Design and Evaluation of a Public Resource Computing Framework

by

James D. Baldassari

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May, 2006

APPROVED:

_____

Dr. David Finkel, Major Advisor

_____

Dr. Craig E. Wills, Reader

_____

Dr. Michael A. Gennert, Head of Department

Abstract

Public resource computing (PRC) is an innovative approach to high performance computing that relies on volunteers who donate their personal computers' unused resources to a computationally intensive research project. Prominent PRC projects include SETI@home, Folding@Home, and distributed.net. Many PRC projects are built upon a PRC framework that abstracts functionality that is common to all PRC projects, such as network communications, database access, and project management. These PRC frameworks tend to be complex, limiting, and difficult to use. We have designed and implemented a new PRC framework called the Simple Light-weight Infrastructure for Network Computing (SLINC) that addresses the disadvantages we identified with existing frameworks. SLINC is a flexible and extensible PRC framework that will enable researchers to more easily build PRC projects.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Public Resource Computing (PRC) is a form of high performance computing (HPC) in which volunteers from around the world donate a portion of their computers' resources to a computationally intensive research project. Researchers who do not have access to supercomputing facilities can use PRC to increase their ability to process compute-intensive data at little to no extra cost.

PRC is related to grid computing, but there are some important distinctions. A grid is traditionally a collection of computers or entire clusters that are owned and maintained by universities, companies, and research organizations. PRC focuses on utilizing the available resources in the personal computers of individual volunteers rather than large networks of computers with persistent network connections and long uptimes. Another distinction is that grids are inherently distributed[1], but PRC systems tend to be centralized, with a single point of contact for all volunteers.

Although PRC is a powerful tool for research, not all research projects are compatible with the PRC model. To take advantage of the computing resources PRC provides, the program that is used to analyze research data, called the *science application*, must be parallelizable. A parallel program can be converted into a client-server system in which the client performs the computations and the server coordinates the clients. The server's primary responsibilities are to partition the input data into smaller *work units* for the clients to compute, and to store the results returned by the clients for further analysis.

There are several PRC frameworks that facilitate the creation of PRC projects by managing many of the responsibilities of the client and server. One of the most widely used PRC frameworks is the Berkeley Open Infrastructure for Network Computing (BOINC)[2]. The advantage of using BOINC is that it abstracts much of the complexity involved in creating and maintaining a large PRC project. BOINC will manage all network communications between the clients and the server, so there is no need for a PRC project developer to write any network code. BOINC manages the project database and connections to it, so developers do not need to write any database connection code or SQL queries. BOINC handles the distribution of work units and collection of results; it can recover from situations in which clients receive a work unit, but never return a result or return an erroneous result. The creators of BOINC designed it to be highly scalable, and the system can currently support on the order of tens of millions of client requests per day[3].

BOINC has an advanced and scalable architecture that is well suited to large PRC projects, but its implementation requires the developers of a PRC project to be very familiar with specific technologies like Linux and MySQL, as well as the C++ programming language. Compounding the difficulties BOINC project developers face is a lack of comprehensive documentation and configuration tools to automate common project development tasks.

The primary goal of this thesis was to create a new PRC framework that simplified the process of creating PRC projects. To accomplish this goal, we first designed a modular, object-oriented architecture for the framework. We named this framework the Simple Light-weight Infrastructure for Network Computing (SLINC). During the design process we tried to make this architecture scalable by allowing the separate modules to be run on different physical computers. After designing the

architecture we researched tools and third-party libraries that would decrease our development time by providing certain functionality that was critical to the framework. Once we had chosen these libraries and tools, we began implementing and testing the architecture.

We evaluated SLINC by performing functional testing, performance testing, and usability testing. The functional tests evaluated the core functionality of the system to verify that each feature worked correctly. The performance testing evaluated the framework's ability to scale when multiple volunteers were contributing to a PRC project simultaneously. The usability testing evaluated SLINC's ease of use and identified areas for improvement. We used the results of the usability testing to address several issues, making the framework easier to use for PRC project developers.

The framework we developed is simple, flexible, and scalable. It can be used with science applications written in many different languages, so project developers can use whichever language they are most comfortable with to develop PRC projects using our framework. SLINC supports a wide range of operating systems and architectures, as well as several different databases, allowing for greater flexibility in deployment. To make SLINC easier to use, we created step-by-step guides for developing PRC projects, tools to expedite and simplify the project creation process, and an example PRC project implementation to use as a reference.

We believe we have succeeded in developing a PRC framework that is easier to use than existing frameworks, yet offers the most important core functionality necessary for a production PRC system. SLINC does not support some of the advanced functionality available in other frameworks, but could be easily extended in the future to offer similar features.

2 Background

Public resource computing is a relatively new subfield of distributed computing and high performance computing, and as such there is comparatively little research in this area. There were two primary sources of information about public resource computing that we used frequently: Louis G. Sarmenta's Ph.D. dissertation[4] and a popular framework for public resource computing called BOINC[2].

2.1 Related Work

Louis F. G. Sarmenta is one of the leading researchers in the field of public resource computing. In his 2001 Ph.D. dissertation[4] he classified new and existing forms of PRC, designed and built a Java based PRC framework, and devised methods for reducing the impact of malicious volunteers. Sarmenta's PRC framework was called *Bayanihan*, which he created to test his contributions to PRC research. Bayanihan was different from previous PRC projects like SETI@home[5] and distributed.net[6] because volunteers did not have to manually download a special client to contribute their computers' resources. Instead, volunteers would visit a website that would automatically load and execute a Java applet[7] in the web browser to perform computations for a PRC project. Sarmenta showed that Bayanihan enabled PRC networks to be formed quickly because software installation was not required on the client machines. Another advantage of having an installation-free client was that users who did not have permission to install software could still contribute to a Bayanihan-based PRC project. Bayanihan improved security on the client machines because unsigned Java applets have restrictions on the types of operations they are allowed to execute[8], preventing a fake PRC client from damaging a volunteer's computer. Since Bayanihan was Java based, it allowed PRC developers to create a single client that could execute on many different platforms without modification.

Sarmenta made another significant contribution to the PRC field in his dissertation when he explored methods to detect and defend against malicious volunteers. A malicious volunteer is one who deliberately returns incorrect results to the server. One method to prevent malicious volunteers from sabotaging the research is to send the same work unit to multiple clients, and to only accept the result returned by the clients if a critical number of clients agree on the result. This method is called *voting*. Another approach is to have the server calculate the result for a particular work unit, send the same work unit to a client, and compare the result the client returns with the result the server computed. If the two results differ, the client can be assumed malicious and be removed from the network. This approach is called *spot-checking*. Sarmenta found that an even more effective and accurate method is to combine voting and spot-checking[9].

In addition to Sarmenta's work in PRC, several Major Qualifying Projects (MQPs) at Worcester Polytechnic Institute (WPI) have explored an approach to PRC that was similar to that of Bayanihan. These MQPs developed and evaluated a framework for PRC called Distriblets[10,11,12]. Like Bayanihan, Distriblets used Java applets as a means to execute computations on volunteers' computers. These volunteers would visit a web site, which would cause a Java applet to be downloaded into their web browsers. The applet would then perform any computations needed by the PRC project. Unique features of the

Distriblets framework included support for volunteers with non-persistent Internet connections and the ability for volunteers to choose which PRC project they would contribute their computer's resources to.

2.2 Berkeley Open Infrastructure for Network Computing (BOINC)

BOINC is a framework for public resource computing that is comprised of both server and client components[13]. The BOINC client itself uses very few system resources; it only acts as an interface between the BOINC server and the volunteers' computers. The application that uses system resources to perform computations for a PRC project is called the *science application*. The BOINC client requests work units from the BOINC server on behalf of the science application and returns the results computed by the science application to the BOINC server. The BOINC client and the science application communicate through the BOINC Application Programming Interface (API), which is written in C++.

The BOINC server is actually a collection of seven different daemon processes, or *server components*, some of which are developed by BOINC and others that each PRC project must implement individually[14]. The *feeder* and *transitioner* are components that are supplied by BOINC. The server maintains a queue of work units that need to be sent to the clients; the feeder retrieves newly generated work units from the database to fill this queue. The transitioner controls the state transitions of work units and results throughout their lifecycles. The lifecycle of a work unit begins when it is generated by the *work generator* and is added to the database. Work units can then go through several state transitions as they are distributed to one or more clients to be processed. For example, if a work unit is distributed to a client, but a result for that work unit is not returned within a predetermined amount of time, then that work unit is said to have *timed out* or *expired*. The transitioner detects work units that have timed out and redistributes them to different clients. The lifecycle of a work unit ends when enough valid results for that work unit have been collected and a single result, called the *canonical result* is chosen for that work unit. The lifecycle for a result begins when it is computed by a client and sent to the project server. There are several state transitions that can occur for results as well. For example, results that have just been received are in a different state than canonical results. The lifecycle of a result can end in three ways: it is determined to be invalid, it is marked valid and selected as the canonical result for a work unit, or it is marked valid and is not selected as the canonical result for a work unit. All results that are invalid or are not selected to be the canonical result are deleted.

The five remaining daemons need to be implemented by each PRC project. The *work generator* generates new work units to be computed by the science application. The *validator* attempts to determine which results are valid by comparing results from several different clients. The *assimilator* processes valid results, which usually means storing them to a separate database for later analysis. The *file deletion* and *database purging* daemons remove files and work units that are no longer needed.

The BOINC architecture is highly modular and scalable. If the project server becomes inundated with client requests, additional servers can be added to the project with daemons running on all of the servers, each handling only a fraction of the total

incoming requests. With a sufficient number of project servers, the only bottleneck in the system is the MySQL server[3].

2.3 Limitations and Disadvantages of BOINC

BOINC is a powerful and robust system for public resource computing, but it has some significant limitations. The BOINC client has been ported to several platforms, but the BOINC server can only be executed on Linux-based operating systems. Having support for only one platform imposes limitations on the operating environments for PRC projects and requires researchers to have experience with Linux system administration in order to create a new BOINC project.

With the power of BOINC comes great complexity. BOINC is a composite of several distinct applications, some of which have been created by the BOINC developers and others that are developed separately by each public resource computing project. To create a BOINC project one must first understand the interactions between the BOINC components. In addition to understanding the BOINC architecture, researchers creating BOINC projects must learn the BOINC programming API and be proficient in Linux system administration, MySQL[15] relational database administration, the Extensible Markup Language (XML), and the C++ programming language. Even though the BOINC system is quite complex, there is very little documentation available about how to create a BOINC project. This lack of documentation is perhaps the largest barrier that researchers face when creating a BOINC project. There are also very few tools to facilitate the creation of new projects, resulting in a long, manual process.

The BOINC system is well suited to large scale PRC projects, but the limitations and complexities of BOINC can be prohibitive factors for researchers interested in creating small to medium size PRC projects.

3 Thesis Goals

3.1 Create a Public Resource Computing Framework

The primary goal of this thesis was to create a framework to support research through public resource computing (PRC) that addressed the usability shortcomings we identified with frameworks like BOINC. Reflecting our commitment to usability, we named our framework the Simple Platform for Accessible Research Computing (SLINC). In order for SLINC to be an improvement over existing frameworks, it had to be more flexible and easier to use. SLINC had to simplify the process of creating a PRC project and also to decrease the amount of time necessary to do so. Additionally, the framework's performance had to scale reasonably well. SLINC was designed to be both modular and extensible so that future developers and researchers would be able to easily modify and extend the functionality of the system. This modularity also improved the scalability of the framework.

One of our goals for the implementation of SLINC was to create a cross-platform server and client that were compatible with the most common operating systems. The cross-platform property of SLINC contributed to our goals of flexibility and ease of use. The implementation of the framework used open standards wherever possible to increase interoperability and facilitate the addition of new features to the system. Lastly, the concept of scalability was extended from the architecture through the implementation using a combination of design patterns, features of the programming language, and third party libraries based on open standards.

3.2 Create a Public Resource Computing Project for Testing

Once the implementation had achieved sufficient functionality, our next goal was to create a simple PRC project, including both a server and a client. This example project allowed us to test the system in several different ways. We were able to test the basic functionality of the framework components, as well as the client-server communication. We also tested SLINC's ability to scale when multiple clients were accessing it simultaneously. The example PRC project allowed us to compare SLINC to BOINC in terms of the complexity of creating a project. Going through the steps of creating a project with SLINC also helped us to create better documentation about the process for researchers who will use the framework to create their own PRC projects.

3.3 Create Documentation and Tools

Our last goal was to make SLINC as easy to use as possible. We accomplished this goal mainly by creating thorough documentation, both in the code itself and in the form of user manuals, implementation guides, and interface specifications. We also developed tools to automate as much of the project creation and maintenance process as possible.

3.4 Major Contributions

Through achieving these thesis goals, we developed a public resource computing framework that was easy to use, scalable, and extensible. Researchers who need access to large amounts of computational power will be able to use SLINC to solve complex problems or modify the system for further research in public resource computing.

4 Design

This chapter describes the design of SLINC, including the initial requirements, important design decisions, tools and third-party libraries that were used, and the system architecture.

4.1 Requirements

The requirements for SLINC were inspired mainly by the disadvantages and limitations we identified in the BOINC system. We wanted our framework to be more flexible than BOINC, but above all we needed to design a framework that was easier to use.

4.1.1 Ease of Use

In order for SLINC to be a successful platform for public resource computing it needed to be easy to use, both for the developers of public resource computing projects and for the volunteers participating in the projects. In the context of SLINC we defined ease of use in several ways. Project developers, those who are responsible for programming the necessary applications, should be able to learn the interface to SLINC quickly. Additionally, the amount of project-specific code that needs to be written should be minimized. Configuring the framework for the specific public resource computing project should be made as simple as possible, for example through the use of a graphical application. Finally, volunteers should be able to easily install and configure the software on their personal computers.

4.1.2 Flexibility

In addition to being easy to use and learn, a public resource computing framework must be flexible enough to be used in a wide range of environments. Different research groups may have very different needs or restrictions in terms of operating systems and database management systems that they are able to use. We decided that SLINC must support some of the most commonly used operating systems and database management systems. In addition to flexibility in operating environments, we recognize that researchers are not all proficient in the same programming languages, and certain languages are better suited to research applications than others. Consequently, SLINC must support a variety of programming languages for the applications it will manage.

4.1.3 Extensibility and Maintainability

SLINC must be designed in such a way that it can be easily modified and extended. Defects may be found at a later time, or certain assumptions about the operating environment may change, so the source code must be organized and well documented to facilitate corrective and adaptive maintenance. Since new features may be required in the future, perfective maintenance must also be considered in the design of

the system. The design of SLINC should be modular and make use of open standards and libraries to make the system easier for future developers to learn and extend.

4.2 Initial Design Decisions

This section describes important design decisions we made before beginning any architecture design or implementation. It was necessary to decide what programming language to use for the framework, what programming languages the framework would support, what type of inter-process communication to use, and which database management systems to support.

4.2.1 Programming Languages

Our requirement that the system be as flexible as possible (see Section 4.1.2 Flexibility) greatly influenced our decisions about the programming languages we would use. The programming language for the framework had to be portable and preferably execute on different operating systems and architectures without modification; we wanted to use a language that we were already familiar with so that implementation could begin quickly; we needed a language that had a variety of third-party tools and libraries available to decrease implementation time. For these reasons we decided to implement the public resource computing framework in Java.

The other programming language decision we had to make was which languages we should allow researchers to use when creating public resource computing projects with SLINC. We decided that at a minimum we should support C++ and Java, due to their ubiquity and frequent use in public resource computing, and other common languages if possible.

4.2.2 Inter-Process Communication

The mechanisms used for inter-process communication (IPC) have a significant impact on the architecture of a system, so we needed to make early decisions about how the different components that comprise the framework would communicate. In keeping with our requirements (see Section 4.1 Requirements), we wanted to use a form of IPC that was based on open standards, was portable, and was implemented in a wide variety of languages. We considered using standard TCP sockets, but that method can be prone to error, especially in languages that do not perform array bounds checking like C and C++. Java Remote Method Invocation (RMI) is a good remote procedure call (RPC) implementation, but it is limited to use in Java, so it would make SLINC less flexible.

One type of IPC that seemed appropriate was XML-RPC[16]. XML-RPC has a simple and open specification; it uses HTTP as its transport protocol and XML as its encoding method, which are both open standards maintained by the World Wide Web Consortium (W3C)[17]. There are implementations of XML-RPC for many languages, including C, C++, Java, Perl, PHP, Python, Scheme, Ruby, ASP, and others[16]. Due to the number of popular programming languages that have XML-RPC implementations, the use of XML-RPC makes SLINC quite flexible. Researchers who develop applications

for SLINC will have the freedom to choose the programming language that they are most familiar with.

4.2.3 Database Support

All public resource computing systems need to store data. Typically, the storage needs of a PRC project include configuration data, work units, results, and information about users. For a large project, storing all of this data to a simple flat file would be much too inefficient. There would be too much latency in searching through such a file for relevant data, and even file size limitations of the file system might become a limitation. For these reasons, systems like BOINC use external databases to store project data. The problem with this approach is that an external database is usually not necessary for a small project because the amounts of data are small enough to be stored in regular files, and the use of a database makes creating and configuring a project more complex.

We wanted SLINC to be easy to use for small projects, powerful enough to support large projects, and flexible enough to be used in many different environments. SLINC supports several different types of database systems to accommodate the needs of different users and projects. Small projects can use an embedded Java database called HSQLDB[18], which saves data to a normal file on disk. Using HSQLDB is easier because the project developers would not have to install and configure a separate database management system, and small projects probably do not need the power of a full database. Larger projects have the option to use a normal database management system. We decided to provide support for some of the most popular databases, such as MySQL[15], PostgreSQL[19], Oracle[20], and Microsoft SQL Server[21].

4.3 Tools

There were several tools we used extensively throughout the design and implementation phases. This section describes the tools we used, why we used them, and how they facilitated our work.

4.3.1 SourceForge

SourceForge[22] is a software development management system. It provides software developers with a complete management system intended for use with medium to large software projects. Developers who use SourceForge have access to version controlled source code and document repositories. They can set and track development goals, post code releases, make project announcements, and perform other useful tasks. WPI has its own SourceForge system located at http://sourceforge.wpi.edu, and our SourceForge project is at http://sourceforge.wpi.edu/sf/projects/jdb_prc_thesis. We use SourceForge primarily for its code repository and for storing documentation. The code repository allows us to access the source code for the PRC framework from anywhere on the Internet. The document repository provides a central location for us to post documentation related to the project.

4.3.2 Eclipse

Eclipse[23] is an open source integrated development environment (IDE). It is used primarily for developing Java applications, but it can be used with other languages as well. Eclipse has many features that can accelerate the code development process, including integration with code repositories like CVS and Subversion, code completion, integrated debugging capabilities, and automated building of executables. We used Eclipse to implement, test, and debug all of the Java code related to our PRC framework. One of the most useful features of Eclipse is its integration with JUnit.

4.3.3 JUnit

JUnit[24] is a framework for performing unit tests on Java code. In software engineering terms, JUnit acts as a *driver* for testing small components of a Java project. JUnit tests allow developers to execute methods, and then make assertions about certain conditions that should be true if the tests were successful. Any number of assertions can be made, and a test is only considered successful if all assertions were correct. Groups of related JUnit tests can be organized into JUnit test suites, which when executed will run all of the JUnit tests in the suite. The JUnit integration in Eclipse facilitates the creation of JUnit tests and test suites. Eclipse allows developers to easily run JUnit tests and to quickly see the results of each test.

4.3.4 Abeille Forms Designer

Abeille Forms Designer[25] is an open source application that facilitates the creation of Java graphical user interfaces (GUIs). It has an editor that allows users to create forms by dragging user interface components onto a graphical representation of a form. Abeille uses the standard Java Swing graphics framework in addition to the JGoodies[26] libraries, which provide advanced layouts and GUI widgets. Using the Abeille Forms Designer application allowed us to rapidly develop user interfaces for our public resource computing system.

4.3.5 Apache Ant

Apache Ant[27] is a powerful build tool for Java projects. Its functionality is similar to that of GNU Make, but Ant is a cross-platform tool. It is written in Java and uses XML-based build configuration files. During the development of SLINC we used Eclipse to manage the builds, but when development was completed we used Eclipse to generate Ant build files for the framework. We modified those build files so that we could use Ant to package our framework into Jar, ZIP, and bzip2 files to make it easier for project developers and volunteers to use. The scripts that package a new public resource computing project into client and server archives simply invoke Ant with custom targets we wrote for these purposes.

4.4 Third-Party Libraries

We used several third-party libraries in our public resource computing framework. These libraries increased our productivity by providing critical functionality for XML-RPC marshalling and transport, database access, and GUI design, allowing us to focus on implementing the framework itself. This section describes each of the libraries we used and their licenses.

4.4.1 XML-RPC Implementations

We used two different libraries to enable XML-RPC communication between processes: Apache XML-RPC[28] and XML-RPC for C and C++[29]. Apache XML-RPC is a Java implementation of the XML-RPC specification. It supports XML-RPC clients and servers. Apache XML-RPC is released under the Apache Software License, Version 2.0.

As its name suggests, the XML-RPC for C and C++ library is an implementation of the XML-RPC specification for the C and C++ languages. It contains a separate library for each language, but both versions of the library support XML-RPC clients and servers. The XML-RPC for C and C++ library is released under a BSD-style license.

SLINC is written entirely in Java, so only the Apache XML-RPC library is used in the SLINC code. However, we wanted to provide both C++ and Java examples of public resource computing projects that use our framework, so we needed the C++ XML-RPC library for the C++ example project.

4.4.2 Hibernate

Hibernate[30] is a Java library that serves as an abstraction layer for database access. It provides the ability to persist objects to a number of different relational database management systems. Perhaps the most useful feature of Hibernate is that it allows developers to write database-independent code. The same source code can be used to store objects to many different types of databases. Changing the type of database that is used is as simple as using a different configuration file or reconfiguring Hibernate using API calls. We used hibernate for all database access, and it allowed us to quickly and easily store and retrieve our objects from various database management systems. Hibernate is released under the GNU Lesser General Public License.

4.4.3 Abeille Forms and JGoodies

We used the Abeille Forms Designer[25] (see Section 4.3.4 Abeille Forms Designer) to create our GUIs. Two libraries are required in order to display these GUIs correctly: the Abeille Forms runtime library and the JGoodies Forms[26] library. Both of these libraries are released under the BSD license.

4.5 Architecture

At a high level, some aspects of SLINC are similar to that of BOINC, but there are many differences. This section explains the architecture of SLINC and justifications for our architectural decisions.

4.5.1 Overview

SLINC consists of several distinct components, similar to the way the BOINC framework was designed. Also like BOINC, some of these components are provided by SLINC, but others need to be implemented by the creators of a public resource computing project. The components are classified into server-side and client-side components. All components that are separate processes communicate via XML-RPC (see Section 4.4.1 XML-RPC Implementations), an open specification for XML-based remote procedure calls. The server-side components are designed to be run on machines maintained by the project administrators, and the client-side components are meant to run on the computers of people who choose to contribute to the project, referred to as *volunteers*.

The server-side components are responsible for maintaining the project database, partitioning input data into work units, distributing work units to clients, and processing and validating results for each work unit. There is a single, central server application for each project that all clients connect to. The client-side components request work units from the project server, compute the result for each work unit, and return the result to the server. There is exactly one client that runs on each volunteer's computer and communicates with the central project server.

4.5.1.1 Tasks: Work Units and Results

In the context of SLINC, a task refers to either a work unit or a result. A work unit is a subset of the total amount of a project's input data that needs to be processed. Usually each work unit is a very small subset of the total amount of work that needs to be done so that an average computer can complete all necessary computations for that work unit in a reasonable amount of time. However, decisions regarding work unit size, content, partitioning, and other work unit properties are left to the creators of each project. The output data generated by processing the work unit is called a result. All tasks are stored as persistent Hibernate objects in the project database.

We have included several work unit properties to give project creators greater flexibility in the way that work units are stored and distributed to clients. The *priority* property controls the order in which work units will be distributed. Work units have a default priority of zero, and work units with higher priorities are guaranteed to be distributed before work units of lower priorities. Work Units are stored in a sorted queue, so if two work units have the same priority, normal first-in-first-out ordering applies. The priority of a result will be the same as the priority of the work unit from which it was computed.

Each work unit has a *point value* associated with it. When a volunteer returns a result for a work unit, the volunteer's score is increased by that work unit's point value. The point value is a way to weight the value of each work unit and reward volunteers in a fair way if all work units are not expected to take the same amount of time or resources to process.

When a work unit is distributed to a volunteer, it is described as being *assigned* to that volunteer, and a work unit can be assigned to more than one user for redundancy and validation (see Section 4.5.1 Overview). The *expiration time* property represents the amount of time each volunteer has to return a result for his or her assigned work unit. If insufficient results have been received for a work unit before the expiration time has elapsed, the work unit is said to have *expired*. When a work unit expires, its expiration time is reset, and the work unit will be assigned to a different user.

4.5.1.1.1 Task States

We designed the work units to be flexible by providing several configurable properties (see Section 4.5.1 Overview), but this flexibility introduced complexity. In order to manage this complexity, we realized that we would need to assign several states to each work unit, and these states would change as work units were created, distributed, and completed. We also knew that some projects would want to have a mechanism for validating results returned by clients, and that result tracking could also be facilitated by assigning states to results. Fortunately, we were able to design a set of states that could be used for both work units and results. All tasks must have one of the four following states: ingress, pending, egress, or spot-check. The spot-check state is a special case that is explained in Section 4.5.1.5.2 Spot-Checking, but every task will progress through each of the other three states during its lifecycle.

4.5.1.1.2 The Task Lifecycle

Figure 1 and Figure 2 depict the state transitions of tasks.

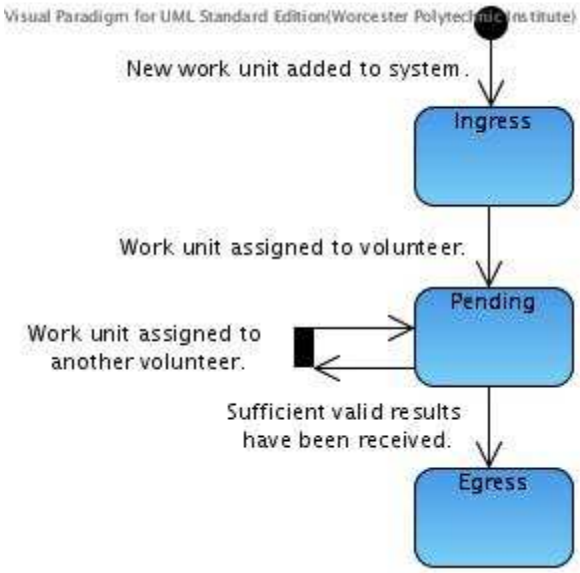

Figure 1: Work Unit States

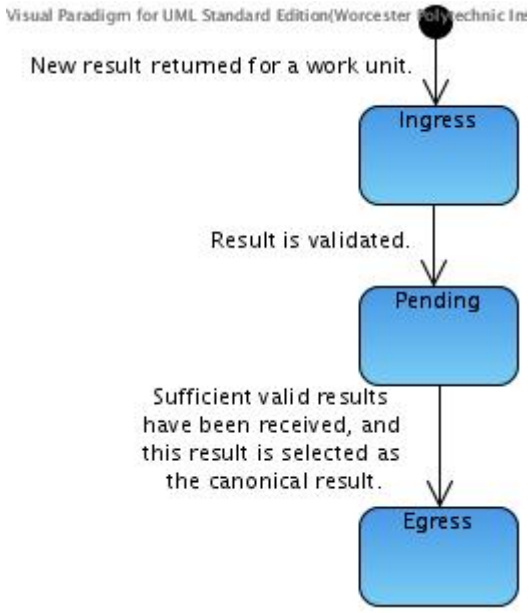

Figure 2: Result States

All tasks begin in the ingress state, transition to the pending state, and end at the egress state. Tasks can never make any other state transitions (e.g. egress to pending) except in one exceptional scenario explained in Section 4.5.1.5.3 Saboteur Mitigation Strategies. When a new work unit is generated and added into the system its state is set to ingress. Similarly, when a new result is returned by a client its state is set to ingress. After an ingress work unit has been assigned to a volunteer, the work unit's state is changed to pending. While the number of valid (i.e. pending) results for a given work unit is less than the minimum number of valid results defined in the project configuration, all valid results for that work unit remain in the pending state.
When the minimum number of valid results for a work unit has been received, one of the results is chosen to be the *canonical result*. The canonical result is the result that is accepted for the work unit, and it is the only one that is saved. All other results for that work unit are purged. The process of selecting a canonical result is explained in more detail in Section 4.5.3 Server Components. Once the canonical result is selected, the canonical result's state is changed from pending to egress, and then the work unit from which the canonical result was computed is transitioned to the egress state. All tasks are stored in memory as well as in the project database except egress tasks, which are only stored in the database to conserve memory.

