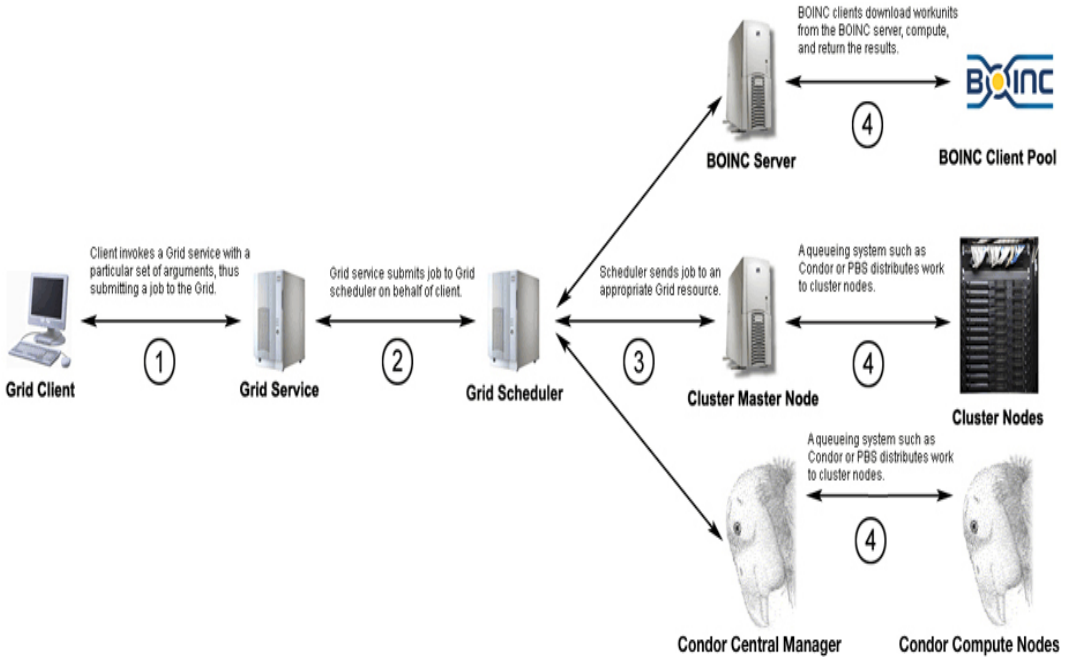

Figure 5.2: Architecture of TLP [adapted from [29]]

TLP uses the globus toolkit, which is a framework for grid protocols and services [7]. Globus provides facilities for resource management, execution, job submissions, etc. TLP treats BOINC as one of the resources connected to the grid. The research team has managed to connect the Globus framework to the BOINC server to allow clients to submit work units to the BOINC server. TLP extends the Globus GRAM interface for job specification and submission. The fields in GRAM's job description document (JDD) are mapped to the BOINC workunits XML fields [19]. The results from the BOINC server are copied to

a directory which is accessible to the Globus job manager for transferring it back to the client. The main goal of the lattice project is to satisfy the computational needs of research projects. This project is idle for connecting and managing different grid resources through a unified grid management system. The TLP team has successfully converted multiple BOINC applications to run on their grid system [22].

Chapter 6

Future Work and Conclusion

Before summarizing our contribution in 6.2, we will discuss the areas in which the research can be continued. The Future Work section 6.1 will discuss some of the new features that can be added to the existing system and describe a new direction in which the research can be carried out.

6.1 Future Work:

The framework is fairly new and there is a broad scope for developing new tools by using this framework. In our design we have developed brokers for BOINC and standalone systems only, but new systems which can handle the Openmalaria tasks can be easily plugged into the framework. Following are some of the areas in which we think the future work can be done with the job submission system.

- **Measuring the performance:** At present we have not evaluated the performance of the system for the BOINC or standalone broker. A good study would evaluate the difference between the running times for a job on the standalone system and the BOINC server. The study could help validate the claim that the jobs runs faster on the standalone system. Also, measuring the performance of the framework under varying loads and running a variety of jobs through the same space would help to understand how the system handles the different jobs and whether the jobs are sent correctly to the

respective brokers of different jobs.