4.5.1.2 Projects

Each public resource computing project is likely to have different requirements, and each could be installed in a different environment. We came to the conclusion that there was a need to store certain project-specific configuration options. Some of these properties include the name of the project, the locations of important project files, the uniform resource locators (URLs) of the server components, and the project database configuration. For a complete list of configuration options, please see Appendix B:

Project Creation Guide. The project configuration is stored to the project database as a persistent Hibernate object.

One important project property that relates to work unit distribution is the *number of valid results (NVR)* property. The NVR property controls the number of times each work unit is distributed. For purposes of redundancy and validation (see Sections 4.5.1 Overview and 4.5.1 Overview) it is often necessary to assign the same work unit to multiple volunteers. In this case, a work unit cannot be considered complete until the specified number of valid results is received for that work unit, each computed by a different volunteer.

4.5.1.3 Client-Server Model

The well-known client-server model appears in several places in our architecture. SLINC includes several processes that communicate via XML-RPC, so it is a natural distinction to designate a server and a client for each interaction. Usually the process that is running an XML-RPC server is referred to as the server. We tried to minimize the number of server processes and processes that are both clients and servers in the framework to decrease the complexity of the system and avoid the introduction of faults. In particular, we wanted to eliminate the need for any component implemented by the project creators to be a server to minimize the amount of work involved in developing a public resource computing project. However, doing so actually increased the complexity of the design in some cases. For this reason one project-specific component, the science application (see Section 4.5.4 Client Components), must be a server as well as a client.

4.5.1.4 Framework Components and Project-Specific Components

Several software components are required to create a public resource computing project using SLINC. Some of these components are provided by the framework, but others must be written specifically for each project. At a minimum, the creators of a public resource computing project must implement a work unit generator and a science application. The work unit generator creates new work units and sends them to the project server to be distributed to clients. The science application takes a work unit as input, performs all necessary computations on that work unit, then returns a result as output. A third optional component for the project creators to implement is the result validator. If the project creators do not want or need result validation, they can choose to use the default validator provided by the framework. All other components are provided by SLINC, including the project server, the transitioner, the spot-check generator, and the client. Each of these components is described in more detail in Sections 4.5.2 Component-Based Architecture, 4.5.3 Server Components, and Appendix B: Project Creation Guide.

4.5.1.5 Result Validation

An important difference between PRC systems and traditional distributed systems is that PRC project administrators do not have control over the entire system. Most computers that contribute to PRC projects are privately maintained, so the operators of PRC projects cannot completely trust the results returned by their users. Some distributed algorithms, such as those used in machine learning[31], can be very sensitive to

erroneous information. For this reason it is important for a PRC project to be able to validate the results it receives.

When designing a PRC system one must also consider the possible presence of malicious users, or *saboteurs*. Saboteurs may deliberately inject incorrect information into the results they return. Sarmenta proposed[32] two methods for validating results and detecting saboteurs: voting and spot-checking.

4.5.1.5.1 Voting

Voting helps prevent cheating by delaying the acceptance of a result until a critical number of matching results is received from different users. A matching result can be defined in several ways depending on the application domain. For a certain distributed application a pair of results might be matching if they are equal at the byte level. For another distributed application a pair of results could be matching if they evaluate to floating-point values that are within a certain number of standard deviations of each other. The voting technique is equally valid regardless of the operation used to test for equality.

Sarmenta showed[32] that linearly increasing the critical number of matching results exponentially reduces the incidence of error in the results. This dramatic decrease in the error rate comes at the price of the work completion rate. The amount of redundant work that needs to be completed in order to compare results can greatly slow the overall progress of a project, so a balance must be made between integrity and speed.

4.5.1.5.2 Spot-Checking

In the spot-checking technique, a computer controlled by the PRC project first computes the result for a work unit, and this result is assumed to be valid. The same work unit is then sent to users with a certain probability. The result returned by the user is then compared to the result that was computed by the project. These two results can be compared in a number of ways as described in Section 4.5.1 Overview. If the result computed by the project and the result computed by the user do not match, that user is likely a saboteur.

Work units and results involved in spot-checking are assigned a special task state to separate them from normal tasks. This task state is called the spot-check state. Work units that are in the spot-check state are assigned to clients much less frequently than work units in the ingress or pending states because it would be inefficient to spot-check a client frequently. Section 4.5.3.5 describes how spot-checks are performed in more detail. The spot-check state is similar to the egress state in that spot-check work units and results are at the end of their lifecycle and can be considered valid.

Spot-checking is more efficient than voting because a spot-check only needs to be performed once. The spot-check could be performed regularly, perhaps after every one-hundred work units, but the wasted work due to spot-checking would still be far less than the work that is wasted by the redundancy of voting. However, spot-checking is not as effective as voting at reducing the error rate in results because spot-checking only tests a single user[32]. Linearly increasing the probability that a user will receive a spot-check causes a linear decrease in the error rate.

4.5.1.5.3 Saboteur Mitigation Strategies

Once a saboteur has been detected there are several actions a PRC project can take to mitigate the damage that saboteur may have caused or may cause in the future. One possible action is to simply write identifying information about that user to a log file for review later. This action would be appropriate if the distributed application were not sensitive to incorrect data or if the methods used to detect the saboteur were known to be imprecise. Another possible action is to ban the identified saboteur. This action will prevent the saboteur from sending any more incorrect data, but the saboteur could have already injected many errors into the results. A more severe action is to not only ban the saboteur, but to also invalidate all results that user has previously returned. A consequence of invalidating the saboteur's results is that those results will need to be recomputed by another user, creating more work for the other users in the system and slowing the progress of the project.

4.5.1.5.4 Design Decisions

Each PRC project will likely have different needs in regard to result validation and saboteur tolerance. Some researchers may need to create a robust PRC project that maximizes the integrity of the results, while others may simply need to start a PRC project as quickly and easily as possible. For these reasons we decided to allow the PRC project creators to decide the amount of result validation and saboteur tolerance that will be used by their project. Any combination of voting and spot-checking may be used, and both the critical number of results for voting and the spot-check probability for spot-checking may be defined by the project creators. For researchers who need to quickly create a project with minimal effort, they can choose not to use any validation or spot-checking. When spot-checking is used and a user fails a spot-check, the failure is always logged. However, the project creators can choose to take further actions including banning the user and optionally invalidating all of that user's prior results. The flexibility of our approach allows large projects to reduce the error rate in the results while giving smaller projects the ability to get started quickly and with minimal effort.

4.5.2 Component-Based Architecture

Although many architectural aspects of SLINC are different from BOINC, we recognized that BOINC'S partitioning of server-side and client-side functionality into several distinct processes, called components, has many advantages. Component-based systems can be easier to maintain if the components are divided in a logical, organized way. In this model dependencies between components are analogous to class dependencies, so if each component has a well-defined set of responsibilities and interfaces to other components, that component can be replaced without impacting other parts of the system.

Our choice to use XML-RPC as the interface between components in SLINC has two additional consequences: language independence and location independence. Since there are implementations of XML-RPC in many languages, each component could be implemented in a different language if such a design decision would be beneficial to the framework. XML-RPC uses HTTP as its transport, which means that any two computers that can communicate across a network can also make XML-RPCs to each other. The implication is that our component-based architecture is inherently distributed. Each

component can execute on a different computer, reducing the average load on each computer in the system.

4.5.3 Server Components

There are five components that together provide all of the necessary server functionality. These components include the project server, work unit generator, the result validator, the spot-check generator, and the transitioner. Each of these components can be executed on a separate computer with the exception of the spot-check generator, which must be located on the same computer as the project server. Figure 3 shows the components in the SLINC framework.

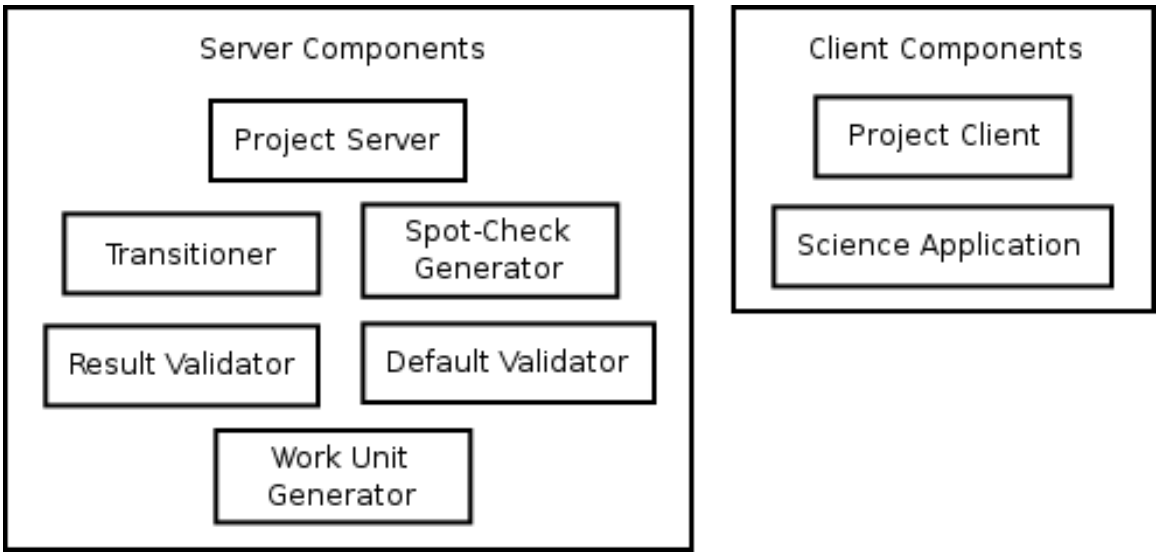

Figure 3: SLINC Components

4.5.3.1 Project Server

The project server is a component provided by the framework that coordinates all of the server components. The primary responsibilities of the project server are to handle XML-RPC requests from the project clients and to act as a proxy between the project database and the other server components. When it is first started, the server initializes Hibernate and reads the project configuration from the project database. The server uses this information to configure and initialize the other server components. The first component the server creates is either a local transitioner or a remote transitioner (see Section 4.5.3 Server Components), based on the project configuration. Following the creation of the transitioner, the project server initializes and starts its XML-RPC server. After the XML-RPC server has been started, the server is completely initialized and can begin processing requests from other components.

Once the server has been started its primary responsibility is to handle XML-RPC requests from other components. These requests include adding new work units to the system, distributing work units to clients, accepting results from clients, adding new volunteers to the project, and several other requests necessary for the system to function. A full specification of the server's XML-RPC interface can be found in

Appendix D: XML-RPC Interface Specification. In handling these XML-RPC requests the server coordinates the actions of all of the components in the system, including both server and client components. Although the server may seem simple compared to other components, the services it provides are critical to the basic functionality of the system.

Although most of the project server's responsibilities are simple, one service the project server provides that is somewhat complex is the *getResultForValidation* remote procedure call. The result validator (see Section 4.5.3 Server Components) uses this RPC to request the next result that needs to be validated. There are three types of validation that can occur: the validation of a single result, the validation of a single spot-check result, or the choice of a canonical result. The server must decide which validation should occur each time the RPC is invoked. Each of these three types of validation has a flag associated with it so that the result validator can determine which type of validation needs to be performed. The server first queries the transitioner to determine whether there are any groups of results for which a canonical result must be selected. If there are, the server always instructs the result validator to select a canonical result so that the other results can be deleted, and the appropriate work unit and result pair can be retired. In this case, the server sends all of the related results to the result validator so that they can be compared. If there are no canonical results that need to be selected, the server tries to retrieve any other results that need to be validated from the transitioner.

4.5.3.2 Work Unit Generator

The work unit generator is a server component that must be implemented separately by each public resource computing project that uses SLINC. The purpose of the work unit generator is to partition a PRC project's raw data into work units that can be sent to clients to be analyzed by the science application (see Section 4.5.4 Client Components). Since work partitioning is clearly a domain-specific problem, this component must be implemented by the PRC project developers. For this reason we have tried to make the implementation of the work unit generator as simple and flexible as possible.

At a minimum the work unit generator must invoke one XML-RPC that sends the work unit data to the project server. There are several versions of this XML-RPC that allow the project developers to have greater control over the work units. The various overloaded XML-RPC handlers for adding a work unit to the system allow developers to specify different combinations of the work unit's identification string, data, expiration time, priority, point value (see Section 4.5.1 Overview). In the simplest case, only the work unit data needs to be sent, and the rest of the parameters assume default values. However, the developers have the option to specify the other parameters if greater control is needed over the work units. When the work unit generator sends a work unit to the project server, the server replies with the number of volunteers who are currently waiting for work units. A volunteer is considered to be waiting for a work unit if that volunteer has requested a new work unit when the server had none to distribute. The number of users waiting can give the work unit generator an indication of how many additional work units need to be generated.

Although only one XML-RPC is required for the work unit generator to function, there are two other XML-RPCs the work unit generator can invoke that are quite useful. The server has an XML-RPC handler that will return the number of ingress work units

(see Section 4.5.1 Overview). The work unit generator can use this XML-RPC as a way to determine whether any additional work units need to be generated without actually sending a work unit to the server.

Another important XML-RPC will return the last work unit that was generated. This functionality is useful when the work unit generator is restarted after being terminated for any reason. By examining the last work unit that was generated, the work unit generator can recover its state without the need to read or write any temporary files. For example, suppose that the work unit generator is designed to generate sequential work units that consist of a single positive integer, starting from one. Furthermore, suppose that after generating five work units a careless project administrator accidentally kills the work unit generator process. Without any way of recovering its state, when the work unit generator is restarted, it would have to start generating work units from one, introducing duplicate work units into the system. However, when the work unit generator is restarted, it can query the server for the last work unit that was generated. The server will reply with the work unit containing the number five. The work unit generator can then recover its state and resume generating work units, starting with the number six. Of course it is possible to save and recover state by writing and reading temporary files, but the XML-RPC method is less error prone because it has already been tested, and by using the XML-RPC the writers of the work unit generator do not need to worry about details such as file permissions. Detailed information about XML-RPCs the work unit generator can invoke can be found in Appendices C and D.

4.5.3.3 Result Validator

The result validator has two responsibilities: to determine whether an individual result is valid and to select the canonical result for a work unit. Each result returned for a work unit is first validated individually. The validator evaluates both ingress results and spot-check results, which are results that are computed by a client for a spot-check work unit. When a sufficient number of results has been validated for a given work unit, the validator will select a single valid result for that work unit to be the canonical result. The canonical result and its work unit are then retired, and all other results for that work unit are deleted. The act of selecting a canonical result from a group of related results is part of the voting technique described in Section 4.5.1 Overview.

The validator can be run from any computer because it interacts with the project server via an XML-RPC interface. The validator connects to the project server periodically, checking for any results that need to be validated. If the server has results that need to be validated, it will respond with the type of validation that needs to be done and the necessary result data. The three types of validation that can occur are validating an individual result, selecting a canonical result, and validating a spot-check result. The validator will decide whether the result is valid and return a Boolean value indicating its decision. If the type of validation was selecting a canonical result, the validator will also send the task ID of the canonical result it selected to the project server via XML-RPC.

The result validator requires specific information about the PRC project in which it is used to function correctly, and so it must be implemented separately for each PRC project by the project developers. If the project creators do not want to implement a result validator, the project can use the default validator, which automatically marks all results as valid and selects the first result it finds as the canonical result. If validation is

not important to the creators of a public resource computing project, then the default validator can reduce their development time. Information about implementing a result validator can be found in Appendix C: Project Programming Guide.

4.5.3.4 Spot-Check Generator

The spot-check generator creates results used in the spot-checking technique described in Section 4.5.1 Overview. The spot-check generator runs in a separate, low-priority thread in the project server. Its only responsibility is to invoke the science application (see Section 4.5.4 Client Components) in order to compute the result for a spot-check work unit. This result is assumed to be valid, and its state is set to spot-check.

The spot-check generator is the only server component that cannot be run on a separate computer from the project server. One reason for this limitation is security. If any computer could run the spot-check generator, then it would be possible for a saboteur to intentionally generate invalid spot-check results and inject them into the system. Another reason why there is no need to run this component on a separate computer is that its task is very simple. The spot-check generator will usually only run once in the lifecycle of the project. After generating the requested number of spot-checks the generator will simply shut down. Also, since the spot-check generator runs in a low-priority thread, it does not consume many system resources, so distributing this component would not significantly reduce the load on the project server.

4.5.3.5 Transitioner

The transitioner is one of the most complex components in the entire framework. It keeps track of the state of every work unit, result, and volunteer in the project. The transitioner distributes work units to clients, receives results from clients, controls the spot-check generator, takes action against identified saboteurs, and performs many other common tasks.

The primary responsibility of the transitioner is to manage every task in the system. It accomplishes this by maintaining seven task queues, each of which is sorted by task priority. All tasks that are in the ingress, pending, or spot-check states are stored in these queues as well as in the project database. Once a task reaches the egress state, it is no longer managed by the transitioner or any other server component. Egress tasks are stored only in the database for later retrieval and analysis by the researchers who created the public resource computing project.

There is both a work unit queue and a result queue for each of the three states managed by the transitioner: ingress, pending, and spot-check. We will refer to these queues using a consistent naming convention. The queue will be named by the state of the tasks in the queue followed by the type of tasks in the queue. For example, the queue that stores work units that are in the ingress state will be referred to as the ingress work queue, and the queue that stores results that are in the pending state will be called the pending result queue. The seventh queue, called the canonical work queue, stores work units for which a sufficient number of valid results have been received as specified by the project configuration. When a work unit is in the canonical work queue it means that enough valid results for that work unit have been received for a canonical result to be selected. The purpose for maintaining this separate queue is that it simplifies and

optimizes the process of identifying those work units for which a canonical result can be selected.

4.5.3.5.1 Work Unit Distribution

The distribution of work units is one of the most important and fundamental functions of the framework. It is critical that work units are distributed correctly and consistently, so the transitioner takes into consideration several factors when determining which work unit will be assigned to a volunteer. Regardless of which work unit is chosen to be distributed, we are assured that it is the highest priority work unit that is available because the work queues are ordered first by priority, then by the traditional first-in-first-out ordering.

The transitioner first decides whether the volunteer will be given a normal work unit or a spot-check work unit. It does so by choosing a random float in the range of zero to one, and then compares it to the project settings. If the random number is less than or equal to the spot-check probability property of the project, then that volunteer will receive a spot-check work unit.

If the transitioner did not assign a spot-check work unit, it will then search through the pending work queue. Starting from the beginning of the pending work queue, the transitioner will search for a work unit that was not previously assigned to the volunteer who is requesting a work unit. If such a work unit is found, the transitioner examines the number of times that work unit has previously been assigned. Since there is always a minimum number of results that must be received for each work unit, work units must be assigned to volunteers at least that minimum number of times. If the work unit that the transitioner is currently examining has been assigned at least the minimum number of times required by the project, then the transitioner will skip that work unit. In this case the transitioner optimistically assumes that the work unit will be completed before it expires. A work unit is said to have expired if enough time has passed such that the current time is later than the work unit's expiration time. If a pending work unit has been assigned enough times but has expired, the transitioner will reset the work unit's expiration time and begin assigning it to other volunteers. The purpose of having the expiration time is so that one work unit is not assigned many more times than is necessary. Instead, it is only assigned a certain number of times every expiration period until enough valid results have been received. If the transitioner cannot find a suitable work unit in the pending work queue, it proceeds to the ingress work queue.

The criteria for selecting an ingress work unit is much simpler than for selecting a pending work unit. If there are one or more work units in the ingress work queue, the transitioner selects the work unit at the head of the queue and assigns it to the volunteer who is requesting a work unit. However, if no work units were selected from the spot-check work queue, the pending work queue, or the ingress queue, the transitioner was not able to find a suitable work unit for the volunteer. In this case an exception is thrown, and the client requesting a work unit on behalf of that volunteer waits for a certain amount of time before requesting a work unit again.

4.5.3.5.2 Result Processing

When a client returns a result for a work unit, that result is passed to the transitioner for processing. The transitioner first increases the score of the volunteer who

computed the result. The volunteer's score is increased by the point value of the work unit for which the result was computed. Since scores are updated before the results are validated, it would be trivial for a volunteer to artificially inflate his or her score simply by sending many results containing arbitrary data to the project server. This possibility exists, but it is not a major concern for us because preventing these types of exploits was not one of our goals. Furthermore, since the framework was designed for small- or medium-sized projects, this type of tampering is unlikely to occur. However, there is a mechanism in the project server that is designed to prevent certain attacks of this type. When a client returns a result to the server, the server performs several checks before passing the result to the transitioner. It first verifies that the volunteer who is returning the result exists as a member of the project. It then checks whether that volunteer has been assigned to the work unit for which he or she is returning a result. A third check it performs is to verify that the data portion of the result is not null. If all three checks pass, the project server passes the result to the transitioner. The result sanitization performed by the server reduces the number of ways in which results can be falsified.

After the transitioner increases the volunteer's score, it checks whether the work unit associated with that result is already in the egress state. If it is, there is no reason to keep the result that was just returned, so the new result is deleted immediately. If the work unit associated with the new result is in the ingress, pending, or spot-check states, the new result is added to the ingress result queue, and its state is set to ingress. This result will eventually be validated by the result validator (see Section 4.5.3 Server Components).

4.5.3.5.3 Spot-Check Generator Control

The transitioner manages all tasks, including spot-check work units and results. The transitioner's responsibilities include starting the spot-check generator thread when necessary. The spot-check generator thread is started by the transitioner when the transitioner is first instantiated. The thread is only started if the number of spot-check work units or spot-check results in the system is less than the minimum number of spot-checks specified in the project configuration. When the spot-check generator thread is started, it attempts to acquire a work unit from the ingress work queue. If an ingress work unit is available, the thread changes that work unit's state to spot-check and adds it to the spot-check work queue. It then invokes the science application and passes it the spot-check work unit. The result returned by the science application is added to the spot-check result queue, and its state is set to spot-check.

4.5.3.5.4 Saboteur Response Strategies

When a saboteur is detected by failing a spot-check, there are three ways in which the transitioner can be configured to respond, which are also discussed in Section 4.5.1 Overview. The least disruptive method is to just log the fact that a volunteer has failed a spot-check. Regardless of the way the project is configured, spot-check failures will always be logged. Another possible response is to ban the identified saboteur. Volunteers are associated with work units, so simply deleting the volunteer from the database would result in inconsistencies. Instead, a Boolean flag is set in the properties of the volunteer indicating that the volunteer is banned. If a banned volunteer tries to request a new work unit or return a result, the project server will ignore the request. The

third possible action is to both ban the volunteer and invalidate all results which have been returned by that volunteer. This action can possibly have a large impact on other tasks in the system. If the results that are deleted are in the ingress or pending states, they can safely be deleted, but the situation is worse when a result from the saboteur has been selected as the canonical result. In this case, the result and its associated work unit have been retired. That is, they are both in the egress state. If this scenario occurs, the transitioner has to delete the canonical result and bring its associated work unit out of retirement. That work unit is placed back in the ingress work queue, and it will have to be computed again.

4.5.3.5.5 Local and Remote Transitioners

Like most of the server components, the transitioner can run on the same computer as the project server or on a remote computer. When configured to run on the same computer as the project server, the project server directly instantiates a *local transitioner*. When the project is configured to use a transitioner running on a separate computer, the server instead instantiates a *transitioner proxy* which forwards all requests over XML-RPC to the *remote transitioner*. The remote transitioner is a process that contains a local transitioner and an XML-RPC server. The remote transitioner's XML-RPC server interprets the requests from the transitioner proxy and passes each one to the appropriate method of the local transitioner.

There are not many circumstances under which a project should use a remote transitioner. The use of a remote transitioner will probably increase the load on the project database because a greater number of connections will have to be established. Network load will also increase due to the extra traffic caused by sending all transitioner requests via XML-RPC to the remote computer. Finally, it is not possible to use the HSQLDB database together with a remote transitioner because HSQLDB only allows a single process to access the database at any given time. The use of a remote transitioner may, however, slightly reduce the load on the project server. Given these facts, the only case in which a project should use a remote transitioner is when network and database load are not issues, but the project server exhibits poor performance.

4.5.4 Client Components

There are two framework components that are required for the client-side of the system to function: the project client and the science application. The project client is provided by SLINC, but the science application needs to be implemented separately by each public resource computing project.

4.5.4.1 Project Client

The project client acts as a layer of abstraction between the project server and the science application. The client provides services to reduce the complexity of the science application. The client parses the project configuration file to determine the location of the project server and other important information. It also collects and stores volunteer information so that the science application only has to implement the functionality necessary to compute results for work units. The client requests work units from the project server on behalf of the science application. When the client has received a work unit from the user, it will start the science application if it is not already running. After

sending the work unit to the science application, the client will wait for the science application to compute the result. After the science application has computed the result, the client will send that result back to the project server for processing.

The client also provides check-pointing services to the science application. Check-pointing is a way to save temporary state information in the event that the client is shut down before computation of a work unit is complete. The client provides an XML-RPC interface for the science application to use in order to store and retrieve check-point data. The science application only needs to send and receive XML-RPC data, but the client actually stores that data to disk and retrieves it when requested by the science application.

4.5.4.2 Science Application

The science application is the component that takes a work unit as input and produces a result as output. This component is the one that actually performs the computations needed by the project. Unlike the other project-specific components like the result validator and work unit generator, the science application requires the use of an XML-RPC server in addition to using an XML-RPC client. This restriction unfortunately may make the science application slightly more difficult to implement for project developers who are not familiar with XML-RPC, but we decided that including an XML-RPC server in the science application was necessary for two reasons. The first reason was that we needed a way to be notified if the science application terminated unexpectedly in the middle of a computation. The only way we could receive this notification is if the project client were invoking an XML-RPC in the science application. In this scenario, when the connection between the client and science application is broken, the client will receive an exception and be able to determine that it was caused by the termination of the science application. If the science application had to act only as a client to the project client, the project client would not receive any notification in the event of the termination of the science application. The other reason is that both the project client and the project server use the science application. The project server uses the science application to compute spot-check results. However, the project server and project client listen on different ports, so the science application would have to somehow determine which port was the correct one to connect to. Although the XML-RPC server in the science application introduces additional complexity, it was the only way to make the framework architecture work reliably. Information on the implementation of the science application can be found in Appendix C: Project Programming Guide.

5 Implementation

This chapter describes the processes we followed during the implementation of the framework, including the general methodology we followed as well as information about the implementation of specific parts of the framework.

5.1 Methodology

Most modules that comprise the framework were implemented and integrated in a bottom-up fashion. That is, all base classes were implemented and tested before classes that depended on them. The main reason we implemented the framework in this way was that we needed to determine how well Hibernate would work in our system and how difficult it would be to make our classes compatible with Hibernate. Only the base classes needed to be stored in the database, so we implemented and tested them with Hibernate first. If we found that Hibernate would not suit our needs, we could have immediately changed our architecture to use a different method. This type of change would have been much more difficult to make if we had implemented the base classes last.

During the implementation of each class we wrote unit tests in JUnit to test the important functionality of that class. Not all classes could be tested in this way, such as the main project server class, but we were able to write JUnit tests for many of the classes. The base classes and the transitioner were especially well suited for unit testing because they contained methods that could be tested without having the server or client running. By the time implementation was completed we had created over 30 unit tests for most of the base classes and the transitioner. We used these unit tests frequently during our regression testing (see Section 6.1 Regression Testing).

5.2 Persistent Classes

Persistent classes are classes whose objects can be stored in a database. The persistent classes were among the first that we implemented because they were the most important base classes, and we needed to verify that Hibernate could be used to easily store and retrieve them from various types of databases. The four classes in SLINC that needed to be persistent were: WorkUnit, Result, Project, and Volunteer. To make these classes compatible with Hibernate we needed to create XML configuration files that described the composition of each class. Once these configuration files were created, we could use Hibernate to store, retrieve, and query objects without needing to write any SQL or JDBC code.

5.3 Components

The server components were implemented after the base classes. The transitioner was implemented first because it only depended on the base classes, and it could be tested without a functioning project server. Following the transitioner the spot-check generator and default validator were implemented because they both interacted with the

transitioner. The project server was the last server component to be implemented because it depended on all of the other components, so it could not be easily tested before the other components were completed. The project client was one of the last components to be implemented, but it was implemented concurrently with the server. Having parts of the client and server working simultaneously facilitated the testing of the system, as all of the communication between the server and client was sent over XML-RPC. Testing the XML-RPC interface outside of the context of an executable was very difficult, so almost all testing involving XML-RPC was performed with the project server and client.

5.4 Public Resource Computing Project Example

Learning to use any framework can be difficult, so we recognized the need to create an example for others to use as a reference. Another reason we wanted to create an example public resource computing project was that we would be able to use it in our functional and performance testing. Our example project is a distributed application for finding prime numbers. We chose this application to be our example because it was simple to implement, and it demonstrated all of the features of SLINC.
The work unit generator for this project partitioned the set of positive integers into ranges which each contained 1,000,000 numbers. To create the work units we only needed to encode the first and last numbers in the range into a byte array. In the science application, we used a very simple algorithm to test whether a given number was prime. The algorithm divided the number by all numbers between two and the square root of the number being tested, inclusive. A consequence of this method was that different work units could require different numbers of operations to compute their results. The number of computations required grew proportionally to the square root of the last number to be checked in each successive work unit. The reason why it might have been a problem to have work units of different complexities was that our performance tests could have been skewed by the work units that took much longer to complete. Fortunately, the growth of the work unit complexity was small enough that our results were not significantly affected by it. More information can be found in Appendix E: Example Project.

6 Testing

This chapter describes the methods we used to test the framework during its implementation. We primarily used regression testing, functional testing to find faults during the development of the framework. After we completed the implementation of the framework, we conducted performance tests to determine whether there existed any serious performance problems. In addition to quantitative testing, we used two forms of qualitative testing to assess the utility of our framework. One method was a comparison of the steps required to create a public resource computing project using BOINC to the steps required to create the same project using SLINC. The other qualitative assessment was a peer review of our work by another researcher in the field of public resource computing.

6.1 Regression Testing

Regression testing was a critical part of our testing process. It allowed us to quickly find faults introduced by changes to the architecture or modifications at the class level. Through regular regression testing we were able to reduce the amount of time between the introduction of a fault and its detection and correction.

6.1.1 Methodology

We created a JUnit test suite for the unit tests we had created (see Section 5.1 Methodology), which allowed us to easily run all of the unit tests and to quickly see which tests had failed. After any significant change to the framework we ran the test suite to verify that the framework was still working correctly. Regression testing was very useful for finding subtle faults during the testing process. For example, after changing the abstract data type that was used to store queues of work units and results in the transitioner, we ran all of our regression tests. We found that four of the transitioner unit tests had failed, and we were able to use that information to quickly isolate the cause of the errors. Without regression testing we would not have found problems like these until the functional testing was performed, which would have made it much more difficult to find the cause of the errors.

We used regression testing to test all of the base classes as well as the transitioner. All of these classes had interfaces that were simple enough that they could be driven by a unit test. We wrote unit tests to save and retrieve the base classes from the database to test the integration with Hibernate. We also tested the transitioner to determine whether each type of state transition was performed correctly.

6.2 Functional Testing

Unit testing was effective in testing individual code modules and for limited integration testing, but it was also necessary to test the server and client components in configurations that would simulate their intended use. We performed many functional tests to verify that the major components of the framework were working correctly. We

found functional testing to be the most effective method for testing inter-process communication between the framework components, and it was also effective for testing functionality that could not easily be driven by a unit test. An example of a code module that could not easily be tested through unit testing was the project server. It was designed to perform many tasks automatically, such as initializing the database, configuring the log files, and starting necessary server components such as the transitioner. Since each of these tasks was designed to be completed automatically by the server process immediately after being started, and the server methods were not intended to be called directly by another process, it would have been difficult to write unit tests for those methods.

6.2.1 Methodology

After the implementation of each major feature was completed, we designed a functional test to verify that the feature was working as intended. We performed these tests regularly during the implementation process in order to detect any faults as soon as possible. Table 1 contains our most frequently performed functional tests. Before each test was performed, we first deleted all tables in the project database and then recreated all of the necessary tables so that each test would begin with an empty database.

Functional Test Name	Components Tested	Description of Test
1. Project Server Initialization	Project Server	**1.** Write a valid project configuration file, and save it in the **cfg** directory. **2.** Start the project server. **3.** Verify that the project server did not produce any error messages, and that it is listening on the correct port for XML-RPC requests.
2. Work Unit Generation	Work Unit Generator, Project Server	**1.** Start the project server. **2.** Start the example work unit generator, which is configured to generate 64 work units if the database does not contain any. These work units should each have a point value of 1 and a priority of 0. **3.** Shut down both components. **4.** Verify that there are 64 work units in the database, that the task IDs are correct, that every work unit is in the ingress state, and that the point value and priority of each work unit is correct.
3. Client Work Unit Request	Project Client, Project Server, Local Transitioner, Work Unit Generator	**1.** Start the project server. **2.** Start the example work unit generator. **3.** Start the project client.

		4. Verify that the client requested a new work unit, and that the transitioner assigned the first ingress work unit to the client.
4. Invocation of Science Application by Client	Project Client, Science Application, Project Server, Local Transitioner, Work Unit Generator	**1.** Perform all steps in functional test 3. **2.** After the client has requested a new work unit, verify that the client has started the science application, which should be listening for XML-RPC requests on a port defined by the project. The science application should also output log files after being started.
5. Computation of Work Unit by Science Application	Project Client, Science Application, Project Server, Local Transitioner, Work Unit Generator	**1.** Perform all steps in functional test 4. **2.** After the science application has started, verify that it is functioning by examining its log file in the **log** directory. It should contain a list of all the prime numbers the science application has found.
6. Work Unit Save/Restore by Client	Project client, Science Application, Project Server, Local Transitioner, Work Unit Generator	**1.** Perform all steps in functional test 5 **2.** After the client requests a new work unit, verify that the client has written the work unit data to disk in the **data** directory. **3.** Shut down the client and science application before the science application has finished computing the result for that work unit. **4.** Start the project client. **5.** Verify that the client does not request a new work unit from the project server, but instead reads the work unit stored in the **data** directory and sends that work unit to the science application for processing. This verification can be performed by examining the client and science application log files.
7. Use of Check-pointing by Science Application	Project client, Science Application, Project Server, Local Transitioner, Work	**1.** Perform all steps in functional test 5 **2.** Verify that the science application is sending check-points to the client

	Unit Generator	by examining the log files and checking for the presence of the **data/checkpoint.dat** file. **3.** After a check-point has been saved, restart the science application and the project client. **4.** Verify that the science application received its last checkpoint and resumed its computations from the point at which the last check-point was saved.
8. Spot-Check Generation	Project Server, Local Transitioner, Work Unit Generator, Spot-Check Generator	**1.** Configure the example project to use some minimum number of spot-checks, for example 3. **2.** Start the project server. **3.** Start the example work unit generator. **4.** After the spot-check generator has completed (see log files), verify that 3 spot-check work units and 3 spot-check results exist in the database.
9. Validation of Results and Selection of Canonical Results	Project Server, Local Transitioner, Work Unit Generator, Default Validator, Project Client, Science Application	**1.** Start the project server **2.** Start the example work unit generator. **3.** Start the project client. **4.** Verify that each result is validated by observing the state transition of each result from ingress to pending. **5.** As soon as the minimum number of results has been received for a work unit, verify that a canonical result is selected by observing one result transition from pending to egress and the removal of all other results for that work unit.
10. Spot-Check Failure Action	Project Server, Local Transitioner, Work Unit Generator, Default Validator, Project Client, Science Application	**1.** Modify the default transitioner to always mark spot-check results as invalid. **2.** Configure the example project to use spot-checks and distribute them with a probability of around 0.25. **3.** Also, set the minimum number of results to 1 so that it is likely that some work units will be retired before the spot-check work unit is distributed.

		4. Choose a spot-check action to test and configure the project to perform that action when a spot-check fails. **5.** Start the project server. **6.** Start the example work unit generator. **7.** Start the project client. **8.** Wait until the spot-check result is received and invalidated. This can be discovered by examining the server log files. **9.** Examine the database and log files to determine whether the appropriate action was taken after the saboteur was discovered. For example, if configured to invalidate all past results, all egress work units should have been transitioned back to the ingress state.
11. Remote Transitioner	Project client, Science Application, Project Server, Remote Transitioner, Work Unit Generator	**1.** Configure the example project to use a remote transitioner. **2.** Start the remote transitioner. **3.** Perform functional tests 1-10.
12. Database Support	Project client, Science Application, Project Server, Remote Transitioner, Work Unit Generator	**1.** Configure the example project to use the HSQLDB database. **2.** Perform functional tests 1-11. **3.** Repeat using the MySQL, PostgreSQL, Oracle, and Microsoft SQL Server databases.

Table 1: Functional Tests

6.3 Performance Testing

SLINC must be scalable if it is to be a useful tool for public resource computing. Most of the functional tests were conducted with only one or two clients. We decided to test the scalability of the system by comparing the amount of time it took to complete a set number of work units with different numbers of clients. We also used the performance tests to determine whether the choice of programming language had any effect on the overall performance of the project server. One of the most significant advantages of SLINC is that project-specific components can be developed in many different languages, so we wanted to identify any significant performance discrepancies between languages. We chose to compare C++ and Java in our performance test. The results of our performance tests appear in Section 7.1 Performance Analysis.

6.3.1 Methodology

We used the example project (see Appendix E: Example Project) to test the performance of the framework. The example work unit generator partitions the set of positive integers into work units containing 1,000,000 consecutive integers. The example science application searches that range of integers for prime numbers using a simple algorithm described in Appendix E: Example Project. We decided to conduct the performance test using both the Java and C++ versions of the example science application. Our reason for doing so was that we wanted to determine whether there were any major performance discrepancies between the two implementations of the application. Since the science application was so simple we did not expect the choice of programming language to have a significant impact on its performance. Therefore, any appreciable difference in performance would probably indicate that one or more components of SLINC behaved differently depending on the programming language used to implement the project-specific components. Programming language independence was critical to of our goal of creating a flexible framework, so it was important to verify that different languages would work equally well when coupled with SLINC.

Our tests required only the most basic functionality from the framework, so the project configuration was simple. For these tests we used the WPI MySQL server (mysql.wpi.edu) as our project database. We configured the project to use the local transitioner and the default validator, and spot-checks were not used. The project was configured to accept one result for each work unit.

We began each test by purging the project database, starting the project server, and executing the example work unit generator. The example work unit generator would then create sixty-four work units and send them to the project server. With 64 work units and 1,000,000 integers per work unit, the total quantity of numbers tested for primality was 64,000,000. After the initial 64 work units were created, we shut down the work unit generator so that it would not generate more work units. Once the work units had been generated, we started one or more clients. We measured the time to complete all work units by examining the server log files. The time to completion was computed by comparing the time the first work unit was requested to the time the last result was returned.

We used several different configurations for the performance test. We varied the number of clients in the system using 1, 2, 4, 8, 16, and 32 computers with one client per computer. We tested each configuration twice, once using the Java science application and once using the C++ science application. The C++ application was compiled with gcc, and we did not optimize the binary in any way. Only the following compile flags were used: *-Wall –g*. The *–Wall* flag enables all warnings during the compilation, and the *–g* flag adds extra debugging symbols to the binary, which may slightly degrade the performance of the application.

Each test was conducted three times, and the average time for each test was used in our analysis. It is important to note that this performance test was incomplete. Public resource computing projects may have hundreds or thousands of users, but our tests used a maximum of only 32 clients. The purpose of this test was not to determine how the framework would react to extreme load, but rather to verify that the framework would behave as expected when multiple clients were used and to determine whether the amount of work completed would scale efficiently as the number of clients increased.

6.3.2 Test environment

To carry out the performance test we built a cluster in the distributed computing lab in the Computer Science Annex. Although we were given over 70 computers to use in the cluster, we had a limited number of switches, cables, and surge suppressors, so we were only able to use 32 of them for our tests. Of those 32, 23 had 350MHz Pentium II processors, and the remaining 9 had 400MHz Celeron processors. All of the machines had 128MB of RAM. The head node was a dual processor Athlon MP 1.2GHz machine with 1GB of RAM. We chose to install an OSCAR[33] cluster due to its ease of installation and compatibility with our older hardware. OSCAR supports many distributed computing architectures and has tools to simplify the use and administration of the cluster. One tool we used in our tests was *cexec*. This tool allows a user to run the same command on all nodes in the cluster simultaneously. We used *cexec* to start all of the project clients simultaneously after the work units had been generated.

Due to the heterogeneous composition of our cluster in terms of processor architecture, we tried to minimize the effects of any performance discrepancy during our testing. The tests using 1, 2, 4, 8, and 16 nodes were all executed on the Pentium II machines. Only the last test, using all 32 nodes, included a mixture of both Pentium II 350MHz and Celeron 400MHz machines.

6.4 Usability Testing

The purpose of the usability testing was to determine whether we had succeeded in creating a public resource computing framework that was easy to use. We performed this testing by comparing SLINC to BOINC and by asking other public resource computing researchers to evaluate our framework.

6.4.1 Comparison with BOINC

Using documentation available on the BOINC website, we made a qualitative comparison of SLINC to BOINC. We compared the two frameworks using three different measures: the number of steps involved in creating a project, system and project requirements, and major features. The purpose of the comparison was to determine whether the process of making a simple public resource computing project was easier using SLINC or using the BOINC. Using this comparison we analyzed the distinguishing features of our framework to determine whether they contributed to our primary objective of creating a public resource computing framework that was easy to use. The comparison with BOINC can be found in Section 7.2.1 Comparison with BOINC.

6.4.2 Peer Review by a PRC Researcher

In addition to the performance testing and qualitative comparison to BOINC, we were interested in the opinions of other researchers in the field of public resource computing. We spoke with David Toth, a graduate student at Worcester Polytechnic Institute who is currently writing a Ph.D. dissertation on the topic of public resource

computing. He agreed to help us test the usability of SLINC by using it to create a simple public resource computing project himself. Using Appendix B: Project Creation Guide, Appendix C: Project Programming Guide, and the latest release of SLINC, David modified our examples to create a new public resource computing project. We asked for his opinion about the current state of SLINC and for suggestions for ways in which we might improve it. We used David's responses in our analysis, which can be found in Section 7.2.2 Analysis of Peer Review.

7 Analysis of Results

We performed several different types of tests on SLINC to determine whether we accomplished our goals. The performance test evaluated the framework's ability to support multiple clients that were simultaneously contributing to a public resource computing project. The comparison with BOINC assisted in evaluating the usability of SLINC. By interviewing a researcher in the field of public resource computing who had tested our framework we were able to obtain important insights about which aspects of SLINC were easy to use and which could be improved.

7.1 Performance Analysis

We used varying numbers of clients to determine whether there were any serious performance problems with the framework when multiple clients were active simultaneously. We also used the performance test to try to identify any possible performance issues with science applications written in different programming languages.

7.1.1 Results

Table 2 shows the results from the three trials of each of the Java science application performance tests, as well as the average time required to complete each test. The average test time for 1 computer was approximately 63 minutes, and the average test time for 32 computers was approximately 2 minutes. Figure 4 is a graph of the three trials for each Java test. We expected an exponential decrease in the time required to complete the test as the number of computers doubled. It is clear that we achieved this exponential decrease in our tests.

Number of Computers	Trial 1 Time (ms)	Trial 2 Time (ms)	Trial 3 Time (ms)	Average Time (ms)
1	3785748	3785712	3783552	3785004
2	1905617	1905044	1903378	1904680
4	966749	966450	966740	966646
8	495291	495671	494140	495034
16	260058	260361	260395	260271
32	141056	141471	141337	141288

Table 2: Time to Complete 64 Work Units with the Java Science Application

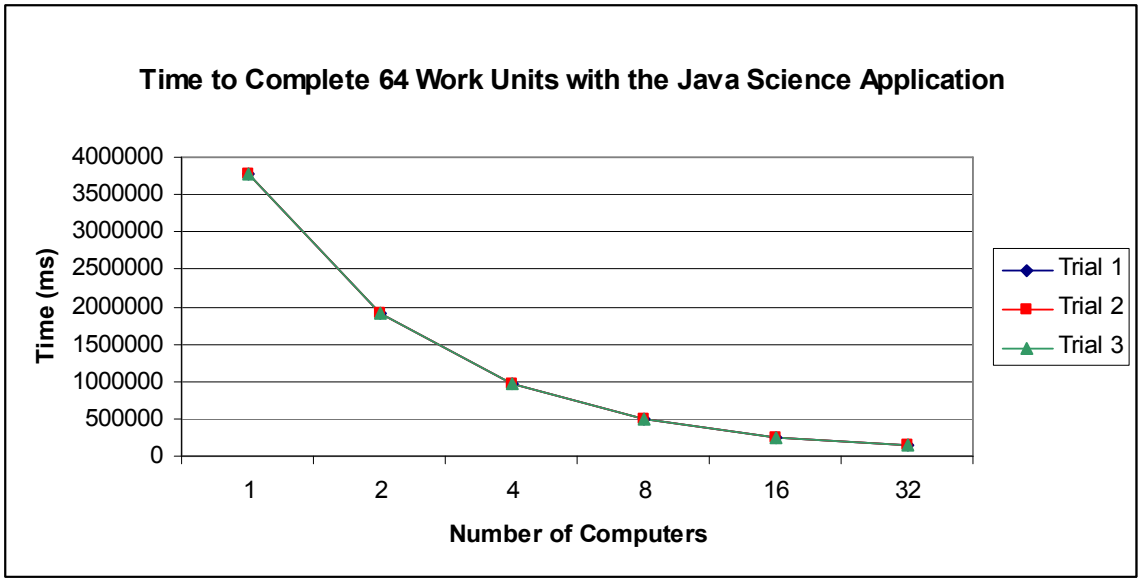

Figure 4: Time to Complete 64 Work Units with the Java Science Application

Table 3 shows the results for our performance tests using the C++ science application. For the C++ application, the average test time for 1 computer was approximately 53 minutes, and the average time for 32 computers was approximately 2 minutes. Figure 5 is a graph of the C++ science application results. Again, it clearly shows an exponential decrease in the amount of time required to complete the tests.

Number of Computers	Trial 1 Time (ms)	Trial 2 Time (ms)	Trial 3 Time (ms)	Average Time (ms)
1	3162388	3156844	3156226	3158486
2	1904458	1905281	1904039	1904593
4	965900	966862	965063	965942
8	494775	493883	494053	494237
16	260280	259806	259948	260011
32	141880	141806	140944	141543

Table 3: Time to Complete 64 Work Units with the C++ Science Application

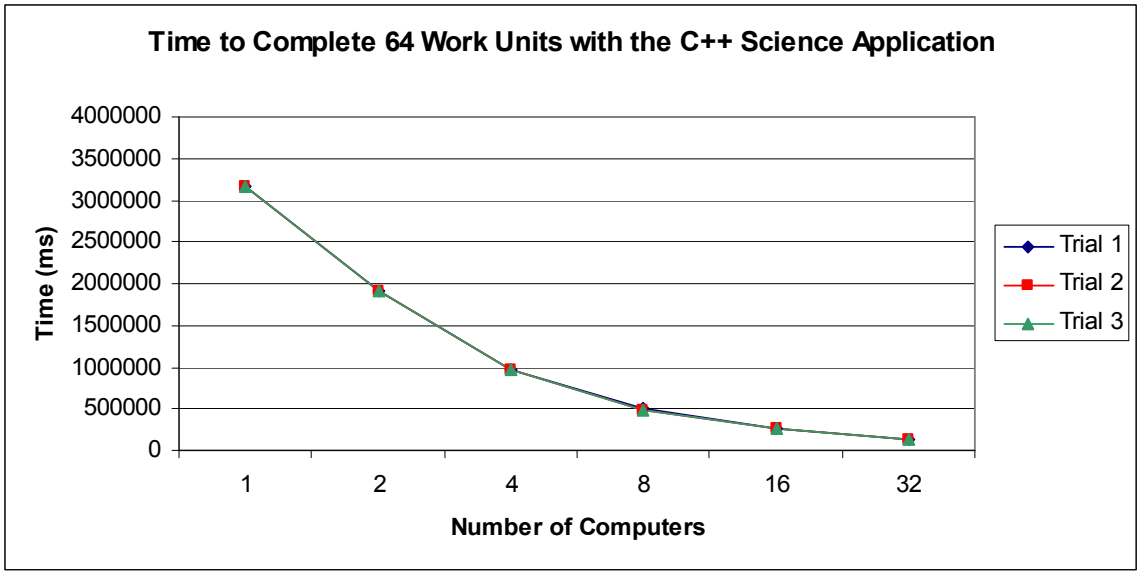

Figure 5: Time to Complete 64 Work Units with the C++ Science Application

The following two graphs show a comparison between the average performance of the C++ science application and the Java science application. Figure 6 uses a linear scale on the time axis, and Figure 7 uses a logarithmic scale. Both graphs show that the performance of the two science applications is roughly equivalent.

It is interesting that the C++ science application had a noticeably lower time than the Java science application when run on only one computer, but this performance advantage disappeared when more than one computer was used. It is not apparent what would cause this performance discrepancy in only one of the tests, but it could be caused by a difference in the way that C++ and Java threads are scheduled. The minimum-priority compute thread that is created with the pthreads library in the C++ science application is scheduled directly by the Linux kernel. However, the corresponding thread in the Java science application is not scheduled by the Linux kernel, but rather by the Java Virtual Machine, which has its own thread scheduler[34]. It is possible that Java's thread scheduler might have allocated less processor time to the thread than the Linux kernel would have. The use of different schedulers may explain the performance discrepancy between C++ and Java, but it does not necessarily explain why the two science applications only performed differently when a single computer was used. One property of the single computer test that is much different from the other tests is its run time. The single computer test runs for about an hour, which is significantly longer than the other tests. It is possible that the two schedulers treat this long-running process differently.

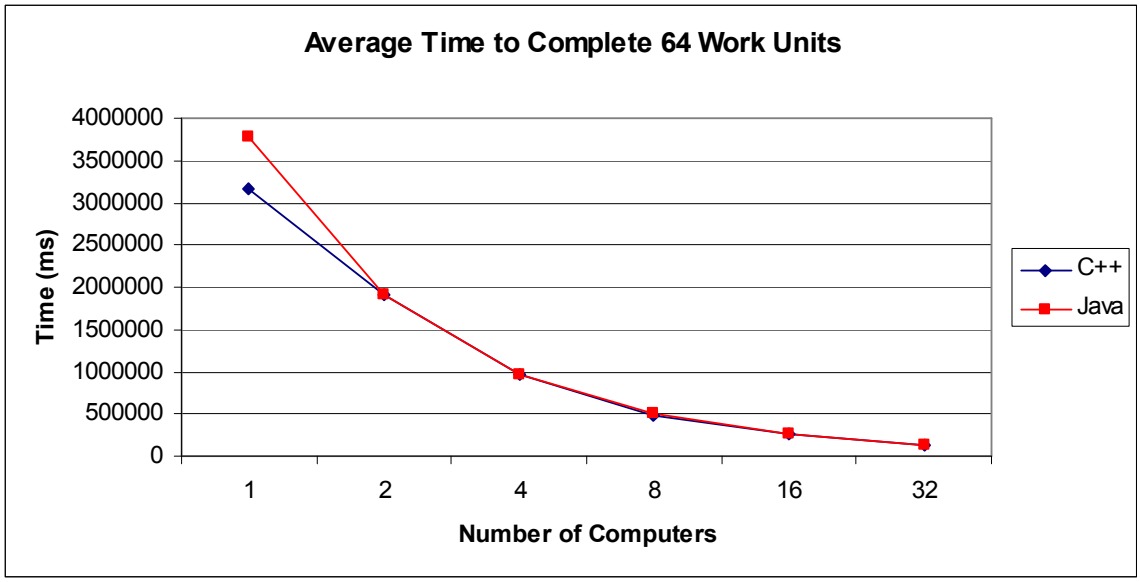

Figure 6: Average Time to Complete 64 Work Units, Linear Time Scale

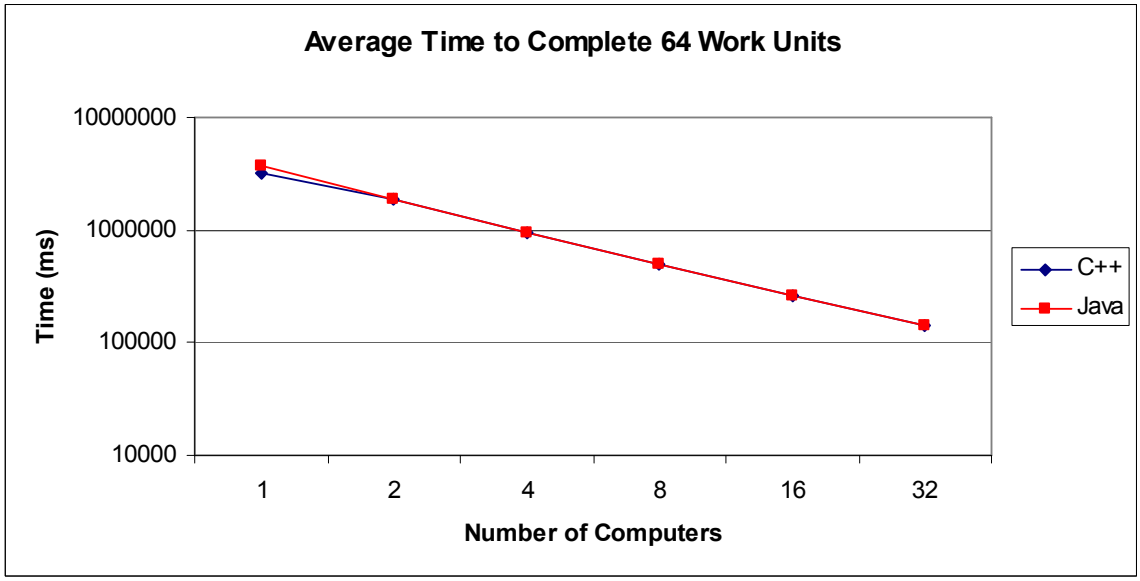

Figure 7: Average Time to Complete 64 Work Units, Logarithmic Time Scale

7.1.2 Analysis

Our goals in the performance testing were to verify that the framework could support multiple clients simultaneously, as well as to identify any serious performance discrepancies between projects developed in C++ and in Java. Our results indicated that both our C++ and Java example projects scaled well up to 32 clients. Unfortunately, we were not able to test SLINC with more than 32 clients, but the results we obtained were encouraging. The results also showed that the C++ and Java versions of the example project performed within one percent of each other in most cases, so there did not appear

to be any major performance discrepancies between the two languages. The results of the performance test suggest that we were successful in our goal of creating a scalable framework, although additional testing with greater numbers of clients would be necessary to verify this claim with greater certainty.

7.2 Usability Analysis

One of the problems we identified with the BOINC framework was that the process of creating a public resource computing project with BOINC was difficult, especially for those who were not familiar with Linux, MySQL, Apache, and C++. Furthermore, the process of installing BOINC and developing even a simple project involved many steps and required a high level of proficiency in Linux administration and C++. We used two methods to qualitatively measure the usability of SLINC: comparing SLINC to BOINC and having a colleague conduct a peer review of SLINC.

7.2.1 Comparison with BOINC

We compared SLINC to BOINC in three ways. We compared the process of creating the simplest possible project with SLINC to the process required to do the same with the BOINC framework. We also compared the requirements of our framework to the requirements of the BOINC framework. Lastly, we compared the major features of the two frameworks.