- **Better reporting mechanism:** In the current implementation, the status of the job is known only by polling the job on the Javaspace and checking its execution status. An enhancement to the report mechanism would be a notification system which informs the front-end servers about the change in the status of the job. This would be a useful feature when the number of jobs submitted to the Javaspace increases.

- **Resource scheduler:** When the job is submitted to the JavaSpace, the first broker that reads the job receives that assignment. A good enhancement would be designing a resource scheduler for the job submission system. The resource scheduler can select the resources which it wants to submit the job; thus, the work load can be distributed proportionally between all of the connected resources. There is a lot of existing research on designing an efficient resource allocation algorithm, so by implementing such an algorithm the framework can be made smart and the user will not have to worry about selecting the resources as the framework would select the best available option from the resources.

- **Resource identifier:** In the present implementation, the broker needs to be installed on the resources and started so that it can communicate with the Javaspaces. An enhancement to the framework would be to develop a system which when supplied the hardware information would contact the resources, get the machine information, install the correct broker code on the system and start communicating with the JavaSpace to handle job execution.

6.2 Conclusion:

In this thesis, we presented a job submission system which can handle Openmalaria jobs to run on standalone and machines that run BOINC servers. The job submission system was designed to process jobs as per the resources available and the priority of the jobs in the system. It demonstrates the novel broker design for BOINC servers. The general purpose architecture of the brokers can be extended to create new system which can run different jobs or extend the existing framework to add new type of resources running the Openmalaria

jobs. The use of JavaSpaces to submit jobs opens doors for plugging in different applications which use the same technology for communication. The JavaSpaces can act as a reliable distributed storage for jobs and provides mechanism for high security, concurrent access, and automatic storage and retrieval of entries. A number of different resources running the broker can be connected to the same framework thus making the framework extensible.

The design of a separate execution broker for monitoring and running BOINC jobs within the BOINC broker makes the system modular and easier to debug. The same execution broker can be used to submit and monitor different types of jobs which run on a BOINC server. By developing the system on top of the Simfraserve infrastructure, we were able to reuse a lot of the existing API and develop new functionality which can be used to design and develop other brokers. The framework has a lot of scope of extension, so newer systems with different capabilities can be connected for executing different variety of tasks.

Bibliography

[1] David P. Anderson. Designing a Runtime System for Volunteer Computing. *Physics*.

[2] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 1999.

[3] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45:56–61, November 2002.

[4] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8 pp. –203, july 2005.

[5] Mark Baker. Cluster computing white paper. *CoRR*, cs.DC/0004014, 2000.

[6] Adam L. Bazinet and Michael P. Cummings. The Lattice Project: a Grid research and production environment combining multiple Grid computing models. *Distributed & Grid Computing–Science Made Transparent for Everyone. Principles, Applications and Supporting Communities*, 2007.

[7] Adam L. Bazinet, Daniel S. Myers, J. Fuetsch, and Michael P. Cummings. Grid Services Base Library: A high-level, procedural application programming interface for writing Globus-based Grid services. *Future Generation Computer Systems*, 23(3):517–522, 2007.

[8] BOINC : Middleware for Volunteer Computing. `https://confluence.pegasus.isi.edu/download/attachments/5242944/boinc.pdf?version=1&modificationDate=1263591118000`. [Online; accessed 29-July-2011].

[9] Rajkumar Buyya. *Economic-based distributed resource management and scheduling for grid computing*. PhD thesis, 2002.

[10] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid. *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, pages 283–289, 2000.

[11] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. *High-Performance Computing in the Asia-Pacific Region, International Conference on*, 1:283, 2000.

[12] Rajkumar Buyya and Manzur Murshed. GridSim : A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Computing*, pages 1–37.

[13] BOINC Database. `http://www.boinc-wiki.info/BOINC_Database`. [Online; accessed 30-June-2011].

[14] Ian Foster, Carl Kesselman, and Marina Rey. Computational Grids 1 Reasons for Computational Grids. 1998.

[15] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: a computation management agent for multi-institutional grids. *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*, pages 55–63.

[16] JavaSpace. `http://java.sun.com/developer/Books/JavaSpaces/introduction.html`. [Online; accessed 30-June-2011].

[17] Jeevak Kasarkod. A Configurable Job Submission and Scheduling System for the Grid A Configurable Job Submission and Scheduling System for the Grid. 2003.