Table 5 shows a comparison of the steps necessary to create the simplest possible project using each framework. Due to its dependency on other software packages, such as MySQL and Apache, there are several steps involved in the installation of BOINC. Installing these other software packages requires proficiency in Linux system administration. Specifically, those installing BOINC must be familiar with installing, configuring, and securing those software packages in a Linux environment. By contrast, SLINC has no dependencies on other software packages, and there is no installation necessary to use SLINC. Everything required to make simple projects with SLINC is included in each release of the framework, available from our WPI SourceForge website (https://sourceforge.wpi.edu/sf/projects/jdb_prc_thesis).

To create a functional project, BOINC requires that an *assimilator* be developed. The assimilator component receives canonical result data from BOINC and sends it to a database that is separate from the project database. The advantage of having an assimilator is that it provides a way to separate the project and work unit data from the result data that needs to be analyzed. The reason that BOINC requires this component is that there is no automated way to extract all the results from the project database and to store them in a more accessible way, as files for example. The reason SLINC does not currently use an assimilator is that the framework does provide a way to extract the canonical results from the project database: the *extract_results* script. We believe that the functionality provided by the assimilator is useful, especially for projects that need to store their results in a unique way in order to make analysis easier. However, we also believe that the assimilator should be an optional component in order to keep the project creation process as simple as possible for projects that do not need the advanced

functionality it provides. The optional assimilator component would be an appropriate future enhancement to SLINC, as described in Section 8.4 Future Work.

Aside from the assimilator component, it is necessary to develop the same components with SLINC as with BOINC in order to create a simple project. These components include the science application and the work unit generator. The method each framework uses to communicate with the project-specific components is very different. BOINC uses an API written in C++ with FORTRAN wrappers, so components developed for the BOINC framework must be written in C++ or FORTRAN. Our interface is based on XML-RPC, and can therefore be implemented in any language for which there is an XML-RPC library. Developers are more likely to be familiar with XML-RPC because it is an open, general purpose web standard. The BOINC API, although it is open, is not a general API; it can only be used with BOINC projects.

Both SLINC and BOINC provide a default validator that simply marks all results as valid. The difference between the two frameworks in regard to the validator is that SLINC provides an integrated default validator that is part of the project server, whereas with BOINC it is necessary to compile the default validator and to configure the project server to execute that validator process. The difference in the way the default validator was implemented in the two frameworks is not a major advantage or disadvantage for either, but it serves as an example of one way in which we tried to minimize the number of steps necessary to create a project.

Perhaps the most significant difference between SLINC and BOINC is the project configuration process. With SLINC, projects can be configured using the Project Builder tool (see Appendix B: Project Creation Guide). The Project Builder is a graphical tool that is modeled after the ubiquitous installation wizards that most people are accustomed to using for installing and configuring software. Each step in the project configuration has its own screen in the Project Builder, and each screen has a text box near the bottom explaining the options that can be configured on that screen. After a user has completed the Project Builder, no further configuration is necessary. The Project Builder generates the project configuration files automatically. BOINC has many configuration steps, indicated by rows 11-17, 19, 21-22, 24, and 26 in Table 5. Each of these steps requires editing an XML file or executing a script.

Table 5 and Table 5 show that there are more steps involved in creating a project with BOINC than with SLINC. In addition, the steps in the BOINC project creation process tend to be more difficult to complete. There are several reasons why the BOINC process can be more challenging. One reason is that several of the steps require specific skills like experience with Linux system administration. Another reason is that BOINC does not have a single, unified configuration utility. Instead, users are required to edit XML configuration files in several different locations to create a functional project. Developing components for the BOINC framework requires not only an understanding of the C++ or FORTRAN programming languages, but also of the BOINC-specific API. All of these factors contribute to the complexity of the BOINC framework. We have addressed these problems in SLINC by making it platform-independent, programming language-independent, and by providing a simple tool for configuring projects.

Step	Description of Step for the SLINC Framework
1	Download the latest release of the framework

2	Configure the project using the Project Builder program. The project should use the HSQLDB database and the default validator.
3	Develop the science application in any language for which there is an XML-RPC library
4	Develop the work unit generator in any language for which there is an XML-RPC library
5	Copy the project-specific components to the appropriate *client* or *server* directories in the main framework directory
6	Run the *make_project_files* script
7	Extract the server distribution file, and run the *start_server* script to start the project server
8	Start the work unit generator

Table 4: SLINC Project Creation Process

Step	Description of Step for the BOINC Framework
1	Install Python
2	Install PHP
3	Install and configure Apache with the PHP module
4	Install and configure MySQL
5	Download the BOINC source code
6	Compile and install BOINC
7	Develop the science application component in C++ or FORTRAN using the BOINC API
8	Develop the work unit generator component in C++ using the BOINC API
9	Compile the provided "sample trivial validator" component
10	Develop the assimilator component in C++ using the BOINC API
11	Run the *make_project* script
12	Append the BOINC Apache configuration to the system Apache configuration
13	Configure the project by editing the XML configuration file
14	Add the project to the database using the *xadd* program
15	Move the MySQL socket to /var/lib/mysql/mysql.sock
16	Rename the science application using the correct naming scheme for the intended platform
17	Create an application version for the science application
18	Sign the science application with the *sign_executable* program
19	Run the *update_versions* script
20	Add work units to the project work folder at $PROJECTROOT/download
21	Create XML configuration files for each work unit
22	Create XML configuration files for each result
23	Run the *create_work* program to add the work units to the project database
24	Add entries for the work unit generator, feeder, transitioner, file

	deleted, trivial validator, and assimilator to the project configuration file
25	Start the project server
26	Edit the project configuration file to allow users to create accounts

Table 5: BOINC Project Creation Process

Table 6 shows a comparison of the requirements and restrictions of each framework. This table clearly presents the most significant advantages of SLINC. Since our entire framework is Java-based, both the server and client can be used on any platform supported by Java. Platform-independence is an important property of SLINC because it allows users to choose the platform they are most comfortable using. The BOINC server is restricted to the Linux platform. Another limitation of BOINC is the choice of languages in which the various server-side and client-side components can be developed. BOINC requires project developers to use C++ or FORTRAN. SLINC has a degree of language-independence due to its use of XML-RPC. Components for SLINC can be developed in any language for which there is an XML-RPC library. This language independence gives users of SLINC the ability to use the programming language that they are most familiar with. SLINC's use of Hibernate allows flexibility in the type of database a project can use. BOINC requires that every project use a MySQL database, but SLINC provides the ability to use the embedded HSQLDB database, MySQL, PostgreSQL, Oracle, or Microsoft SQL Server. The option to use the HSQLDB database is convenient for small projects because it allows projects to be created quickly without the need to install and configure an external database. Projects that need better performance have the option to use one of several types of databases. The databases that are supported by SLINC can easily be extended to include any database that is compatible with Hibernate. In addition to being flexible, SLINC is simple to use because it does not have any external software dependencies other than the Java Virtual Machine. Everything required to use the framework is provided.

Requirement	SLINC	BOINC
Platform	Server and client tested on Linux and Windows, but should function on any platform supported by Java	Server limited to Linux platform; client supported on Windows, Linux, and Mac OS X
Programming Language for Project-Specific Components	Any for which there exists an XML-RPC library	C++ and optionally FORTRAN
Framework Interface	XML-RPC interfaces, documented in Appendix D: XML-RPC Interface Specification	C++ APIs with FORTRAN wrappers
Database	HSQLDB, MySQL, PostgreSQL, Oracle, or Microsoft SQL Server, but can be extended to support	MySQL

	any database that is compatible with Hibernate	
Other Software	JVM version 1.5 or later	Apache, Python, PHP

Table 6: Comparison of Framework Requirements

Table 7 shows a comparison of the major features of each framework. Several of the features, representing the core functionality of a public resource computing framework, are present in both SLINC and in BOINC. Some of these features include result validation, saboteur detection, clients that are supported on multiple platforms, and the ability to distribute server components in order to accommodate heavy server loads. The other features of the two frameworks are quite different, reflecting possible differences in design goals. Our goals were to design a public resource computing framework that was flexible and easy to use so that researchers could rapidly develop and deploy projects. The features of the BOINC framework suggest that the goals of its designers were to create a single framework to support many projects simultaneously. Security also seems to have been a major concern for the BOINC developers; each project's science application must be signed with that project's private key, providing a way to verify the authenticity of a science application by using the project's public key. One of the major advantages of SLINC over BOINC is the complete example project and templates provided with the project. Future project developers will be able to learn about how to create projects using SLINC by examining an example of a simple, but completely functional, project. The component templates are intended to simplify and shorten the development process for projects written in the C++ and Java languages. The templates contain all code necessary to build a project with the exception of certain methods which are specific to each project. These methods include the science algorithm, the method that generates work units, and the method that validates results. The code that controls the logic of each component, as well as the implementation of the XML-RPC interfaces, is provided to minimize the amount of code that project developers need to write. The use of the templates is optional, so the project developers wanted to implement all of the code for each component themselves, they would be free to do so using the documentation in Appendix C: Project Programming Guide and Appendix D: XML-RPC Interface Specification.

Feature	SLINC	BOINC
Component-based architecture	Yes	Yes
Result validation	Yes	Yes
Voting	Yes	Yes
Spot-checking	Yes	Yes
Cross-platform server	Yes	No
Cross-platform client	Yes	Yes
Server components can be distributed for scalability	Yes	Yes
Single utility for configuring projects	Yes	No

Support for hosting multiple projects with the same project server process	No	Yes
Support for executing multiple science applications from the same project client process	No	Yes
Support for exporting results to another database	No	Yes
Support for extracting results from the database and writing them to disk	Yes	No
Support for signed science applications	No	Yes
Included example components	Science Application, Work Unit Generator, and Result Validator implemented in C++ and Java.	Science Application implemented in C.
Included template components	Science Application, Work Unit Generator, and Result Validator implemented in C++ and Java.	None

Table 7: Comparison of Framework Features

7.2.2 Analysis of Peer Review

We interviewed David Toth, a graduate student at WPI who is currently writing his Ph.D. dissertation in public resource computing, to gain insights about the usability of SLINC and how it might be improved. We provided David with the latest release of SLINC as well as our framework documentation, which consisted of Appendix B: Project Creation Guide, Appendix C: Project Programming Guide, and Appendix D: XML-RPC Interface Specification. David's goal was to use SLINC to create a simple public resource computing project.

Before creating his own project, he went through the process of creating a project using the example components we wrote (see Appendix E: Example Project). David provided us with several suggestions for improving the usability of the Project Builder tool and accompanying documentation. Originally, we did not have any help system in the Project Builder itself, only the separate documentation. He suggested that we develop a simple help system in the Project Builder so that users would be able to view basic information about each step in the Project Builder without the need to search through the separate documentation. Our response to this suggestion was to include a help section at the bottom of each screen in the Project Builder. These help sections were simply text boxes that were not editable, so we were able to add them to the Project Builder without a great expenditure of time and effort. The help sections were meant to provide quick answers to common questions, and were thus extremely detailed. If users needed more

detailed information, they could refer to the complete documentation. Upon reviewing our modifications, David commented that the help sections made the Project Builder easier to understand and use. He also identified certain Sections of the documentation that seemed ambiguous or confusing and suggested ways in which we could improve them. We updated our documentation based on these comments.

David's next step was to create his own simple public resource computing project based on SLINC. An important usability issue that he identified immediately was the requirement that project developers learn XML-RPC in order to write the project components that would interact with SLINC. His observation was that the XML-RPC code for a given component would never need to change; only the logic specific to each project would need to change. David's suggestion was to separate the implementation of the XML-RPC code from the project-specific logic by creating templates for each component. Each project could simply add its own logic to certain classes in these templates without the need to implement the XML-RPC code. However, each programming language would need its own set of component templates. Although we agreed that component templates would make the project creation process easier, we realized that creating templates for many programming languages would require a significant time investment. The compromise we reached was to develop templates for C++ and Java. Our choice to create templates for the C++ and Java programming languages was partially based on their established use in public resource computing. C++ is used in BOINC projects, and Java was used in Sarmenta's Bayanihan[4] system. Developing a project using any other language would require writing the necessary XML-RPC code using information from Appendix C: Project Programming Guide and Appendix D: XML-RPC Interface Specification.

8 Conclusions

Our primary goal in this thesis was to design and implement a public resource computing framework that addressed the usability issues we identified with similar frameworks like BOINC. Our focus was on ease of use, but it was also important to create a framework whose core functionality was equivalent to existing frameworks and whose performance could scale efficiently as volunteers joined the project.

The architecture of the framework is modular and scalable. It is possible to execute most of the server components on separate computers, reducing the load on each individual computer. Our design decisions centered on making the framework as flexible, extensible, and easy to use as possible. Our choice to develop the framework in Java had several important implications. It allowed SLINC to be inherently cross-platform, which made it both flexible and easier to use because future project developers would be able to use whichever operating system they were most familiar with. Since most of our development was in Java, we were able to use the Eclipse IDE and the Apache Ant build system, which simplified our development process. Using Java also allowed us to take advantage of other Java libraries such as Hibernate and HSQLDB. By using the Hibernate library we were able to support many different types of databases without much effort. Another advantage of Hibernate was that it allowed SLINC to support the HSQLDB embedded database. The HSQLDB database is convenient for users who do not have experience installing and configuring a traditional RDBMS. If HSQLDB is used, it will run as part of the main server process, so there is no need to install, configure, and start an external database. Our decision to use XML-RPC for communication between the framework components was important because it provided the flexibility for project developers to write their project components in potentially any language.

In addition to flexibility, usability was an important concern for us. We identified the properties of other frameworks that made them difficult to use and tried to find ways to make public resource computing more accessible to the average researcher. Configuring BOINC was difficult because there are several different configuration files that need to be edited by hand. To address this problem we developed a single, unified configuration utility with a graphical wizard-style interface. We also found that developing projects with BOINC was difficult because it required strong C++ skills and there was very little example project source code available. Our solution was to provide a complete example of a working project, including all components that could be implemented by project developers, as well as component templates. These template components contain all necessary code except the algorithms that are specific to each project. Project developers can use these templates to rapidly develop complete public resource computing projects by filling in the appropriate methods with project-specific code to generate work units, process science data, and optionally to validate results. We have implemented the examples and templates in Java and C++, but since these project-specific components communicate with the framework via XML-RPC, they can be written in potentially any programming language.

8.1 Functionality

SLINC provides all of the core functionality necessary to create and maintain a public resource computing project, as well as some advanced features and features that are not present in other frameworks. At a high level the core features we implemented were the ability to generate work units from some data source, distribute the work units to clients running on volunteers' computers, compute the result for each work unit using a domain-specific algorithm, and to return the results to the project server. These high-level core features are comprised of several other core features, such as the ability to manage the lifecycle of a work unit to guarantee that a result will be accepted for that work unit in a timely manner.

In addition to the core features necessary for a public resource computing framework to be useful, we also implemented several advanced features and features that are unique to SLINC. The advanced features available in our framework are the three types of result validation that we implemented, including validation of individual results and Sarmenta's voting and spot-checking techniques. There are several unique features in SLINC. The framework has cross-platform compatibility and the project-specific components are programming language-independent. There is a graphical utility to simplify the project configuration process. We also provide a tool to extract the results from the database and write them to a file system so they can be accessed more easily.

8.2 Performance

The client computers that comprise public resource computing projects are typically personal computers with modest performance capabilities. Due to the relatively low performance of personal computers, public resource computing projects often need hundreds or even thousands of volunteered computers to process their science data in a reasonable amount of time. Any public resource computing framework needs to be able to support a large number of clients, a significant percentage of which might be attempting to access the project server simultaneously at any given time. Thus, the performance of the framework must scale efficiently as new volunteers join a project and begin contributing their resources. Although we did not have the resources to test the performance of SLINC with hundreds or thousands of clients, we were able to analyze the framework's performance with 1, 2, 4, 8, 16, and 32 clients contributing to the project simultaneously. We found that the performance of the framework scaled efficiently to 32 clients, and that there was no significant performance difference between the C++ and Java implementations of the example project.

8.3 Usability

Usability was the major problem we found with other public resource computing frameworks like BOINC. During the design and implementation of SLINC we endeavored to make it as easy to use and as flexible as possible. The cross-platform nature of the framework makes it flexible, but also easier to use because there is no need to learn a new operating system in order to use the framework. The Project Builder tool facilitates the project configuration process by providing a simple interface with help information for choosing project configuration options. We have developed a complete

and thoroughly documented example project in both C++ and Java to demonstrate how the project components should be implemented. To minimize development time, we have also created template components in C++ and Java that can be filled in with project-specific code to create a complete project. We also created step-by-step documentation explaining the process of developing a public resource computing project using SLINC.

8.4 Future Work

Although we have succeeded in creating a fully functional public resource computing framework, there are several features that could be added or further developed to provide additional functionality.

8.4.1 Additional Scalability Testing

Public resource computing projects can have hundreds or thousands of active volunteers, but we were only able to test SLINC with 32 clients. We would like to be able to analyze the performance of the framework with a large number of clients. This testing could be done by using a larger cluster or by developing a program to simulate a pool of clients. If we were to find any scalability problems in this performance testing, we could then use a code profiler tool to determine where the bottlenecks were and attempt to remove them.

8.4.2 Assimilator Component

The BOINC framework has a server component that ours does not: the assimilator. The function of the assimilator is described in detail in 7.2.1 Comparison with BOINC, but its basic purpose is to move the retired work units and valid results out of the project database and into a different database. This functionality is useful because it keeps the project database small so that access times are minimized. It also maintains a cleaner separation of the work units and results that are still in progress and those that are complete. The disadvantage of the assimilator is that it is a project-specific component, so using an assimilator would require more work on the part of project developers, which would not support our usability goal. However, the assimilator could be implemented in a similar way as the validator. The framework could provide a default assimilator, which would just keep all work units and results in the project database, but the project developers could choose to implement a custom assimilator that would transfer the completed work units and results to a different database.

8.4.3 Security Enhancements

Security is implemented in a very simple way in SLINC because security was not one of our primary goals. Certain XML-RPCs, like those used to shut down the project server or retrieve all valid result data, are password-protected. However, the password is sent in clear text; it is not encrypted in any way. A better approach would be to have the project server start two XML-RPC servers, one that is for normal client access and one that is for secure administrator access. The Apache XML-RPC library supports SSL-

encrypted communications, so passwords could be transmitted securely. It also might be necessary for some projects to encrypt the work unit data and results that are transferred between the project server and the clients, so it could be useful to provide the option to encrypt the public XML-RPC server as well.

8.4.4 Additional Spot-Check Configuration Options

David Toth suggested that we enhance the way the framework performs spot-checking to include a feature that is present in other frameworks. This feature allows projects to specify that every client should have to pass a spot-check before it has computed a specified number of work units. For example, a project might want to specify that every client be sent a spot-check work unit within the first 10 work units it receives. This feature would reduce the impact of a saboteur because that saboteur would be detected more quickly. If the project were configured to invalidate all of the saboteur's past results, there would be a smaller number of results that would need to be recomputed. This feature would require a way to uniquely identify every client rather than every volunteer because there might be multiple clients associated with a single volunteer, and each of those clients would be required to pass the spot-check. Clients could be uniquely identified by having them generate some hash or a public-private key pair the first time they are started; they could then send their unique identifier to the project server.

8.4.5 Support for Database Migration

Some project developers may decide to use the HSQLDB database for their project because it is the simplest, but as their project grows the performance of HSQLDB may degrade significantly. It could be a useful feature to allow a project to migrate its project database to a different database. For example, the project database could be migrated from HSQLDB to MySQL for better performance. This operation would require exporting the database schema to the new database, then transferring all information from the old database to the new database. The database migration could be performed through the Project Builder tool. Having the ability to change databases would increase the flexibility of the framework because projects could start with a simple configuration and move to a higher performance configuration at a later time if necessary.

Appendix A: Design Diagrams

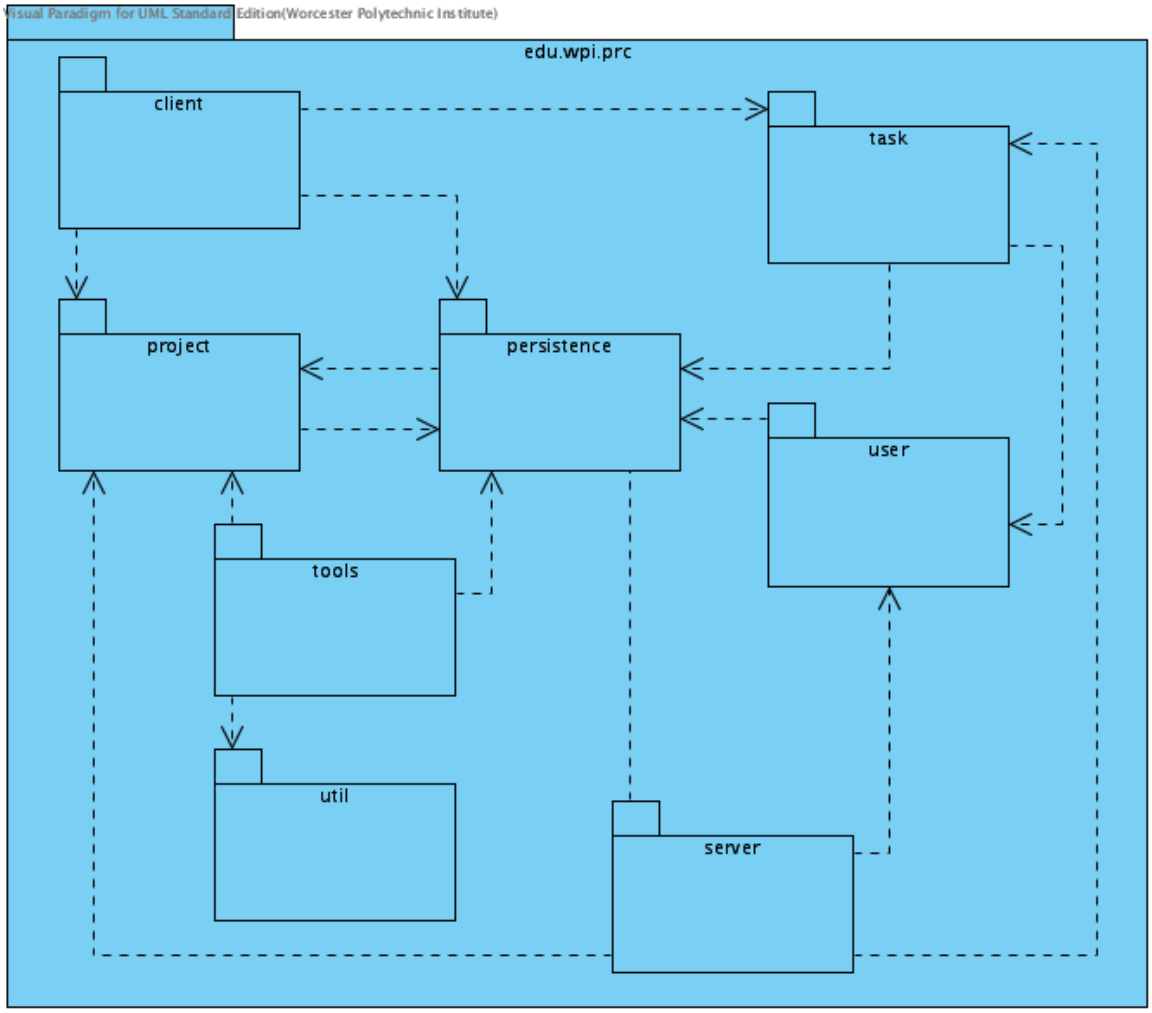

Figure 8: Package Dependency Diagram

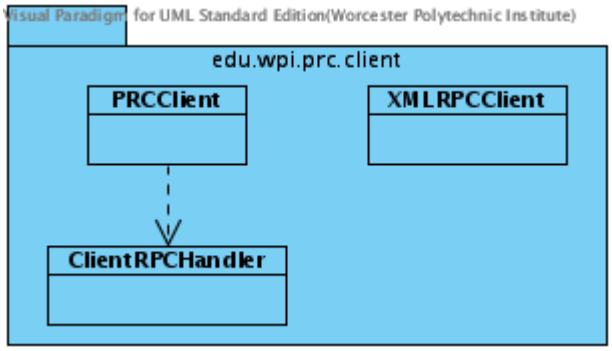

Figure 9: Client Package Diagram

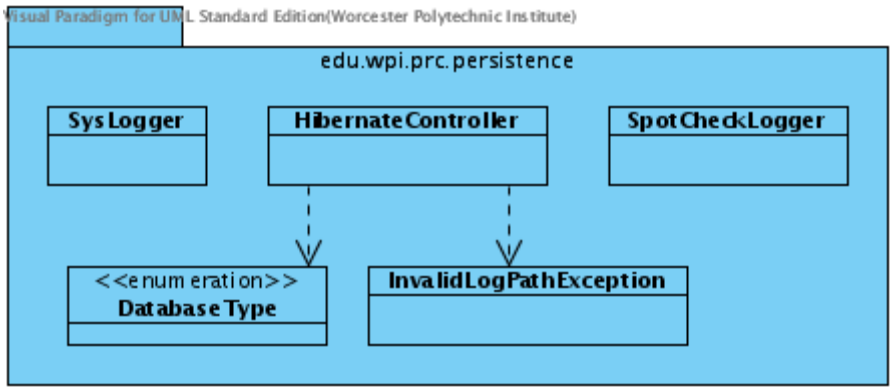

Figure 10: Persistence Package Diagram

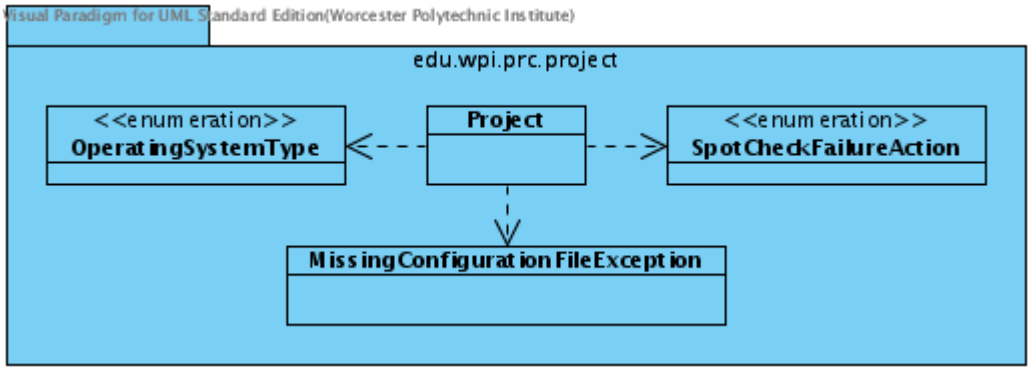

Figure 11: Project Package Diagram

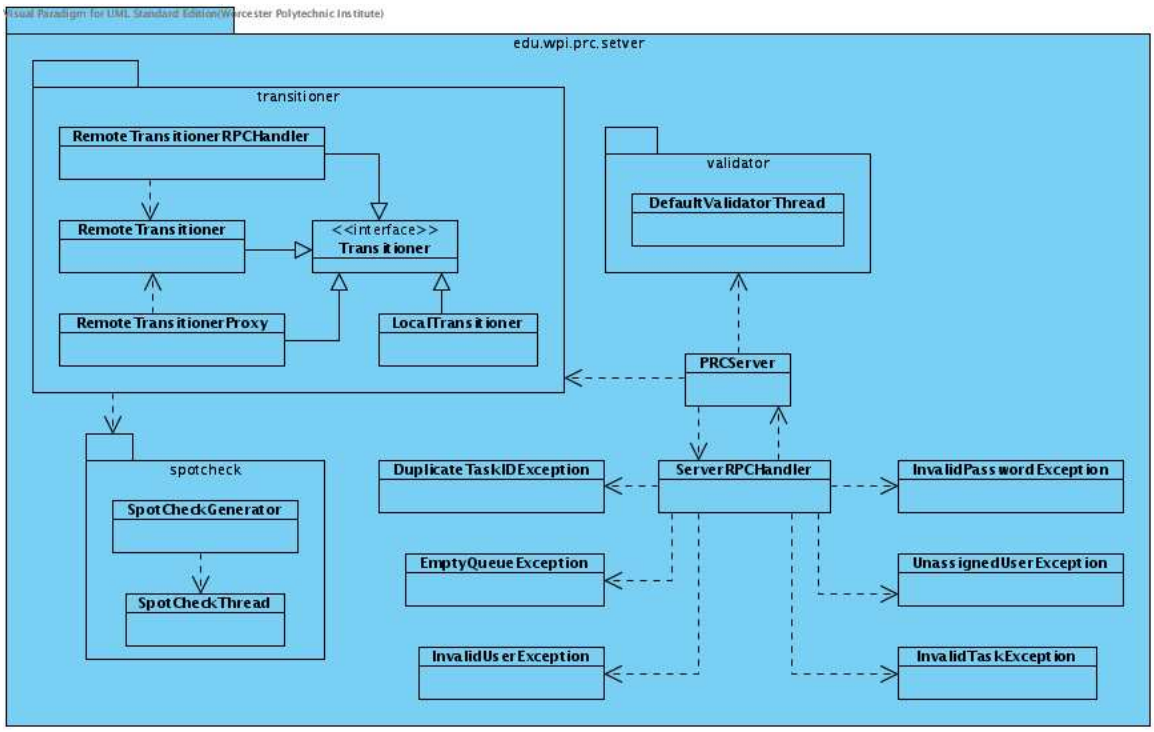

Figure 12: Server Package Diagram

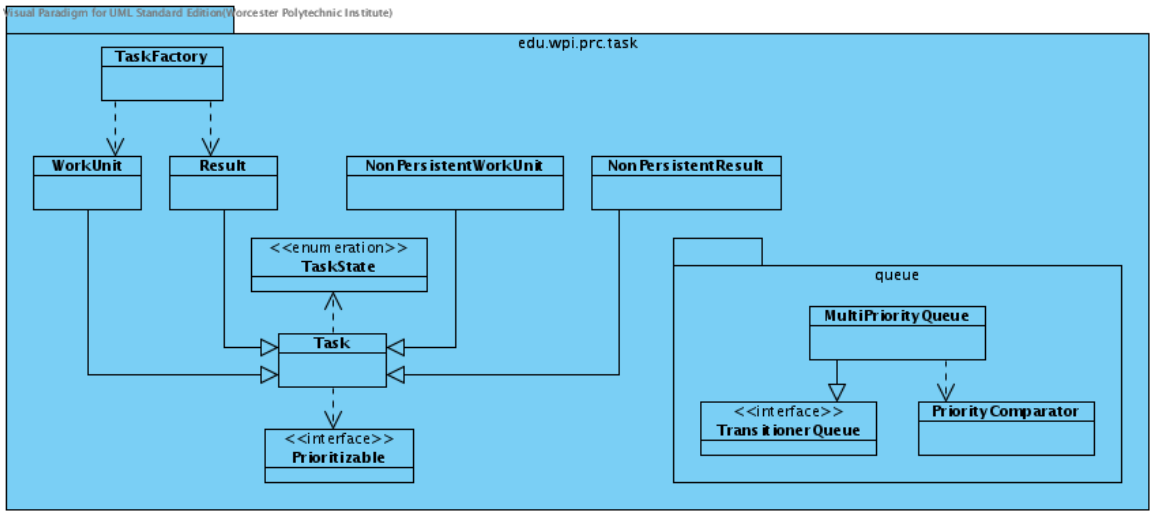

Figure 13: Task Package Diagram

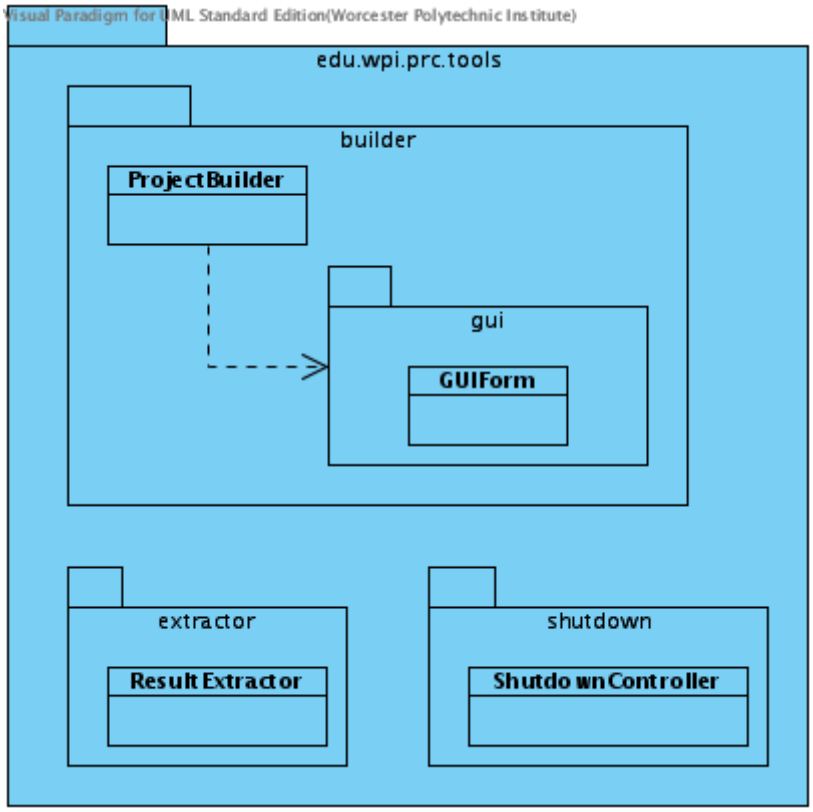

Figure 14: Tools Package Diagram

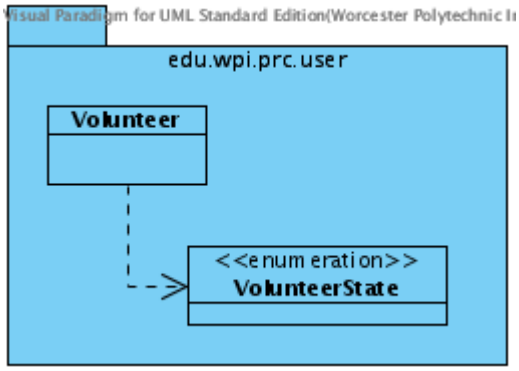

Figure 15: User Package Diagram

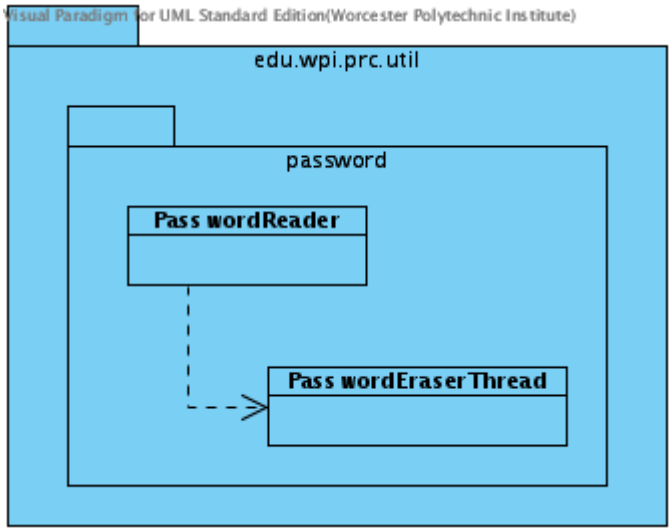

Figure 16: Util Package Diagram

Appendix B: Project Creation Guide

There are four main tasks that need to be performed to create a public resource computing project using SLINC: configuring a new project, developing project-specific components, preparing project distribution files, and deploying the project. Many of these steps include automation to facilitate the project development process, but due to the flexibility of our project there are many configuration options. This guide provides step-by-step instructions for creating a new public resource computing project. In order to use SLINC, you must have version 1.5 or later of the Java Runtime Environment (JRE). To compile the Java example project components, you will need version 1.5 or later of the Java Development Kit (JDK). If you do not have the correct version of Java installed, you can obtain it from Sun's Java website: http://java.sun.com.

Step 1: Configuring a New Project

First, obtain the latest release of the public resource computing framework from the following URL:

https://sourceforge.wpi.edu/sf/frs/do/listReleases/projects.jdb_prc_thesis/frs.prcf

There are two formats for each release, a tar.bz2 file and a zip file. The tar.bz2 files achieve better compression and are recommended for users who will create a PRC project on a UNIX-like system because tar.bz2 files retain the proper execute permissions on the scripts. On a UNIX-like system, use the command **tar xjf prcf.tar.bz2** to extract the archive. The file size may seem large, but most of that space is being used by the libraries needed to build the C++ examples as well as the libraries for connecting to various types of databases.

After you have obtained the latest release of the framework, you should choose a convenient directory to work from and extract the framework release there. Note that the platform on which the project is developed has no effect on which platforms the project server and client may run on. A project developed on a Linux system can run on UNIX-like platforms, Windows platforms, or both. Similarly, a project developed in Windows can run on both Windows and UNIX-like systems. The framework itself is cross-platform, but your project will only be cross-platform if you decide to create project components that can run on both UNIX-like and Windows systems. If you develop these components in a cross-platform language like Java or Python, there is no additional work required to make your project compatible with several different platforms.

We will now explain the process of configuring a new project using the *Project Builder* tool. Find the files that begin with *project_builder* in the location where you extracted the archive. We will refer to any files with the extension *.sh* or *.bat* as scripts. The scripts ending in *.sh* are for use on UNIX-like systems, and those ending in *.bat* are for use on Windows systems. Start the *project_builder* by invoking the appropriate script for your platform. In Windows this can be done by double-clicking the *project_builder.bat* file or by navigating to the correct directory in the Windows command prompt, typing *project_builder.bat*, and then pressing the *Enter* key. On UNIX-like systems the script should be started by navigating to the framework directory

in a shell, then typing *./project_builder.sh*. Running the *project_builder* script will start the Project Builder tool that is used to configure new or existing projects.

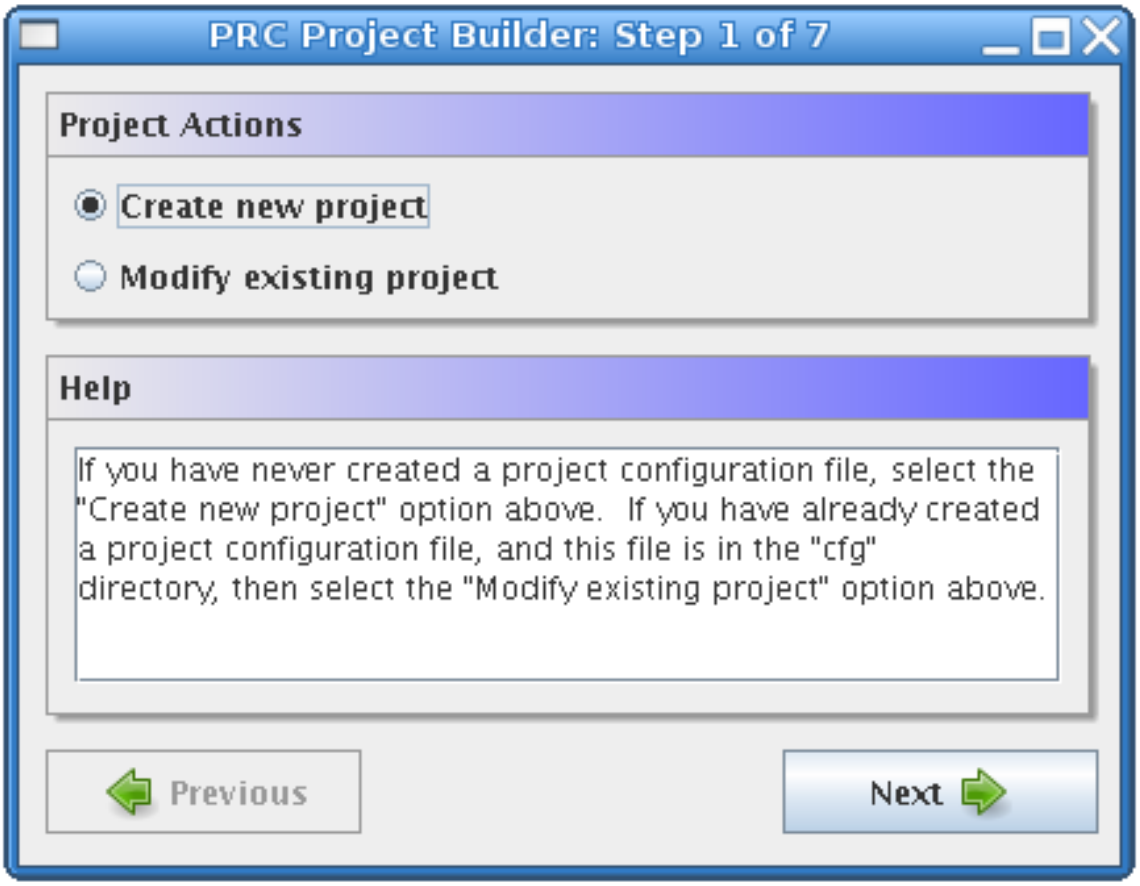

Figure 17: Project Builder Tool, Project Actions

The first form in the Project Builder asks whether you want to create a new project or modify an existing project. In this case, select *Create new project*, and click the *Next* button.

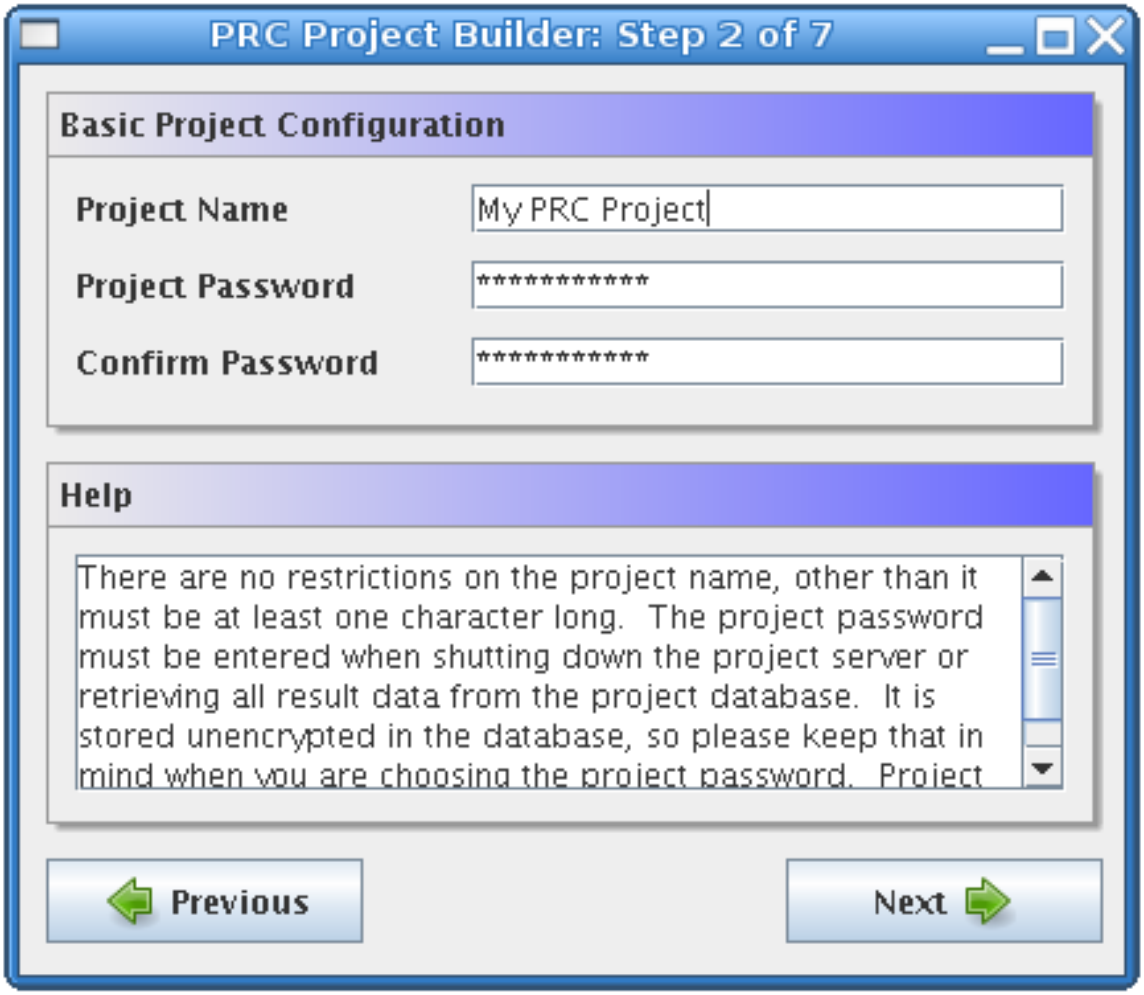

Figure 18: Project Builder Tool, Basic Project Configuration

On the second form you must select a project name and a password for the project. There are no restrictions on the name except that it must be at least one character long. The password is used when you want to perform certain restricted actions on the project. Currently, the only restricted actions are shutting the project server down and requesting data about all completed work units and results. Shutting down the project server must be done via an XML-RPC request; a password is required so that only project administrators are able to perform this action. Retrieving all completed work units and results can result in a large database and network load, so this function is also restricted. The password must be at least six characters long to discourage brute-force password cracking attempts.

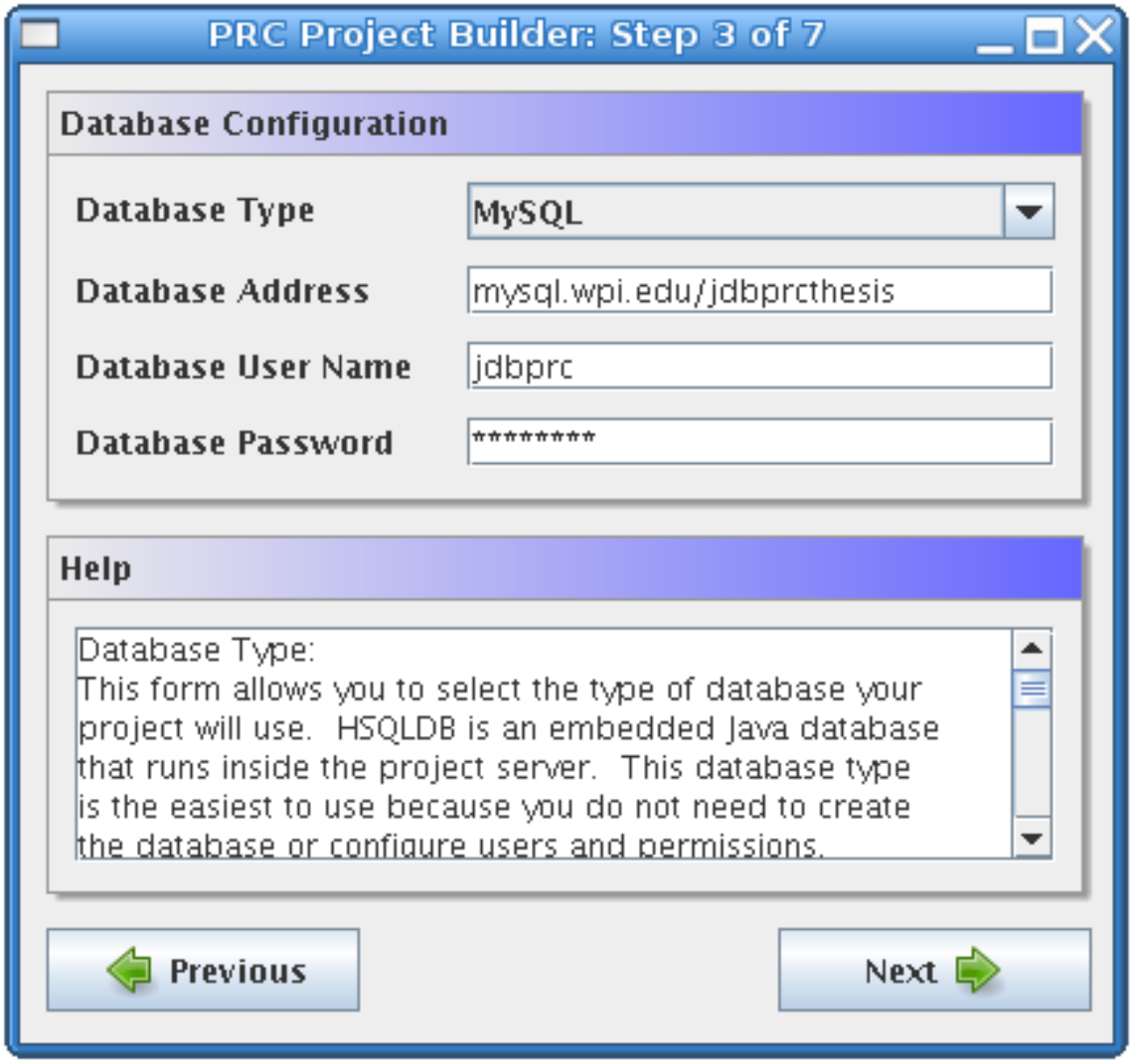

Figure 19: Project Builder Tool, Database Configuration

The third form allows you to configure the project database. The framework supports five types of databases: HSQLDB, MySQL, PostgreSQL, Oracle, and Microsoft SQL Server. HSQLDB is an embedded Java database that stores its data in a simple file format. Using HSQLDB is the easiest option because it does not require manually installing the database software, creating a database, and adding a user. However, the other types of databases probably outperform HSQLDB. Another limitation of HSQLDB is that only one process may access it at a time, so it is not possible to use a remote transitioner if your project uses an HSQLDB database. If you choose to use HSQLDB, the *Database Address* should be the location where the HSQLDB data files will be stored on disk. This address can be a relative or absolute file path. It is recommended that you place these files in the *data* directory; an example address for an HSQLDB database is: *data/prcdb*. If you want to use one of the other types of databases, you will be required to configure the database yourself, including installing the database software, creating a database for your project, and creating a user who has permissions to *CREATE*, *DROP*, *SELECT*, *INSERT*, *UPDATE*, and *DELETE* on that database. This database must be created before completing the last step in the Project Builder. To use a non-HSQLDB

database you will need to enter the URL of the database in the *Database Address* field. The database address usually consists of the hostname of the computer that hosts the database followed by a forward slash, and then the name of the database. It is important to note that the framework may not be able to establish the database connection if an IP address is used instead of a hostname. An example of a MySQL database address is: *mysql.wpi.edu/jdbprcthesis*. The computer or computers on which the project server components will run must be able to access the address you enter. The *User Name* and *Password* fields must contain the user name that should be used to access the project database and the password associated with that user. It is not necessary to have a user name and password for HSQLDB databases.

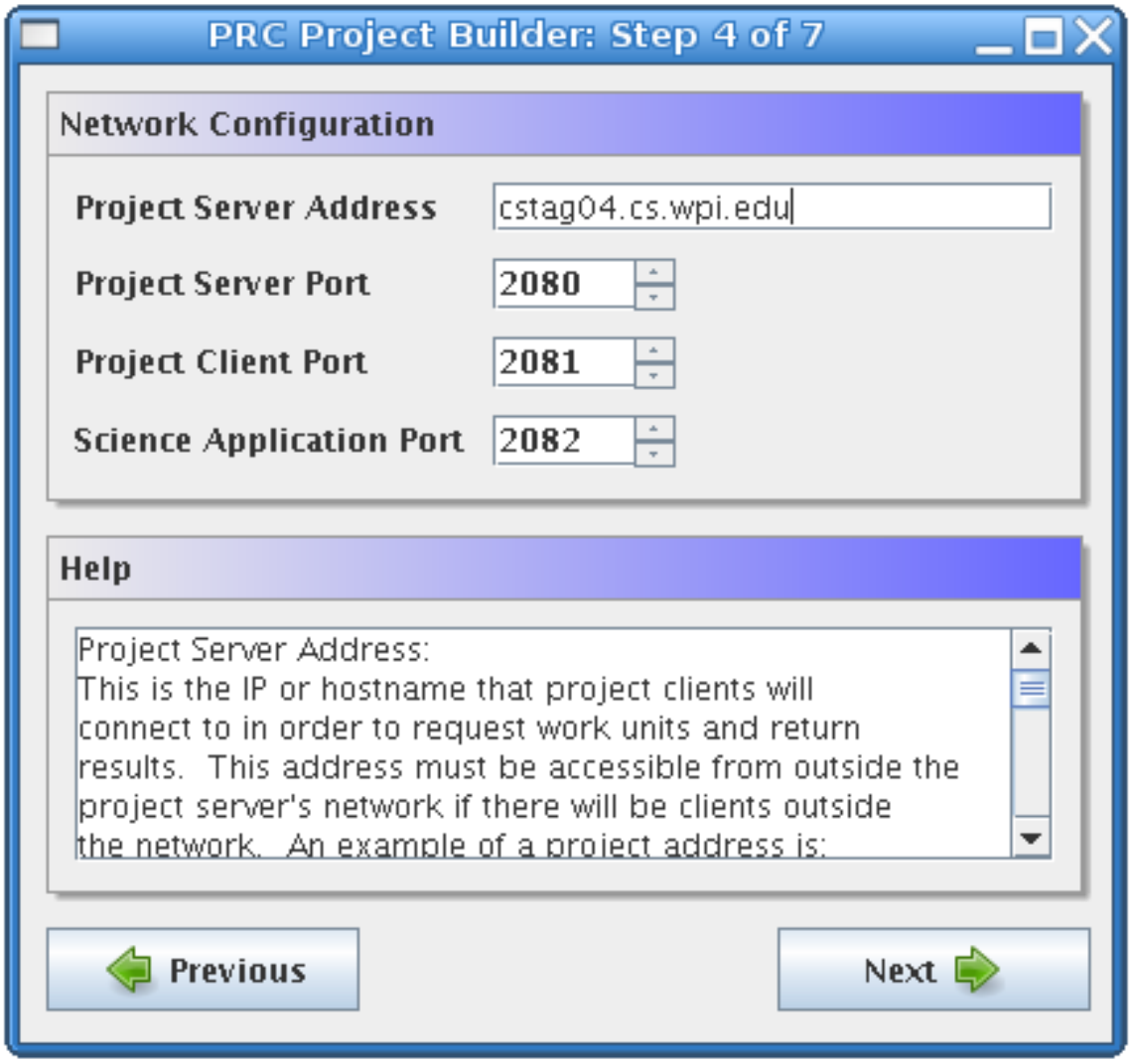

Figure 20: Project Builder Tool, Network Configuration

The fourth form in the Project Builder is used for configuring project network settings. The *Project Server Address* field is the IP address or fully qualified domain name that all project components will use to contact the server. Examples of valid server addresses are cstag04.cs.wpi.edu or 130.215.29.35. Of course the computer on which the project server will run must actually have this address or the other components will not

be able to connect to the project server. Also note that all server components and client components must be able to access this address. If there will be clients outside of your local area network, you must make sure that they will be able to connect to the server address that you will use. The *Project Server Port* is the TCP port on which the project server will listen for XML-RPC connections from the project client and other server components. The *Project Client Port* is the TCP port on which the project client will listen for XML-RPC connections from the science application. The project client listens on the localhost address, so there is no need to configure this parameter. The *Science Application Port* is the TCP port on which the science application will listen for XML-RPC connections from the project client or server. Since the science application is not provided by the framework, the science application developer must make sure that the science application actually listens on this port. It is also important to verify that the port numbers you have chosen will be available on the computers on which each component will run. Keep in mind that the science application will run on the same computer as the project client, and the science application may also run on the same computer as the project server if the project is configured to use spot-checking. Since different components may run on the same machine, it is necessary to use different port numbers for the project server, project client, and science application.