[18] George Macdonald. *The epidemiology and control of malaria*. Oxford University Press, 1957.

[19] Daniel S. Myers, Adam L. Bazinet, and Michael P. Cummings. Expanding the reach of Grid computing: Combining Globus-and BOINC-based systems. *Grid computing for bioinformatics and computational biology*, page 71, 2008.

[20] Article on JavaSpace. http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/. [Online; accessed 31-May-2011].

[21] Andy Oram. Peer to Peer : Harnessing the Power of Disruptive Technologies. *Context*.

[22] Projects running on The Lattice Project. http://lattice.umiacs.umd.edu/research. [Online; accessed 28-June-2011].

[23] Ross, Ronald. *The prevention of malaria*. London, 1911.

[24] T Smith, N Maire, A Ross, M Penny, N Chitnis, A Schapira, A Studer, B Genton, C Lengeler, F Tediosi, D de Savigny, and M Tanner. Towards a comprehensive simulation model of malaria epidemiology and control. *Parasitology*, 135(13):1507–16, November 2008.

[25] BOINC Backend state variables. http://boinc.berkeley.edu/trac/wiki/BackendState. [Online; accessed 31-May-2011].

[26] Claudio J. Struchiner, Elizabeth M. Halloran, and Andrew Spielman. Modeling malaria vaccines I: New uses for old ideas. *Mathematical Biosciences*, 94(1):87 – 113, 1989.

[27] Swiss Tropical and Public Health Institute. http://www.swisstph.ch/. [Online; accessed 29-July-2011].

[28] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, February 2005.

[29] The Lattice Project Architecture. http://lattice.umiacs.umd.edu/architecture. [Online; accessed 28-June-2011].

[30] Wikipedia. BOINC — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing. [Online; accessed 22-March-2011].

[31] Ann Wollrath, Roger Riggs, Jim Waldo, Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System A Distributed Object Model for the Java ™ System. (June), 1996.

[32] How BOINC works. http://boinc.berkeley.edu/wiki/How_BOINC_works. [Online; accessed 30-June-2011].

Appendices

Appendix A

BOINC Workunit Template:

```
<file_info>
        <number>0</number>
</file_info>
<file_info>
        <number>1</number>
        <no_delete/>
</file_info>
<file_info>
        <number>2</number>
        <no_delete/>
</file_info>
<workunit>
<file_ref>
        <file_number>0</file_number>
        <open_name>scenario.xml</open_name>
</file_ref>
<file_ref>
        <file_number>1</file_number>
        <open_name>densities.csv</open_name>
</file_ref>
<file_ref>
        <file_number>2</file_number>
        <open_name>scenario_12.xsd</open_name>
</file_ref>
<command_line>--compress-checkpoints=1</command_line>
```

```
26 <min_quorum>1</min_quorum>
27 <target_nresults>1</target_nresults>
28 <rsc_memory_bound>330000000</rsc_memory_bound>
29 <rsc_fpops_bound>150000000000000</rsc_fpops_bound>
30 <rsc_fpops_est>20000000000000</rsc_fpops_est>
31 <rsc_disk_bound>450000000</rsc_disk_bound>
32 <delay_bound>300000</delay_bound>
33 <max_error_results>4</max_error_results>
34 </workunit>
```

Listing A.1: BOINC Workunit Template.

Appendix B

BOINC Results Template:

```
<file_info>
        <name><OUTFILE_0/></name>
        <generated_locally/>
        <upload_when_present/>
        <max_nbytes>10000000</max_nbytes>
        <url><UPLOAD_URL/></url>
</file_info>
<file_info>
        <name><OUTFILE_1/></name>
        <generated_locally/>
        <upload_when_present/>
        <max_nbytes>10000000</max_nbytes>
        <url><UPLOAD_URL/></url>
</file_info>

<result>
        <file_ref>
                <file_name><OUTFILE_0/></file_name>
                <open_name>output.txt.gz</open_name>
        </file_ref>
        <file_ref>
                <file_name><OUTFILE_1/></file_name>
                <open_name>scenario.sum</open_name>
        </file_ref>
</result>
```

Listing B.1: BOINC Results Template.