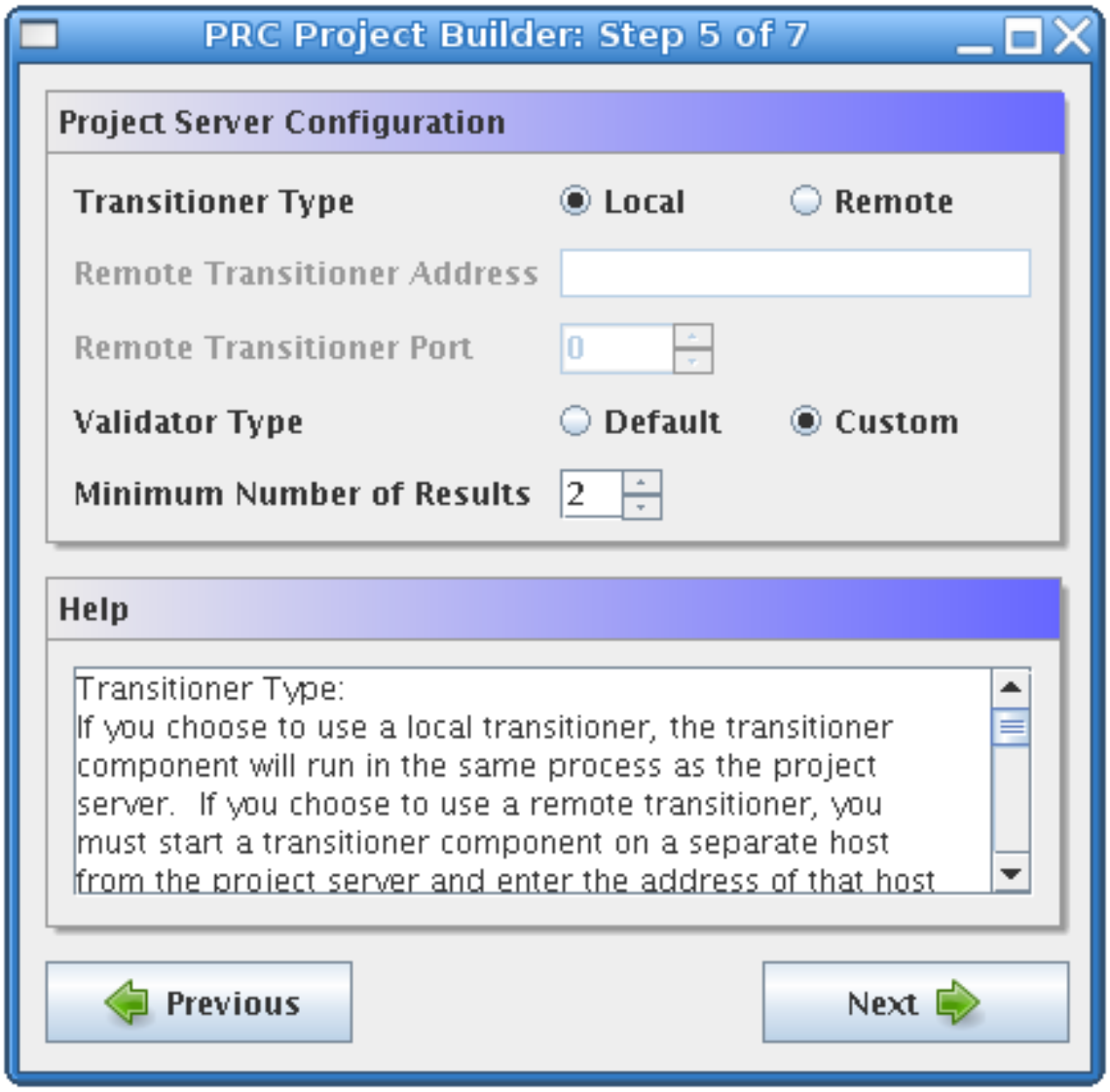

Figure 21: Project Builder Tool, Project Server Configuration

The next form is the project server configuration. The first part of this form is the transitioner configuration. The transitioner is a server component that keeps track of all work units and results in the project. You will almost always want to use a local transitioner, which will cause the transitioner to run as part of the project server. However, if the computer on which the project server is running is experiencing extreme performance problems, you have the option to run the transitioner on a different computer, as long as that computer can connect to the project server. To use a remote transitioner, click the *Remote* radio button, and then enter the address for the computer on which the transitioner will run. The format of the transitioner address is the same as that of the project server. For example, a valid transitioner address would be cstag04.cs.wpi.edu or 130.215.29.35. The *Remote Transitioner Port* is the port number on which the transitioner should listen. This port should be different from all of the other ports you have configured. The next part of the form is for configuring the type of result validator your project will use. You can either use the default validator, which simply

marks all results returned by clients as valid, or you can choose to implement your own result validator. If you want to implement your own result validator, choose *Custom* for the validator type. You do not need to configure any address or port for the custom validator because the server never needs to connect to the validator. The validator simply polls the server on a regular basis, asking if there are any results that need to be validated. The last part of the form is the *Minimum Number of Results* value. This setting controls the number of results that must be received for each work unit before that work unit may be retired. You may want to use a value greater than one if you are writing a custom validator. In that case, your validator will be able to compare all of the results that have been returned for the same work unit and choose a single one of those results to be the *canonical result*. The canonical result is the one result that is accepted for a given work unit. All other results for that work unit are discarded after the canonical result has been chosen. Using a minimum number of results that is greater than one with a custom validator can be used as a way of identifying incorrect results by comparing results returned by different users. It is important to realize that, as the minimum number of results increases, the efficiency of your public resource computing project decreases because the same work unit is being computed multiple times. Also, if the number of users participating in the project is smaller than the minimum number of results, no work units would ever be retired because a work unit cannot be assigned to the same user twice.

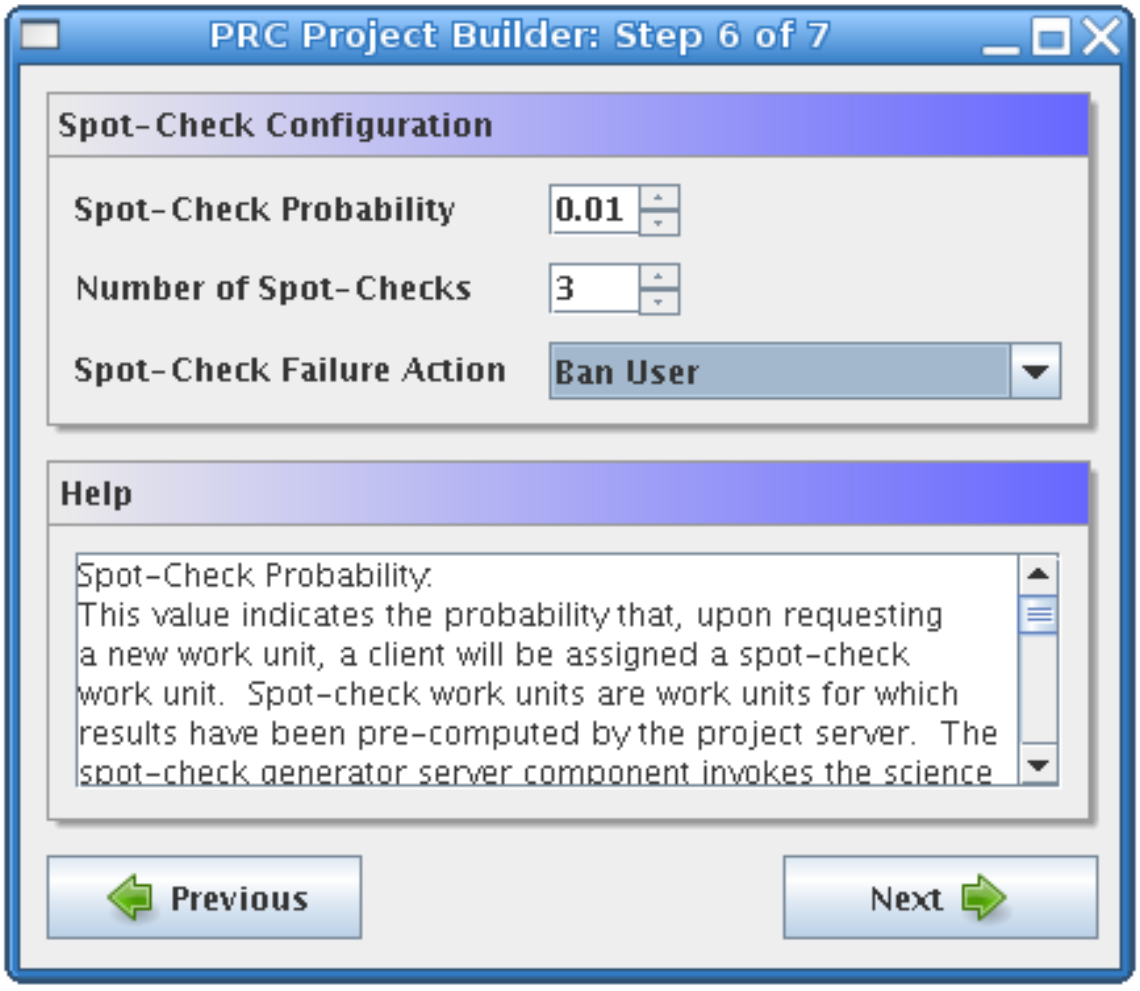

Figure 22: Project Builder Tool, Spot-Check Configuration

The sixth form is spot-check configuration. A spot-check is a method used to identify malicious users, or *saboteurs*. A saboteur is defined as a user who deliberately sends invalid results to the project server. When spot-checking is used, the project server computes the result for one or more work units by invoking the project's science application. The results it computed are referred to as *spot-check results* and are assumed to be valid because they have been computed by a trusted source, the server. The work units from which the spot-check results were computed are called *spot-check work units*. Once one or more spot-check results have been computed, the server can send a spot-check work unit to a client, which will not know that the work unit it received is a spot-check. The validator can then compare the result returned by that client to the accepted spot-check result. If the validator determines that the result returned by the client is not valid, the user who submitted that result is identified as a saboteur. Once a saboteur has been identified, the project server can take actions to mitigate the risk introduced by that user. The first value in the form, *Spot-Check Probability*, allows you to specify the probability that any given client will be assigned a spot-check work unit when it requests a new work unit. This value should be between zero and one. The next configurable parameter is the *Number of Spot-Checks*. This value specifies how many spot-check results the server should generate. You may want to have more than one spot-check if

you are concerned that a single spot-check work unit will eventually be identified by saboteurs, who will return the correct result upon receiving that work unit, circumventing the spot-check. Generally you will want to specify a small value here, between one and three, unless you suspect that saboteurs are trying to identify the spot-check work units in order to evade detection. The *Spot-Check Failure Action* determines what course of action the project server will take once a saboteur has been identified. If a saboteur is identified, this event will be logged to the spot-check log file, called *spotcheck.log* in the *log* directory. If this is the only action you wish the server to take, select *Log Only*. Another option is to ban the identified user. When a user is banned, that user will no longer be able to request new work units or return results. The third option is to both ban the user and delete all results that user has previously returned. This is the most extreme action you can take against identified saboteurs. If the saboteur submitted a result that was later chosen as the canonical result for a work unit, deleting that result means that the associated work unit will have to be recomputed by other users.

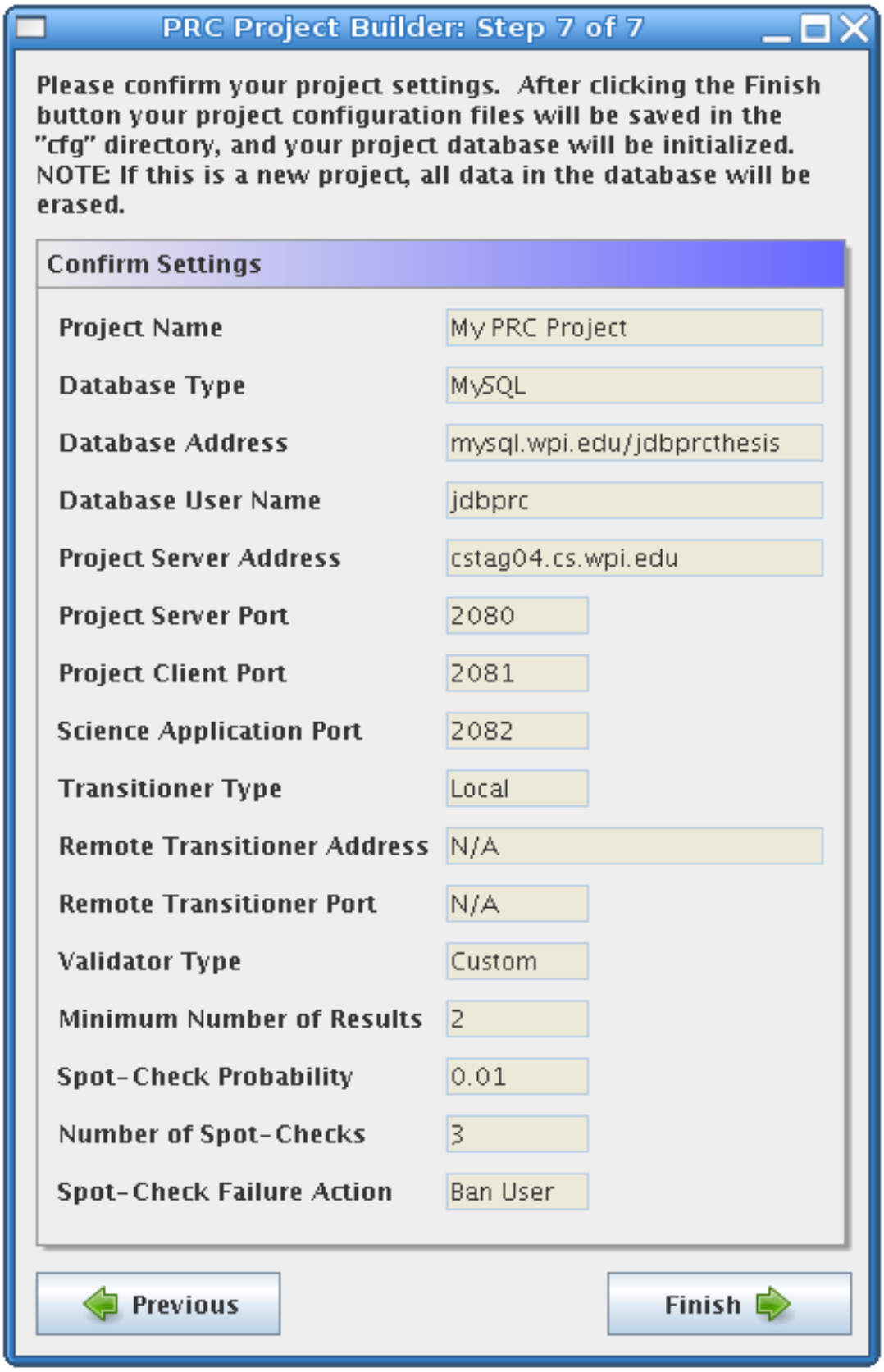

Figure 23: Project Builder Tool, Confirm Settings

The last step in the Project Builder is to confirm your settings. At this point you can go back to any previous form and change the information. When you have confirmed your settings, the project configuration files will be saved to disk in the *cfg* directory, and the project database will be initialized. Please note that initializing the project database will erase any data that it currently contains. If you are modifying an existing project, the database will have already been initialized, so this step will not be performed. When the Project Builder has finished configuring and initializing your project, you can begin developing project-specific components or preparing your project distribution files if you have already created the project-specific components.

Before developing your own project components, you may wish to first practice creating a project using the example components provided by the framework. The C++ and Java example components are located in the *example* directory after the framework file has been extracted. There is a folder for the C++ implementation and a separate folder for the Java implementation. Each of these folders has a *build* script that compiles the example source files and, if necessary, any libraries that are needed by the examples. The script also creates *client* and *server* directories. To create a project using the example components, first run the *build* script in either the C++ or Java example directory, then copy the *client* and *server* directories with all of their subdirectories to the main framework directory. The main framework directory is the directory where you extracted the framework. You can then continue to **Step 3: Preparing Project Distribution Files**. Note that you must have the Java Development Kid (JDK) version 1.5 or later installed to compile the example Java project. Furthermore, if you are building the Java example on a Windows platform, the **javac** executable must be in your path. You will know that **javac** is not in your path if, after executing the *build* script, there are no *.class* files in the example Java directory. To add this program to your path, navigate to the Control Panel, and open the System configuration. Select the Advanced tab, and click on the Environment Variables button. Select the Path variable, and click the Edit button. Add a semi-colon at the end of the line, the paste in the path to the *bin* directory of your JDK installation. For example, the path might be C:\Program Files\Java\jdk1.5.0_06\bin. Click OK when you are finished, and the **javac** program should be in your path. You can now execute the *build* script in the Java example directory.

Step 2: Developing Project-Specific Components

Project-specific components are executables that are not provided by SLINC, but instead must be implemented separately by each project. Most public resource computing projects have a large set of data that can be processed in smaller subsets. We will use the term *science data* to refer to the entire set of data that needs to be processed by a public resource computing project. In order to process all of the science data in a distributed way, the data must be partitioned into many subsets, and there must be an algorithm that accepts a subset as input and computes some result as output. The results produced by this algorithm can later be recombined to draw meaningful conclusions about the original data set.

There are two components that each project must develop to in order have a complete public resource computing project: a *work unit generator* and a *science application*. The work unit generator partitions your data set into smaller subsets called

work units, which can be of any size. The science application takes a work unit as input, executes your project-specific algorithm on the data, and produces a *result* as output, which also does not have any size limitation.

In addition to the required components there is one optional component that you may wish to develop: the *result validator*. The result validator has three responsibilities. The first is to determine whether each individual result for a work unit is valid. The second is to decide whether a client has passed or failed a spot-check. The last responsibility is to choose a single result from the set of all results for a work unit, and to mark that result as *canonical*. When a canonical result is chosen, the canonical result will be saved permanently in the database, and all other results for that work unit will be deleted. Although the result validator must perform all three tasks, one or more of these tasks can be *stubbed*. For example, instead of writing an algorithm to determine which result should be selected as the canonical result, you can simply always designate the first result that was received as the canonical result, which is what the default validator does. That way the validator would have fulfilled its responsibilities, but you would have only had to write a few lines of code. Of course this assumes that the result that is selected as the canonical result is not important for your project. Any or all of the three validator tasks can be stubbed in this way, so you only need to implement the functionality that is necessary for your project to be successful.

Information about how to implement each of these components can be found in Appendix C: Project Programming Guide. Once you have implemented the necessary project components, please continue to **Step 3: Preparing Project Distribution Files**.

Step 3: Preparing Project Distribution Files

After all of your project-specific components have been developed, it is necessary to package them, along with the framework, into server and client distribution files. Once these files have been created you will be able to uncompress them on a server or client computer and execute the appropriate components more easily. In order to build these files you will need to copy your project-specific components into the correct framework directories. When you extracted the framework archive in step one, the *server* and *client* directories should have been created. The server directory is where all server components should be placed, and the client directory is where your science application should be. First, copy your work unit generator, and optionally your result validator, into the server directory. Note that you will need to copy all binaries, libraries, and other files required for your components to execute, but not necessarily the source code. You may also want to write a script in that directory for easily starting and stopping these components. The framework does not automatically start and stop your server components because you may want them to execute on a separate computer from the server.

Next, copy your science application into the client directory. Again, verify that all necessary files and libraries have been copied into that folder. After you have copied your client components into the client directory you need to create one or more scripts in that directory using a specific naming convention. The purpose of these scripts is to begin execution of your science application. These scripts are run by the project client in the event that it receives a new work unit, but your science application is not running. If your client is designed to run only on Windows systems, create a script called

run_sci_app.bat with the appropriate Windows commands to start your science application. If your client is only meant to run on UNIX-like systems, you need to create a script called *run_sci_app.sh* with the appropriate UNIX commands and the execute permissions set, which is very important. If your science application is designed to run on both types of platforms, you will need to write both a *run_sci_app.sh* script and a *run_sci_app.bat* script. The client will detect what type of platform on which it is running and will call the appropriate script for that platform.

As mentioned in step one, there are examples of client-side and server-side components in the *example* directory. You will find a directory containing a C++ example and another directory with a Java example. Run the *build* script in either one of those directories, and when it has completed, client and server directories will have been created that contain all the necessary files to execute the client-side and server-side example components. If you want, you can create distribution files for the example project by copying the contents of example/[c++ or java]/client/* to the client directory and example/[c++ or java]/server/* to the server directory. Both the C++ and Java examples have a *lib* directory that must be present for them to execute correctly, so it is important to copy this folder as well.

When the client and server components have been copied to the appropriate directories, and you have written the necessary *run_sci_app* scripts, the project distribution files can be built. Before creating the distribution files, the project must have already been configured using the Project Builder, and the project configuration files *project_server.properties* and *project_client.properties* must be in the *cfg* directory. Please see step two if you have not yet configured the project. To create the project distribution files, simply run the *make_project_files* script that should be in the main framework directory. This script will build tar.bz2 and zip files for the client and server, which will appear in the current directory when the script has completed. The files will be named *prcf_client* and *prcf_server* followed by the tar.bz2 or zip extension. These files should be used to deploy your public resource computing project.

Step 4: Deploying the Project

To deploy your project, copy the *prcf_server.tar.bz2* or *prcf_server.zip* file that you created in step three to the computer which will be the main server for your public resource computing project. If your server is a UNIX-like system, you should use the tar.bz2 file so that the scripts will have the correct permissions set. Choose a convenient directory for your public resource computing project, and uncompress the distribution file there. Start the server by running the appropriate *start_server* script for your platform. The server should output status information to the console, and you can also monitor its progress by inspecting the *server.log* file in the *log* directory. If your platform is Windows, and the Windows firewall asks you whether you want to block the Java program from accessing the network, choose to unblock it. As explained in step three, the framework does not provide any way to start your other server components, the work unit generator and result validator, because these components might be run on different computers. At this time you should start your server components, which should be located in the *server* directory. If your project uses a remote transitioner, you can use the *start_remote_transitioner* script to start that component. Your project should now be ready to accept client connections. Please be aware that if you need to shut down the

project server, you must use the *stop_server* script. Failing to do so may cause some or all of your data to be lost. If you execute the *stop_server* script, and several minutes later the server has not shut down, there may be a problem with one of the XML-RPC connections to your server that is preventing the shutdown from completing. In this case you may need to execute the *emergency_shutdown* script. This script will shut down the server immediately, without waiting for all client transactions to complete. The *emergency_shutdown* script should only be used as a last resort to shut down the server because it may cause inconsistency in the database if a transaction has begun but has not yet completed.

If your project uses an HSQLDB database, and you need to view the contents of the database directly, you can use the *hsqldb_client* script that is included in the server distribution file. First, you will need to shut down the project server using the *shutdown_server* script. It is only necessary to shut down the server to view the contents of an HSQLDB database. All other types of databases should allow simultaneous access. After the server has been shut down, run the *hsqldb_client* script, which will start the HSQLDB client. Go to the *File* menu and select *Connect*. This action will display a connection dialog. The only field you need to modify is the URL field. It should contain the string *jdbc:hsqldb:file:* followed by the path to your HSQLDB database. For example, the URL might be *jdbc:hsqldb:file:data/prcdb*. After you connect, you will be able to execute SQL queries in the window at the top right of the HSQLDB client.

The volunteers who will contribute their computers' resources to the project should download the *project_client.tar.bz2* or *project_client.zip* file. After uncompressing these files, they should use the *start_client* script to start the project client. The first time the project client is executed, it will prompt the volunteer to enter his or her desired user name and a valid e-mail address. That information is then stored to the *cfg/client.properties* file, where it will be read on subsequent executions of the project client.

At some point you will want to access the result data that was computed by the science application. The server distribution file contains a script called *extract_results* for this purpose. When the *extract_results* script is run, you will be prompted for your project password. This script is password protected because it can cause a high network and database load, so it is more highly susceptible to a denial-of-service (DOS) attack. This script connects to the project database and retrieves the data for all results, and then writes this data out to disk. A folder called *results* will be created, which is the base directory for all results. Inside that directory, directories will be created for each work unit, and will be named by the ID of that work unit. Inside each work unit directory, a file called *result.dat* will be created. This file contains the canonical result data for that work unit.

Appendix C: Project Programming Guide

The purpose of this appendix is to provide specific information about how to write client and server components to interface with our public resource computing framework. The three types of components that can be developed separately for each public resource computing project are the *work unit generator*, *result validator*, and the *science application*. Each of these components communicates with SLINC via XML-RPC. A full specification of all XML-RPCs available to project developers can be found in

Appendix D: XML-RPC Interface Specification. We encourage all new project developers to look at the example components included with the framework; we believe that these are valuable resources when developing a public resource computing project for the first time with SLINC.

There are two ways in which the project components can be developed. We have developed template components that can be used to quickly build the project components. Alternatively, project developers can write all of the code for the components themselves. The first part of this guide describes how to use the templates to build project components, and the second part of the guide explains how to build the components without using the templates.

Developing Template-Based Components

We have provided component templates written in both C++ and Java. If a different language is desired, it will be necessary to develop all project component code using the second part of this guide, **Developing New Components**. The C++ and Java component templates are located in the *templates* directory where the framework distribution file was extracted. There are templates for the science application, work unit generator, and result validator. Navigate to either the *c++* or *java* directory inside the *templates* directory, and then continue to the appropriate section of this appendix: **Using the C++ Templates** or **Using the Java Templates**.

Using the C++ Templates

For the rest of this section we will call the *c++* directory inside the *templates* directory *$CTBASE*. The source code for the C++ templates is located in *$CTBASE/src*. In that directory there should be five directories: *common*, *lib*, *sci_app*, *validator*, and *work_gen*. The *lib* directory contains the source code for the necessary XML-RPC libraries, which will be built by the *build* script.

To create a work unit generator component using the templates it is necessary to edit the `WorkUnitGenerator` class, contained in *$CTBASE/src/work_gen/generator.cpp* and *$CTBASE/src/work_gen/generator.hpp*. The parts of the template files that will or might need to be changed are denoted by a *TODO* comment. There are three methods that need to be modified in the `WorkUnitGenerator` class: `recoverState`, `generateWorkUnit`, and `generateAndSend`. If the work unit generator were ever restarted, it would need some way to reinitialize its state information so that it could resume generating work units from the appropriate place in the science data. The `recoverState` method retrieves the last work unit that was generated from the server and uses that work unit to determine what work unit should be generated next. This method retrieves the last work unit using the provided `getLastWorkUnit` method of the `WorkGeneratorClient` class. If the project server throws an exception while retrieving the last work unit, it means that no work unit was ever generated, so the work unit generator should begin generating the first work unit.

The `generateWorkUnit` method in the `WorkUnitGenerator` class generates the next work unit and returns a byte vector representing the new work unit. It should also update some state information in the `WorkUnitGenerator` class to indicate the next work unit that should be generated.

The last `WorkUnitGenerator` method that needs to be edited is `generateAndSend`. This method generates a given number of work units by calling `generateWorkUnit` and then sends each work unit to the server by calling one of the overloaded `WorkGeneratorClient::sendWorkUnit` methods, which are declared in *$CTBASE/src/work_gen/work_gen_client.hpp*. All of the `sendWorkUnit` methods require the work unit byte vector as a parameter, but some allow extra parameters to be passed such as the work unit ID, the priority of the work unit, the point value of the work unit, and the amount of time before the work unit should expire. All of these variants are documented in both the `WorkUnitGenerator::generateAndSend` method and the `WorkGeneratorClient` class.

In addition to the three previously mentioned methods, project developers may wish to edit the work unit generator's `main` function, located in *$CTBASE/src/work_gen/work_gen.cpp*. The `main` periodically queries the project server to determine how many ingress work units it has. If the server has less than a certain number of work units, which we will refer to as the *low-water mark*, the `main` will invoke the `WorkUnitGenerator::generateAndSend` method to send more work units to the server. The `main` can be edited to change how often the server is queried, change the low-water mark, or the number of work units that are generated and send to the server when the low-water mark is reached.

The next component that should be modified is the science application, located in *$CTBASE/src/sci_app*. The class that needs to be modified is `ScienceDataProcessor`, located in *$CTBASE/src/sci_app/sci_data_proc.cpp* and *$CTBASE/src/sci_app/sci_data_proc.hpp*. There are two methods that need to be modified: `run` and `scienceAlgorithm`. The `run` method takes a byte vector representing the work unit to compute as its parameter. If the science application uses check-pointing, the first action that should be taken by the `run` method is to check for previously saved check-points by calling the `getCheckpoint` method of the `ScienceApplicationClient` class. This method will return a byte vector representing the check-point. The template demonstrates how to do this. If a check-point vector is of length zero, it means that no check-point was found. If a check-point was found, the `run` method should set some private member variables to indicate where in the work unit to resume processing. The next action taken by the `run` method is to initialize and start a low-priority thread, called the *compute thread*, in which the `scienceAlgorithm` method will execute. This is also demonstrated in the science application template. Any data needed by the science algorithm will have to be passed as a `void*` because that is the way the *pthreads* library handles parameter passing to threads. After starting the compute thread, the `run` method should join with the compute thread, causing the `run` method to block until the compute thread has terminated. The compute thread will return a `void*` representing the result for the work unit, which will then have to be cast to the appropriate type and converted to a byte vector.

The `scienceAlgorithm` method uses information passed by the `run` method to compute the result for a work unit. If the science application uses check-pointing, the `scienceAlgorithm` method should periodically save a check-point by calling the

`saveCheckpoint` method of the `ScienceApplicationClient` class, as demonstrated in the template. When the computation of the result has completed, the `scienceAlgorithm` method must return the result as a `void*` to the `run` method, which will convert it to the appropriate type.

Implementing the result validator component is optional, but the template for this component can be found in *$CTBASE/src/validator*. The class that needs to be modified is *ResultValidator*, located in *$CTBASE/src/validator/result_validator.cpp* and *$CTBASE/src/validator/result_validator.hpp*. There are three methods in the `ResultValidator` class that must be implemented: `validateSingleResult`, `selectCanonicalResult`, and `validateSpotCheck`. The `validateSingleResult` method examines the data from a result and decides whether or not it is valid. This method should return true if the result is valid or false if the result is invalid. If necessary, it is possible to examine the work unit from which the result was computed using the `ResultValidatorClient` class; this is demonstrated in the template source. The `selectCanonicalResult` method examines all of the valid results for a work unit and determines which one of those results should be chosen to be the canonical result. This method should return the result ID of the result that was chosen. It is possible to examine the work unit from which the results were computed in the same way as was done when validating a single result. The `validateSpotCheck` method compares the spot-check result computed by a client to the accepted spot-check result that was computed by the project server to determine whether the client passed the spot-check. This method should return true if the client passed the spot-check or false if the client failed the spot-check. Again, it is possible to examine the spot-check work unit from which the spot-check result was computed.

Using the Java Templates

For the rest of this section we will call the *java* directory inside the *templates* directory *$JTBASE*. The source code for the Java templates is located in *$JTBASE/src*. In that directory there should be four directories: *common*, *generator*, *science*, and *validator*.

We will first examine the work unit generator template, located in the *$JTBASE/src/generator* directory. The class that needs to be edited to create a functional work unit generator is the `WorkUnitGenerator` class, which can be found in the file *$JTBASE/src/generator/WorkUnitGenerator.java*. The best way to learn what needs to be implemented is to examine the `WorkUnitGenerator` class, which is thoroughly documented, and contains the string *TODO* wherever there is something that may need to be implemented or changed. There are at least two methods that must be implemented in that class: `recoverState` and `generateWorkUnit`. If the work unit generator were ever restarted, it would need some way to reinitialize its state information so that it could resume generating work units from the appropriate place in the science data. The purpose of the `recoverState` method is to reinitialize this state information. This method uses the supplied `WorkGeneratorClient` class to retrieve the last work unit that was generated from the server via XML-RPC. If the server throws an Exception, it indicates that there were no previously generated work units, so the work unit generator should begin generating the first work unit. Otherwise it will return a work unit, which can be inspected to determine where to resume generating the next work unit.

The other `WorkUnitGenerator` method that needs to be edited is the `generateWorkUnit` method. This method generates a work unit from the science data, converts that work unit into a byte array, and returns that byte array to the calling method. Another method that may need to be edited is `sendWorkUnit`. This method uses the `WorkGeneratorClient` class to send the generated work unit to the project server. However, there are several options for how the work unit can be added. The simplest way is to send only the byte array containing the work unit data to the server. It is also possible to specify other options, such as the desired work unit ID, priority, point value, and expiration time. There are several combinations of these options that can be used, with each combination corresponding to a call to a different overloaded `WorkGeneratorClient.sendWorkUnit` method. These methods are documented in `sendWorkUnit`. The last aspect of the template that may need to be changed is the `main` method located in *$JTBASE/src/generator/Generator.java*. The `main` controls when the work unit generator should generate more work units and how many new work units should be generated at once.

The next component template that needs to be modified is the science application template, located in the *$JTBASE/src/science* directory. To create a working science application from the template it is necessary to edit the `ScienceDataProcessor` class in the *$JTBASE/src/science/ScienceDataProcessor.java* file. The methods in that class that will need to be edited are `computeResult` and `scienceAlgorithm`, although these two methods could be combined into a single method if desired. The `computeResult` method will be started in its own low-priority thread by the `ComputeThread` class. The work unit data, a byte array, is passed into the `ScienceDataProcessor` class in its constructor. If the science application uses check-pointing, the `computeResult` method should first attempt to retrieve the last check-point from the project client; the template code for the `computeResult` method demonstrates how to do this. If any check-point was found, it should be used to initialize the science algorithm, possibly by setting certain private member variables. The `computeResult` method should then execute the science algorithm on the work unit and produce a result. The result has to be converted into a byte array before being returned by the `computeResult` method.

It is possible to perform all necessary computations in the `computeResult` method, but it is usually a cleaner design to use a separate method to actually execute the science algorithm; the `scienceAlgorithm` method is used for this purpose. The intended design is to use the `computeResult` method to interpret the work unit and perform any initialization that needs to occur. Then the `computeResult` method can call the `scienceAlgorithm` method to compute the result for the work unit. After a result has been computed, the `computeResult` method can convert the result into the byte array that needs to be returned. During the execution of the science algorithm, check-points can periodically be saved by the project client. The template source for the `scienceAlgorithm` method demonstrates how to do this.

The third template component, the result validator, is optional. If result validation is required, this component can be implemented. The template for the result validator is located in *$JTBASE/src/validator*. The class that needs to be edited is `ResultValidator`, which is located in the file

$JTBASE/src/validator/ResultValidator.java. There are three methods that need to be implemented, each corresponding to a different type of validation: `validateSingleResult`, `selectCanonicalResult`, and `validateSpotCheck`. The `validateSingleResult` method decides whether a single result is valid by examining the result data. The method should return true if the result was valid or false if the result was not valid. If necessary, it is possible to retrieve the work unit from which that result was generated; this operation is demonstrated in the template source. The `selectCanonicalResult` method examines the set of valid results returned for a work unit and selects one of those results to be the canonical result. The method should return a string, the result ID of the canonical result. If it is necessary to examine the work unit for which the results were computed, it is possible to retrieve that work unit data as demonstrated in the template source. The `validateSpotCheck` method decides whether a client passed a spot-check. It does so by comparing the spot-check result data computed by the client to the accepted spot-check result data that was computed by the project server. As in the other two methods, it is possible to examine the work unit from which the spot-check result was computed. This method should return true if the client passed the spot-check and false if it failed the spot-check.

Building Template-Based Components

After modifying the templates to add the required functionality, it is simple to build the template-based components so that they may be used in a project. There are separate *build* scripts for the C++ and Java templates, located in the base of each template directory, *$CTBASE* and *$JTBASE*. It may be necessary to modify the build script or makefile if additional classes have been added or if additional libraries are required. When the *build* script is executed, it will first compile all necessary binaries; then it will cerate *client* and *server* directories that have all required binaries and libraries. These directories are analogous to those created by the *build* script for the example project. The *client* and *server* directories should be copied to the directory where the framework was extracted, at which point the *make_project_files* script can be used to build the *.tar.bz2* and *.zip* files as described in Appendix B: Project Creation Guide.

Developing New Components

Before explaining each component in detail, we will present some conventions that will be used throughout this appendix, as well as a brief introduction to XML-RPC. All text that contains source code will be written in the `Courier New` type face. We assume that the Apache XML-RPC[28] library will be used for Java applications or the xmlrpc-c[29] library will be used for C and C++ applications. The information in this appendix applies to all XML-RPC implementations, but the source code is specific to those two libraries. The Apache and xmlrpc-c libraries require that a parameter always be passed to the XML-RPC call. However, some of the XML-RPC handlers that are being called do not take any parameters. The solution is to pass an empty parameter to the XML-RPC library when executing RPCs that do not take any parameters. In Java, parameters are passed as `java.util.Vector` objects; passing an empty parameter means passing a `Vector` that contains zero elements, as in `new Vector()`. In C++, parameters are passed as `xmlrpc_c::paramList` objects. To pass an empty

parameter, create a `paramList` object without using the `new` operator, and then pass that newly created object to the appropriate XML-RPC method.

Although the `xmlrpc_c::paramList` object is used to send parameters to an XML-RPC with the xmlrpc-c library, the return value of an XML-RPC is often an `std::vector` object. The return value will be an `std::vector` if it is a compound value, for example a string and two integers. To avoid using implementation-specific terminology, and to keep a consistent style, we will simply refer to compound return values as vectors. The meaning of the term *vector* should be interpreted appropriately based on the programming language and XML-RPC library you have chosen to use.

The way in which most XML-RPC libraries handle parameters is not necessarily intuitive, so it deserves further explanation. No matter how many parameters an RPC accepts, the client executing that RPC must wrap them in a vector. Recall that in the context of building parameters for an XML-RPC, the term *vector* may mean `java.util.Vector`, `xmlrpc_c::paramList`, or some other type, depending on your programming language and XML-RPC library. Also, even though the client makes the XML-RPC call by passing a vector, the RPC handler itself should not accept a vector as its parameter, but rather the object types contained within the vector. For example, consider an RPC that accepts two strings followed by an integer, and returns a boolean. The client executing that RPC would create a vector and add the strings and integer in the specified order. The signature of the RPC handler, however, should look like the following:

```
boolean HandlerName(String arg0, String arg1, int arg2)
```

Notice that the signature of the handler does not take a vector, but the types contained within the vector. The XML-RPC library removes the elements from the vector, searches for a handler with a matching signature, and then calls that handler, passing in the elements that were extracted from the vector.

It is important to be familiar with XML-RPC before attempting to develop any project components. The following block of code demonstrates how to execute an XML-RPC using the xmlrpc-c library in C++. For complete examples, please see the *example* directory in the latest release of the framework.

```
// The HTTP connections will be made using libwww.
xmlrpc_c::clientXmlTransport_libwww xmlrpcTransport;

// Create the XML-RPC client.
xmlrpc_c::client_xml xmlrpcClient(&xmlrpcTransport);

// Create an "empty parameter."
xmlrpc_c::paramList params;

// Create an RPC with the name of the RPC handler and the
// parameter list.
xmlrpc_c::rpcPtr xmlrpc("server.getLastWorkUnit", params);

// Set the URL of the XML-RPC server.
// Note the format, http://host:port/RPC2
```

```
xmlrpc_c::carriageParm_curl0 cParm("http://localhost:2080/RPC2");

// Execute the RPC.
try {
      xmlrpc->call(xmlrpcClient, &cParm);
} catch (char const* e) {
      cerr << "Exception thrown while making XML-RPC: ";
      cerr << e << endl;
      exit(1);
} catch (girerr::error e) {
      // The host could not be found, or the connection
      // was refused.
      cerr << "ERROR: Could not connect to the XML-RPC ";
      cerr << "server.  ";
      exit(1);

}

if (xmlrpc->isSuccessful() == false) {
      // Some XML-RPC problem occurred.
      // Ex. RPC not found on server.
      // Ex. Incorrect number of parameters passed to RPC.
      // Ex. Server threw an exception (not necessarily a
      // problem).
      if (xmlrpc->isFinished()) {
            // NOTE: xmlrpc->isFinished() must be true
            // before calling xmlrpc->getFault().
            xmlrpc_c::fault err = xmlrpc->getFault();

            // err.getCode() will be 0 if an exception was thrown
            // by the server.
            if (err.getCode() == 0) {
                  // Take some action if exceptions are expected.
            }
            else {
                  // An exception was not thrown by the server.
                  // Some other problem has happened.
                  // Read the error message.
                  xmlrpc_c::fault err = xmlrpc->getFault();
                  cerr << "recoverStartRange: XML-RPC error ";
                  cerr << err.getCode() << ": ";
                  cerr << err.getDescription() << endl;
            }
      }
}

else {
      // XML-RPC was successful.  Now extract the data.
      // xmlrpc->isFinished() must be true in order to continue.
      assert(xmlrpc->isFinished());

      // Get the xmlrpc_c::value object that represents
```

```
      // the result returned by the server.
      xmlrpc_c::value result = xmlrpc->getResult();

      // Cast the result to an std::vector<xmlrpc_c::value>
      // if it is not a simple data type (like an integer).
      std::vector<xmlrpc_c::value> resultVector =
            xmlrpc_c::value_array(result).vectorValueValue();

      // Now it is possible to extract each element of the
      // vector, casting each element to its known type.
}
```

The next example demonstrates the execution of the same RPC, but in Java using the Apache XML-RPC library.

```
// Create the XML-RPC client.
private XmlRpcClient xmlrpcClient = null;

// Initialize the client.
try {
      // Set up the XML-RPC client to connect to the project
      // server.
      // Note the format, http://host:port/RPC2
      xmlrpcClient = new
            XmlRpcClient("http://localhost:2080/RPC2");
} catch (MalformedURLException e) {
      // The URL given to the client was invalid.
      e.printStackTrace();
      System.exit(1);
}

// Execute the RPC.
try {
      Object returnValue =
            xmlrpcClient.execute("server.getLastWorkUnit",
                  new Vector());
      Vector resultVector = null;

      // Get the return value.
      try {
            resultVector = (Vector)returnValue;

            // Now it is possible to extract each element of the
            // vector, casting each element to its known type.
      } catch (Exception e) {
            // The return value was not a Vector, it was an
            // Exception.
            // We can interpret the Exception and take appropriate
            // action.
      }
```

```
} catch (XmlRpcException e) {
      // Some XML-RPC problem occurred.
      // Ex. RPC not found on server.
      // Ex. Incorrect number of parameters passed to RPC.
      // Ex. Server threw an exception (not necessarily a
      // problem).

      System.exit(1);
} catch (IOException e) {
      // The host could not be found, or the connection
      // was refused.

      System.exit(1);
}
```

Creating an XML-RPC server is slightly more complicated, but it is necessary to do so in order to develop a science application. The following block of code is an example of the steps needed to create a C++ XML-RPC server with one RPC handler, called `sciapp.shutdown`. A more complete example can be found in the *example* directory of the framework.

```
// Thread in which the XML-RPC server will run.
pthread_t serverThread;

// Flag that controls whether the application should be running.
bool isRunning = true;

// BEGINNING OF XML-RPC HANDLERS
// ...

/**
 * Sets the isRunning flag to false so the loop in main will
 * terminate, causing the application to shutdown.
 * @return Returns true.
 */
class ShutdownMethod : public xmlrpc_c::method {
public:
      /**
       * This method is called when the sciapp.shutdown RPC is
       * handled.
       * @param paramList List of RPC parameters.
       * @param retvalP Pointer to the return value of the
       * method.
       */
      void execute(xmlrpc_c::paramList const& paramList,
            xmlrpc_c::value* const retvalP) {
            isRunning = false;
            *retvalP = xmlrpc_c::value_boolean(true);
      }
};
```

```
// ...
// END OF XML-RPC HANDLERS

/**
 * Starts the XML-RPC server in a separate thread.
 * @param arg Pointer to the xmlrpc_c::serverAbyss object to run.
 */
void* startServerThread(void* arg) {
      // Cast the void* argument to a pointer to an
      // xmlrpc::serverAbyss
      xmlrpc_c::serverAbyss* server =
            static_cast<xmlrpc_c::serverAbyss*>(arg);
      server->run();
      pthread_exit(EXIT_SUCCESS);
}

/**
 * Cleanly shuts down the application.
 */
void shutdown() {
      isRunning = false;
      pthread_cancel(serverThread);
      pthread_join(serverThread, NULL);
      exit(EXIT_SUCCESS);
}

int main(int argc, char** argv) {
      xmlrpc_c::registry xmlrpcRegistry;

      // Register the RPC handlers.
      xmlrpc_c::methodPtr const shutdownMethod(
            new ShutdownMethod);
      // "sciapp.shutdown" is the name of the RPC
      // shutdownMethod is the handler.
      xmlrpcRegistry.addMethod("sciapp.shutdown",
            shutdownMethod);
      // ...

      // Initialize the XML-RPC server.
      // The second parameter is the port number on which to
      // listen.
      // The third parameter is the path to the log file.
      xmlrpc_c::serverAbyss abyssServer(xmlrpcRegistry, 2082,
            "log/sciapp.xmlrpc.log");

      // Start the XML-RPC server thread.
      // Pass a pointer to the server as an argument to
      // startServerThread().
      pthread_create(&serverThread, NULL, startServerThread,
            &abyssServer);

      // XML-RPC server is in its own thread, so make the main
```

```
      // thread sleep until the application should terminate.
      while (isRunning == true) {
            sleep(5);
      }

      // Shut down the application.
      shutdown();
}
```

The next block is the Java implementation of the code necessary to start an XML-RPC server containing the `sciapp.shutdown` RPC handler.

```
/**
 * This is the class that will start the XML-RPC server.
 */
public class ExampleXMLRPCServer {
      // The XML-RPC server object.
      private WebServer xmlrpcServer = null;

      // Flag that controls whether the application should be
      // running.
      private boolean isRunning = false;

      /**
       * Initializes and starts the XML-RPC server.
       */
      private void initXmlRpcServer() {
            try {
                  // Instantiate the web server.
                  // The parameter is the port number on which to
                  // listen.
                  xmlrpcServer = new WebServer(2082);

                  // Register the RPC handler class.
                  // Any public methods in the handler will be
                  // treated as RPC handlers.  It is possible to
                  // use "this" as the RPC handler, but it is
                  // cleaner to use a separate class.
                  // We pass a reference to "this" to the handler
                  // class so that the handler can access our
                  // methods.  An alternative would be
                  // to make ExampleXMLRPCServer a singleton.
                  // We will call the handler "sciapp"
                  // All clients connecting to this server will use
                  // RPCs in the form: sciapp.rpcName
                  xmlrpcServer.addHandler("sciapp", new
                        ExampleXMLRPCHandler(this));
            } catch (Exception e) {
                  System.err.print("Error initializing the ");
                  System.err.println("XML-RPC server: " + e);
                  System.exit(1);
```

```
            }
        }

        /**
         * Starts the XML-RPC server and begins handling requests.
         */
        public void run() {
            if (isRunning == false) {
                // Note: the server starts in a new thread.
                xmlrpcServer.start();
                isRunning = true;
            }
        }

        /**
         * Performs any tasks necessary to cleanly shut down the
         * science application.
         * @return Returns true.
         */
        public boolean shutdown() {
            xmlrpcServer.shutdown();
            isRunning = false;
            return true;
        }

        public static void main(String[] args) {
            ExampleXMLRPCServer server =
                new ExampleXMLRPCServer();
            server.run();

            // The XML-RPC server is in another thread, so just
            // sleep until the application is shut down.
            while (sciApp.isRunning()) {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            // The application is shut down.
            System.exit(0);
        }
}

/**
 * This is the class that handles the RPCs.
 * Each method in this class is interpreted
 * as an RPC.
 */
public class ExampleSciAppRPCHandler {
    // Reference to the ExampleXMLRPCServer
```

```
	// that created this handler.
	private ExampleXMLRPCServer server = null;

	/**
	 * Constructor that takes a reference to the
	 * ExampleXMLRPCServer that instantiated this object.
	 * @param server The ExampleXMLRPCHandler that instantiated
	 * this RPC handler.
	 */
	public ExampleXMLRPCHandler(ExampleXMLRPCServer server) {
		this.server = server;
	}


	// BEGINNING OF XML-RPC HANDLERS
	// ...

	/**
	 * Performs any tasks necessary to clearnly shut down this
	 * application.
	 * @return Returns true.  Note: All XML-RPC handlers MUST
	 * return a value.
	 */
	public boolean shutdown() {
		return server.shutdown();
	}

	// ...
	// END OF XML-RPC HANDLERS
}
```

Server-Side Components

The two server-side components that can be developed by each project are the work unit generator and the result validator. Neither of these components needs to implement an XML-RPC server because they both periodically poll the project server, and only take action based on the result of each poll. The server will never need to contact them directly.

Work Unit Generator

The purpose of the work unit generator is to partition a project's science data into smaller work units to be computed by the science application, which runs on each volunteer's computer. There is no limit to the amount of data that can be stored in a work unit. Figure 24 shows an overview of the work unit generator's control flow.

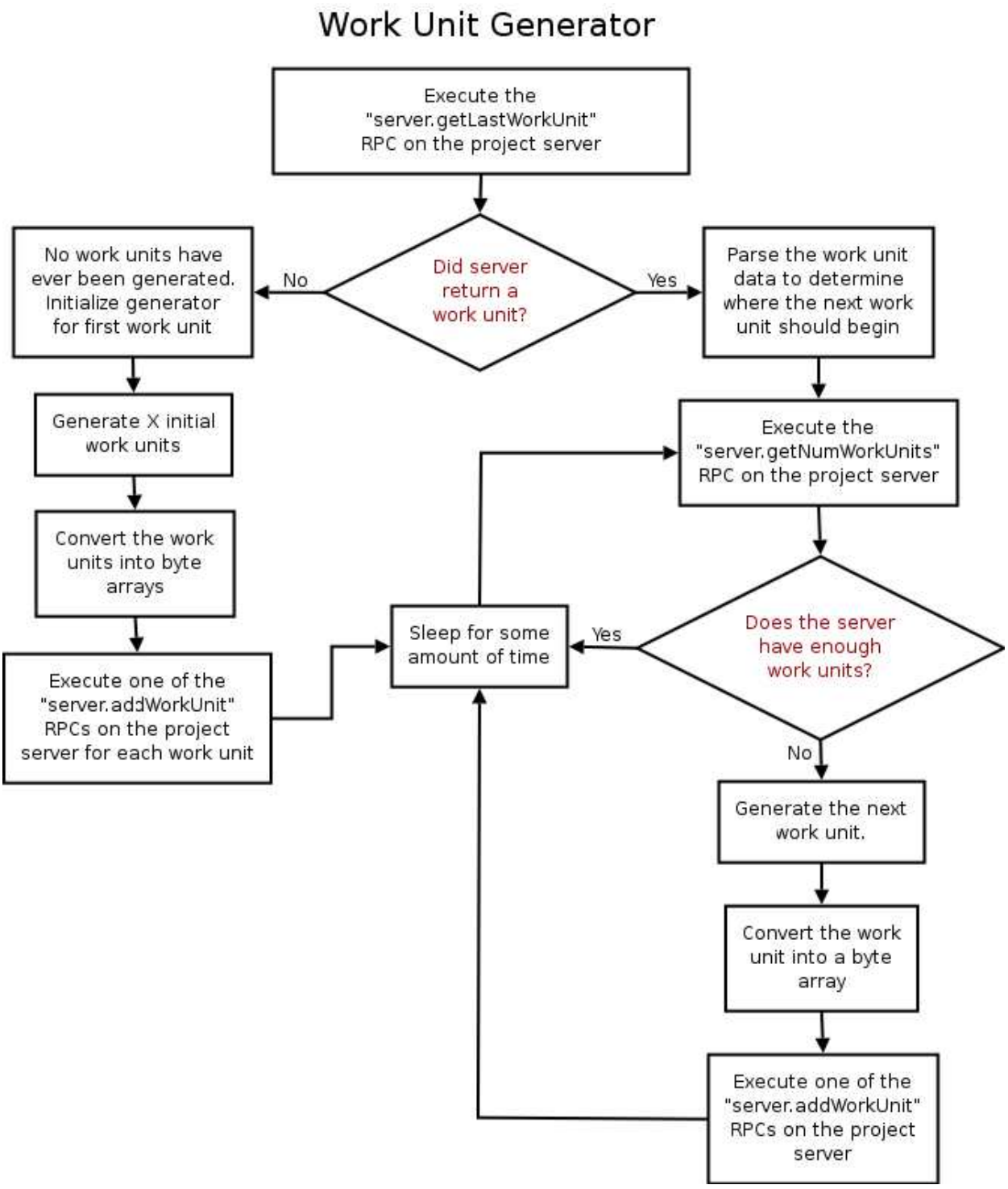

Figure 24: Work Unit Generator Control Flow

The first step taken by the work unit generator after it is started is to execute the `server.getLastWorkUnit` RPC on the project server. This RPC instructs the server to return all information about the last work unit that was generated, including its ID, data, creation date, point value, and priority. Complete specifications of this RPC and all other RPCs that project developers can use can be found in

Appendix D: XML-RPC Interface Specification. The reason that this RPC should be executed when the work unit generator is first started is that the generator will need to know at what position in the science data to resume partitioning. If the work unit generator is being started for the first time, there will not be any work units in the project database, so the project server should notify the generator, which would then start generating work units from the beginning of the science data. Thus, there are two possible return types for the `server.getLastWorkUnit` RPC: either it is a vector or some other object representing an exception. If the return value is not a vector, it does not matter what that other object is; it is only important to know whether the return value is a vector or not. If it is not a vector, then an exception has occurred. Depending on the XML-RPC library that is used, determining whether the return value is a vector can be done in two ways. Using the xmlrpc-c library, it is possible to query the status of the `xmlrpc_c::rpcPtr` object after the RPC has been executed. If the `isSuccessful()` method of the instance of that object returns true, it means that the RPC completed successfully, and a vector has been returned containing the relevant work unit information. If that method returns false, then the server threw an exception, meaning there were no work units in the project database. The process is simpler using the Apache library. The return value of the RPC will initially be a `java.lang.Object`. To determine whether that object is a vector, cast it to a `java.util.Vector` inside a `try` block. If a `java.lang.ClassCastException` was thrown as a result of the cast, then the object was an exception. If the cast was successful, then the server did not throw an exception, and it is now possible to access the individual elements in the vector.

If there were no work units in the database, the work unit generator should generate one or more work units to be sent to the project server. The first time the work unit generator is started, we recommend that several work units be generated so that there will be enough in the system to distribute to clients when they begin connecting. After generating these work units, the work unit generator should execute one of the `server.addWorkUnit` RPCs on the project server. There are several variants of this RPC, each accepting different combinations of parameters. These RPCs will be explained shortly.

If the server did return a work unit, the work unit generator should inspect that work unit to determine what part of the science data to partition next. In most cases, it will only be necessary to inspect the data contained in the work unit, which is at index 1 in the vector. When the work unit generator has determined what data will be assigned to the next work unit, there are two options for how to proceed. The first option is to query the server to determine the number of ingress work units in the database. An ingress work unit is a work unit that has never been sent to a volunteer to be processed. Although ingress work units are not the only type of work unit that can be sent to a client, there should always be ingress work units in the system to guarantee that whenever a project client requests a work unit there will be one available. The number of ingress work units can be queried by executing the `server.getNumWorkUnits` RPC. This RPC does not require any parameters, and it returns an integer greater than or equal to zero indicating the number of ingress work units in the project. The work unit generator should decide whether the number of ingress work units the project has is sufficient, and if not, it should generate some number of work units and send them to the project server

via one of the `server.addWorkUnit` RPCs. The second option the work unit generator has is to skip executing the `server.getNumWorkUnits` RPC, and instead generate and send a single work unit to the server by executing one of the `server.addWorkUnit` RPCs. These RPCs all return an integer indicating the number of clients that are waiting for work units. A client is said to be waiting for a work unit if it requested a work unit when the project server did not have any to distribute. The existence of clients that are in a waiting state reduces the efficiency of a project because instead of performing useful computations, these clients are instead in a sleeping state, waiting to request a work unit from the server at a later time. After sending the newly generated work unit to the project server, the work unit generator can decide whether the number of clients waiting for work units is acceptable or not, and it may choose to generate and send additional work units.

There are several variants of the `server.addWorkUnit` RPC to allow some degree of control over the work units, if desired, while providing a simple interface for projects that do not need the advanced control features. The version of this RPC that allows for the most control over the work units takes five parameters:

Parameter Index	Type	Description
0	String	The work unit ID to use. **Must be unique.**
1	Byte[]	The work unit data.
2	Integer	The priority of this work unit, where priority >= 0.
3	Integer	The point value of this work unit, where points >= 0.
4	Double	The number of seconds until the work unit should expire, where seconds >= 0.

Recall that clients executing this RPC must add each of these parameters to a vector in the order specified in the **Parameter Index** column, and then execute the RPC by passing that vector to the XML-RPC library. All XML-RPC parameters must be wrapped in a vector, even if only one parameter is required, so to be concise we will no longer mention adding parameters to a vector before executing an XML-RPC.

The first parameter allows specification of the work unit ID. This ID is displayed in the log files when the work unit is assigned and when results are returned for this work unit. When extracting the results from the database using the *extract_results* script, a directory is created for each work unit, and the name of each directory will be the work unit ID. If one of the `server.addWorkUnit` variants that does not require a work unit ID is executed, a work unit ID will be generated by the server. The second parameter is the byte array representing the data to be stored in that work unit. The third parameter is the priority of the work unit. A priority value of 0 indicates the lowest priority, and higher values indicate higher priorities. Work units with higher priorities are guaranteed to be distributed to clients before work units with lower priorities. Priorities are relative, so if two work units have the same priority, the work units will be distributed according to a first-in-first-out (FIFO) ordering. If one of the `server.addWorkUnit` variants that does not require a priority is executed, the default priority of 0 is used. The fourth parameter is the point value for the work unit. Some work units may require more computations than others, so volunteers can be rewarded differently depending on the difficulty of the work unit assigned to them. The

point value of a work unit is added to a volunteer's score after the volunteer returns a result for that work unit. If the result is later found to be invalid, the volunteer's score is decreased by the point value of the work unit. If one of the `server.addWorkUnit` variants that does not require a point value is executed, the default point value of 1 is used. The last parameter is the work unit's expiration time. This parameter is used to impose a limit on the amount of time a work unit has between being added to the project and being retired. The transitioner optimistically assumes that all clients will return results promptly. It will therefore only assign a work unit a certain number of times. That number is the minimum number of results that is required for each work unit, defined in the project configuration. However, if a client is assigned a work unit, and then never returns a result, that work unit might never be retired. The purpose of the expiration time is to prevent this from happening by limiting the amount of time a work unit can spend in the system. If a work unit is not retired before its expiration time, that work unit can be assigned to other volunteers. The expiration time is specified in seconds. Note that if the expiration time is shorter than the average time required for a client to complete a work unit, many work units will expire unnecessarily, resulting in wasted work because those work units will be assigned to additional volunteers. If one of the `server.addWorkUnit` variants that does not require an expiration time is executed, the default expiration time of one day is used.

Different projects may have different needs for the level of control over work units, so there are several variants of the `server.addWorkUnit` RPC, each allowing different combinations of parameters to be used. All of these variants are presented in

Appendix D: XML-RPC Interface Specification.

Result Validator

The result validator decides whether each individual result returned by a client is valid, decides whether a client has passed a spot-check, and selects a canonical result from the set of all results returned for a particular work unit. The canonical result is the result that is accepted as the final result for a work unit; all other results for that work unit are deleted. Implementing the result validator is optional because the framework provides a default validator. The default validator marks all results as valid, marks all spot-checks as passed, and selects the first result received as the canonical result. If a custom result validator is implemented, the project developers can also choose to use the default behavior for one or more functions of the validator. For example, if a project only needs the spot-checking functionality of the result validator, the validator can be made to always mark normal results valid and to select the first result returned as the canonical result. The use of a default behavior for a particular function will be referred to as *subbing* that function. If a custom result validator is implemented, all three functions must be implemented, but any number of them may be stubbed. Figure 25 shows an overview of the result validator's control flow.

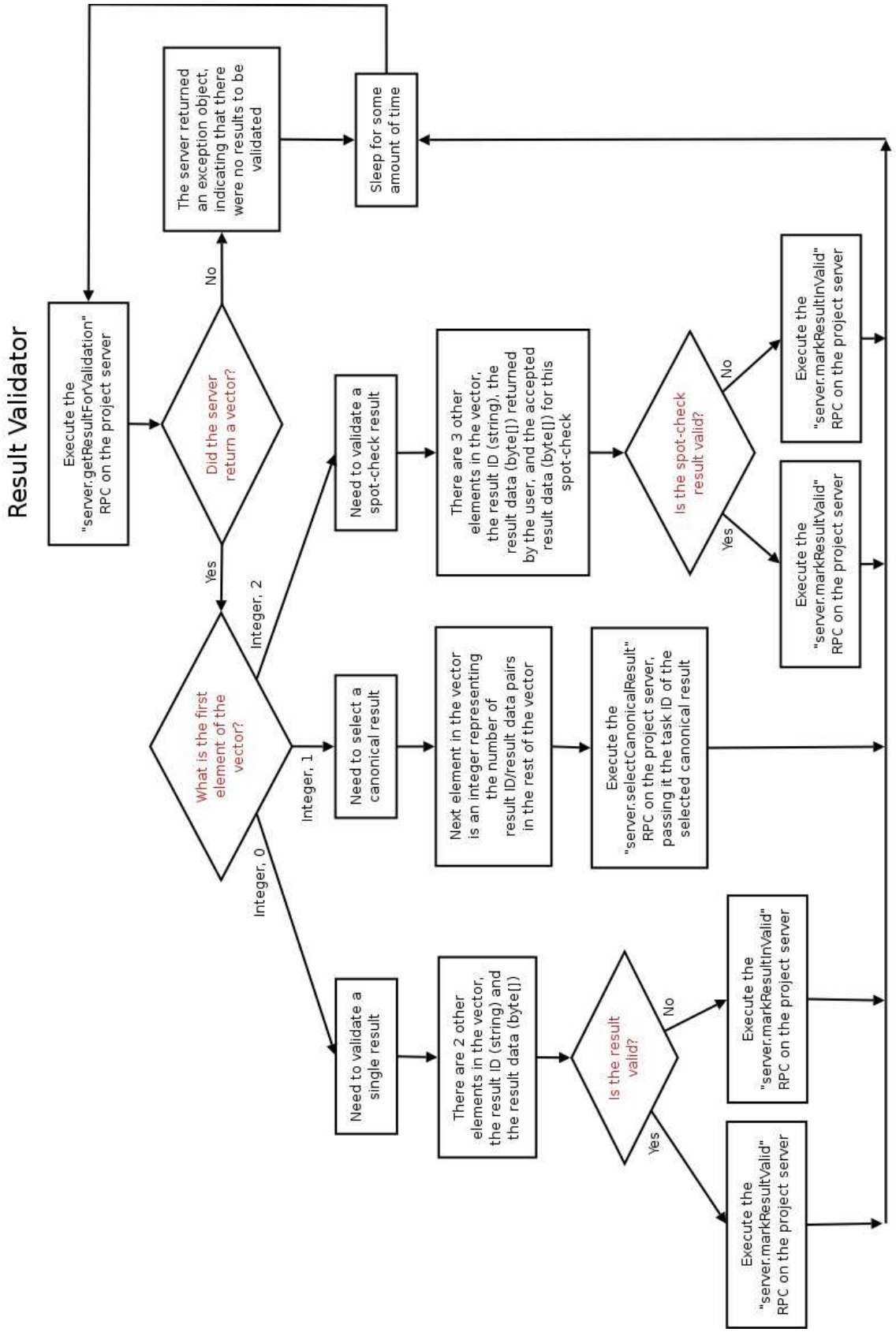

Figure 25: Result Validator Control Flow

The first step taken by the result validator is to execute the `server.getResultForValidation` RPC on the project server. This RPC does not take any parameters. This RPC will return an exception if there are no results that need to be validated. If this is the case, the validator should sleep for some amount of time and then execute the RPC again. If there is a result to validate, the server returns a vector in one of three formats. The first element (index 0) of the vector will always be an integer, which indicates the type validation that the validator needs to perform. A value of 0 indicates that the vector contains a single result to be validated. In this case, the vector will contain two other elements. The element at index 1 will be a string representing the ID of the result to be validated. The element at index 2 will be the data from that result. The result validator should decide whether the result returned by the project server is valid or not. If the result is valid, or this function is a stub, the result validator should execute the `server.markResultValid` RPC on the project server. Otherwise, the validator should execute the `server.markResultInvalid` RPC. Both of these RPCs take a single parameter, the ID of the result to mark valid or invalid. If the result was successfully marked valid or invalid, the return value of the RPC will be true. If the return value is false, it indicates that the given result ID could not be found.

If the first element of the vector is 1, it signifies that the validator needs to select a canonical result. In this case, the element at index 1 indicates the number of results that are contained in the vector. Each result is comprised of two separate elements in the vector. The first element is the result ID, and the second is the data for that result. For example, if the element at index 1 is the integer 2, it means that there are four more elements in the vector. The element at index 2 is the result ID of the first result, and the element at index 3 is the data for the first result. The element at index 4 is the result ID of the second result, and the element at index 5 is the data for the second result. The composition of the vector is presented as a table in

Appendix D: XML-RPC Interface Specification; the table format might be easier to understand. After all of the results have been examined, the validator should select one to be the canonical result. It should then call the `server.selectCanonicalResult` RPC on the project server. This RPC takes one parameter, the ID of the canonical result selected by the validator. It returns a boolean indicating whether the RPC was successful. A value of true indicates that the canonical result was successfully selected, and a value of false means that the given result ID could not be found.

If the first element of the vector is 2, the result that is being validated is a spot-check. There will be three other elements in the vector. The element at index 1 will be the ID of the result. The element at index 2 will be the spot-check result data returned by the client. The element at index 3 will be the accepted spot-check result data that was computed by the server. The validator should compare these two results to determine whether the result submitted by the client is close enough to the accepted result. If the validator decides that the client passed the spot-check, it should execute the the `server.markResultValid` RPC, passing the result ID as the parameter. If the client did not pass the spot-check, the validator should instead execute the `server.markResultInvalid` RPC.

After each result validation, the validator should sleep for some amount of time, and then execute the `server.getResultForValidation` RPC again.

Science Application

The purpose of the science application is to execute a project-specific algorithm on a work unit in order to produce result data that can later be analyzed by the project. The science application is unique because it can be used in two different ways. The primary use of the science application is to execute a project's science algorithm on each volunteer's computer. The project client acts as a proxy between the project server and the science application. The project client requests work units from the project server, starts the science application, and sends it the work unit. The client then waits to receive the result, which it returns to the project server. The client instructs the science application to compute the result for a work unit by executing an XML-RPC on the science application. For this reason, the science application must implement an XML-RPC server. The work unit generator and result validator do not need an XML-RPC server because they regularly poll the project server instead of being contacted by the server. This polling method is not possible with the science application for two reasons. In the event that the science application crashed while computing the result for a work unit, the project client would not be able to detect the crash if it were simply waiting for the science application to send a result. However, if the project client initiated a connection with the science application by executing a synchronous XML-RPC, the project client would be notified by a Java Exception that the connection was reset, indicating a problem with the science application. The client could then restart the science application. The second reason why polling was not the best choice is related to the second use for the science application, which is to compute the accepted spot-check results on the computer where the project server is running. If the project uses spot-checks, the project server will start the science application and instruct it to compute the results for one or more work units, which will then become the spot-check work units and

results. Figure 26 shows an overview of the science application's control flow for both of its uses.

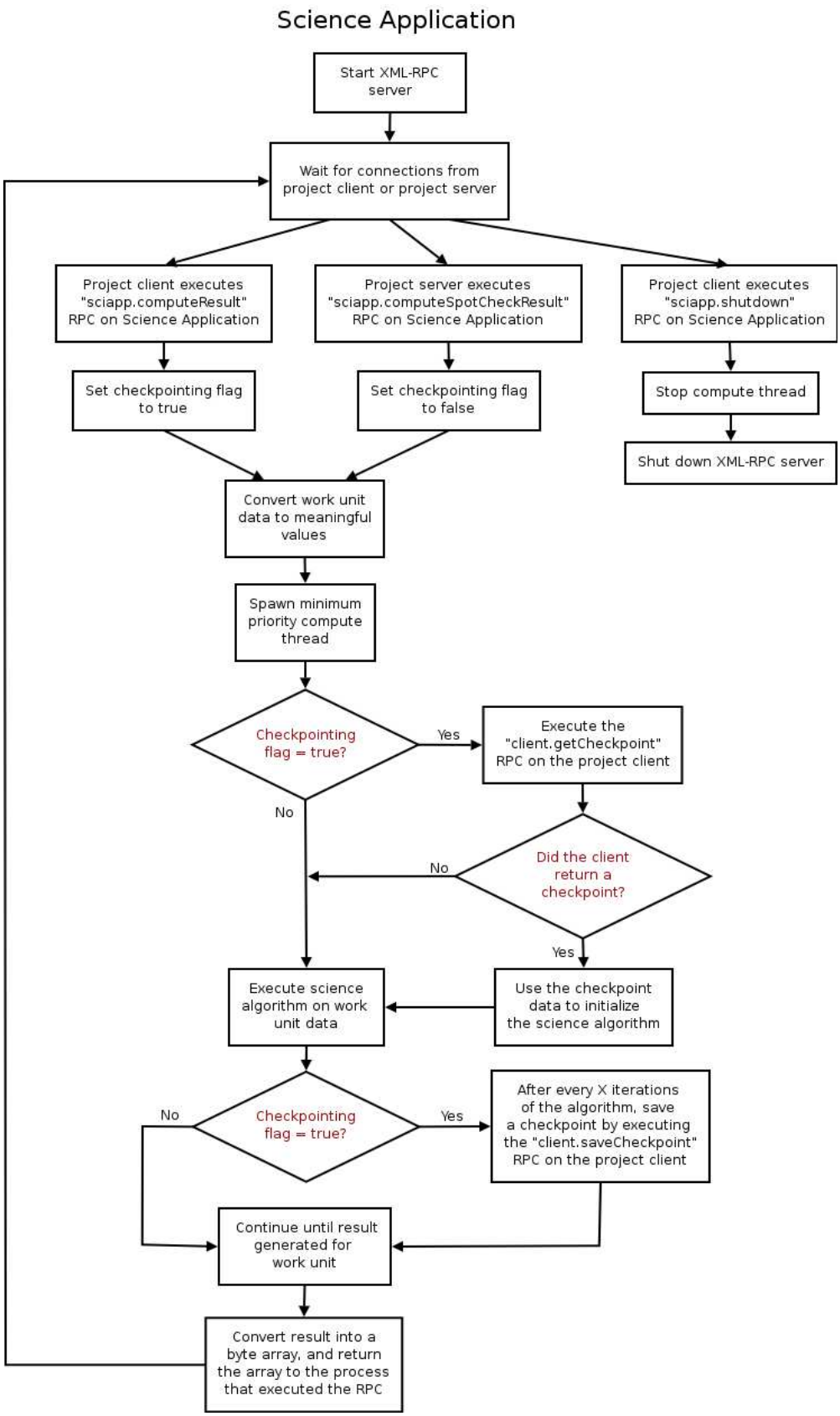

Figure 26: Science Application Control Flow

When the science application is started, it should immediately initialize its XML-RPC server and begin listening for XML-RPCs. Example code for creating an XML-RPC server can be found at the beginning of this appendix and also in the *example* directory of the latest framework release. There are three RPC handlers that the science application must implement. Before explaining the implementation of these RPCs, it is necessary to discuss an important feature of SLINC: check-pointing. If the science application is shut down before completing a computation and returning a result, the computation would normally have to be started from the beginning the next time the science application was started. This process is inefficient if more than a few minutes are required to compute the result for a work unit. Check-pointing addresses this problem by allowing the science application to periodically save its state and to retrieve this state information the next time it is started. This state information is referred to as a *check-point*. By periodically saving check-points, the science application can resume a previously started computation by retrieving the last check-point and initializing the science algorithm to begin at the appropriate point in the work unit data. Check-points could be saved by writing a file to disk and retrieved by reading that file, but the project client provides an RPC that performs those functions so that the science application does not need to perform disk I/O itself.

To save a check-point, the science application should execute the `client.saveCheckpoint` RPC on the project client, which always listens for connections on the localhost, or 127.0.0.1, address. This RPC takes a single byte array as its parameter, which is the data to be saved in the check-point. To retrieve a previously saved check-point, the science application can execute the `client.getCheckpoint` RPC on the project client, which does not take any parameters. This RPC returns a vector containing a byte array. If the byte array has length 0, the project-client did not find a previously saved check-point. Otherwise, the array contains the data from the last check-point that was saved.

There are two RPCs for computing results that the science application must implement. One of them is called `sciapp.computeResult`, and is executed by the project client when it has a new work unit for which a result must be computed. The other is called `sciapp.computeSpotCheckResult`, which is executed by the project server to compute spot-check results, if spot-checking is used by the project. These two RPCs perform exactly the same function; the reason for having the project client and project server call separate RPCs is that the check-pointing functionality is only present in the project client, not in the project server. When the `sciapp.computeSpotCheckResult` RPC is executed, the science application should therefore disable spot-checking, if it is used.

When one of the `computeResult` RPCs is executed, the science application should convert the given byte array into useful values that can be passed to the science algorithm. It should then spawn a new thread in which to execute the science algorithm; we will refer to this thread as the *compute thread*. This thread should have the lowest possible priority so that the science algorithm will only use processor time when there are no higher priority threads that require it. Using a higher priority thread may disrupt the volunteers' use of their own computers, which may reduce volunteer participation. Before executing the science algorithm, the compute thread should first execute the `client.getCheckpoint` RPC on the project client to determine whether the science

algorithm should resume a previously started computation. If the project client returns a check-point, the science algorithm should be initialized to resume the previous computation. Otherwise, the science application should be executed, starting at the beginning of the work unit. During its execution, the science algorithm may periodically call the `client.saveCheckpoint` RPC on the project client, passing it a byte array containing state information so that the computation can be resumed if the science application were shut down before the completion of the current computation. When the science algorithm has finished computing the result for its assigned work unit, the compute thread should terminate. The science application should then encode the result into a byte array, wrap it in a vector, and send it to the project client via the return statement of the `computeResult` RPC that was executed.

It is very important to write a script called *run_sci_app* that will start the science application. This script is executed by the project client to start the science application. If the science application is designed to run on Windows platforms, this script should be called *run_sci_app.bat*; if it is designed to run on UNIX-like platforms, the script should be called *run_sci_app.sh*. If it is designed to run on both platforms, both scripts will be needed. The science application detects what platform it is running on and executes the appropriate script for that platform. If the *run_sci_app.sh* script is used, it is very important that all users have read and execute permissions on the script, for example 755. Inside each subdirectory of the *example* directory in the framework there are example *run_sci_app* scripts for both Windows and UNIX.

Appendix D: XML-RPC Interface Specification

This appendix contains information about all XML-RPCs that can be executed by project components or which project components must implement. For each RPC there is a description of its function, a description of each parameter it accepts, information about any exceptions it throws, and the specification of its return value.

XML-RPCs Available to Project Components

`server.getLastWorkUnit()`
Description: Used by the work unit generator, this RPC returns information about the last work unit that was generated. This information is useful for determining where in the science data to resume partitioning after the work unit generator has been restarted. If no work units have ever been added to the project, this call throws an exception.
Parameters: 0
Throws Exception: If the project has zero work units.
Returns: vector

Vector Index	Type	Description
0	String	The work unit ID.
1	byte[]	The work unit data.
2	Date	The creation date of the work unit.
3	Integer	The point value of the work unit.
4	Integer	The priority of the work unit.

`server.getNumWorkUnits()`
Description: Returns the number of ingress work units in the project database/transitioner.
Parameters: 0
Throws Exception: Never.
Returns: integer >= 0, the number of users waiting for work units

`server.addWorkUnit(String, byte[], int, int, double)`
Description: Adds a work unit to the database/transitioner, and returns the number of clients that are blocked, waiting for a work unit.
Parameters: 5

Vector Index	Type	Description
0	String	The work unit ID to use. **Must be unique.**
1	byte[]	The work unit data.
2	Integer	The priority of this work unit, where priority >= 0.
3	Integer	The point value of this work unit, where points >= 0.
4	Double	The number of seconds until the work unit should expire, where seconds >= 0.

Throws Exception: If the given work unit ID is already in use by another work unit or result.
Returns: integer >= 0, the number of users waiting for work units

`server.addWorkUnit(String, byte[], int, int)`
Description: Adds a work unit to the database/transitioner, and returns the number of clients that are blocked, waiting for a work unit.
Parameters: 4

Vector Index	Type	Description
0	String	The work unit ID to use. **Must be unique.**
1	byte[]	The work unit data.
2	Integer	The priority of this work unit, where priority >= 0.
3	Integer	The point value of this work unit, where points >= 0.

Throws Exception: If the given work unit ID is already in use by another work unit or result.
Returns: integer >= 0, the number of users waiting for work units

`server.addWorkUnit(byte[], int, int, double)`
Description: Adds a work unit to the database/transitioner, and returns the number of clients that are blocked, waiting for a work unit.
Parameters: 4

Vector Index	Type	Description
0	byte[]	The work unit data.
1	Integer	The priority of this work unit, where priority >= 0.
2	Integer	The point value of this work unit, where points >= 0.
3	Double	The number of seconds until the work unit should expire, where seconds >= 0.

Throws Exception: Never.
Returns: integer >= 0, the number of users waiting for work units

`server.addWorkUnit(byte[], int, int)`

Description: Adds a work unit to the database/transitioner, and returns the number of clients that are blocked, waiting for a work unit.

Parameters: 3

Vector Index	Type	Description
0	byte[]	The work unit data.
1	Integer	The priority of this work unit, where priority >= 0.
2	Integer	The point value of this work unit, where points >= 0.

Throws Exception: Never.

Returns: integer >= 0, the number of users waiting for work units

`server.addWorkUnit(byte[])`

Description: Adds a work unit to the database/transitioner, and returns the number of clients that are blocked, waiting for a work unit.

Parameters: 1

Vector Index	Type	Description
0	byte[]	The work unit data.

Throws Exception: Never.

Returns: integer >= 0, the number of users waiting for work units

`server.getResultForValidation()`
Description:
Parameters: 0
Throws Exception: If there are no results to be validated.
Returns: vector (3 possibilities)

Return Type 0: Vector containing a single result to be validated.

Vector Index	Type	Description
0	Integer	Enumeration indicating the format of the rest of the vector. For return type 0, this value is 0.
1	String	The result ID.
2	byte[]	The result data.

Return Type 1: Vector containing several results from which a canonical result must be selected.

Vector Index	Type	Description
0	Integer	Enumeration indicating the format of the rest of the vector. For return type 1, this value is 1.
1	Integer	The number of results in this vector, n.
2	String	The ID of result 0.
3	byte[]	The data from result 0.
4	String	The ID of result 1.
5	byte[]	The data from result 1.
…	…	…
2n + 1	String	The ID of result n, where n = the value of vector index 1.
2n + 2	byte[]	The data from result n.

Note: There are 2n+3 elements in the entire array.

Return Type 2: Vector containing a spot-check result to be validated.

Vector Index	Type	Description
0	Integer	Enumeration indicating the format of the rest of the vector. For return type 2, this value is 2.
1	String	The ID of the result returned by the client.
2	byte[]	The data from the result returned by the client.
3	byte[]	The data from the accepted spot-check result.

`server.getAssociatedWorkUnit()`
Description: This is an optional RPC that can be used by the result validator. The **`server.getResultForValidation()`** RPC only returns the result data, so if the validator needs to examine the work unit data from which that result was computed, it can use the **`server.getAssociatedWorkUnit()`** RPC.
Parameters: 1

Vector Index	Type	Description
0	String	The ID of the result for which to find the associated work unit data.

Throws Exception: If no result with the given ID was found.
Returns: vector

Vector Index	Type	Description
0	Byte[]	The work unit data associated with the given result.

`server.markResultValid(String)`
Description: Marks a result as valid.
Parameters: 1

Vector Index	Type	Description
0	String	The ID of the result to mark valid.

Throws Exception: Never.
Returns: boolean: true if the result was successfully marked valid, false if the given result ID was not found.

`server.markResultInvalid(String)`
Description: Marks a result as invalid.
Parameters: 1

Vector Index	Type	Description
0	String	The ID of the result to mark invalid.

Throws Exception: Never.
Returns: boolean: true if the result was successfully marked invalid, false if the given result ID was not found.

`server.selectCanonicalResult(String)`
Description: Selects a result to be the canonical result for a work unit.
Parameters: 1

Vector Index	Type	Description
0	String	The ID of the canonical result.

Throws Exception: Never.
Returns: boolean: true if the result was successfully selected to be the canonical result, false if the given result ID was not found.

`client.saveCheckpoint(byte[])`
Description: Saves a work unit to disk for later retrieval using the `client.getCheckpoint` RPC. Only one check-point is saved, so subsequent executions of the RPC will overwrite the previous check-point.
Parameters: 1

Vector Index	Type	Description
0	byte[]	The check-point data.

Throws Exception: Never.
Returns: boolean: true if the check-point was successfully saved to disk, false if any error occurred, such as insufficient permissions to create the file.

`client.getCheckpoint()`
Description: Saves a work unit to disk for later retrieval using the `client.getCheckpoint` RPC.
Parameters: 0
Throws Exception: Never.
Returns: vector

Vector Index	Type	Description
0	byte[]	The last check-point data that was saved. If no check-point was found, the array will have length = 0. Otherwise, it will contain the saved check-point data.

XML-RPC Handlers to be Implemented by the Science Application

`sciapp.computeResult(byte[])`
Description: Computes the result for the given work unit.
Parameters: 1, byte[]: the work unit data
Throws Exception: Never.
Returns: vector

Vector Index	Type	Description
0	byte[]	The result data.

`sciapp.computeSpotCheckResult(byte[])`
Description: Computes the spot-check result for the given spot-check work unit.
Parameters: 1, byte[]: the work unit data.
Throws Exception: Never.
Returns: vector

Vector Index	Type	Description
0	byte[]	The result data.

`sciapp.shutdown()`
Description: Shuts down the science application.
Parameters: 0
Throws Exception: Never.
Returns: boolean: always returns true because all RPCs must have a return value.

Appendix E: Example Project

We created an example project for our public resource computing framework primarily for two reasons. The first reason was that we needed an actual project to perform our functional testing. The example project was useful throughout the development process for finding faults and determining whether our XML-RPC based IPC was working correctly. The second reason we created an example project was to provide future users of SLINC with a working example to use as a reference as they developed their own projects.

The problem our example project solves is finding prime numbers. Searching for prime numbers was one of the original applications for public resource computing. In 1995 a project called the Great Internet Mersenne Prime Search (GIMPS)[35] was founded. The purpose of GIMPS was to find previously undiscovered Mersenne primes. Our example project is less ambitious. It simply finds all prime numbers within an assigned range.

The example project is comprised of three components: the work unit generator, the science application, and the result validator. The work unit generator generates fixed-length ranges of integers that should be searched for prime numbers. Each work unit contains a range of one million integers. The first work unit contains the range [2, 1,000,002], the second contains the range [1,000,003, 2,000,004], and so on.

The science application is started by the project client, which runs on each volunteer's computer. The science application is responsible for executing the prime finding algorithm on the ranges contained in each work unit. The algorithm it uses to find prime numbers is fairly simple:

```
FindPrimes(startRange, endRange) {
     foundPrimes ← ∅
     for (currentTest ← startRange to endRange) do {
          isPrime ← true
          for (divisor ← 2 to √currentTest ) do {
               if (currentTest % divisor = 0) then {
                    isPrime ← false
                    exit innermost loop
               }
          }
          if (isPrime = true) then {
               foundPrimes ← foundPrimes ∪ currentTest
          }
     }

     return foundPrimes
}
```

Figure 27: Prime Finding Algorithm

This algorithm tests a number for primality by dividing it by every number between two and the square root of the number being tested. If the remainder of any of those division operations was zero, then the number being tested was not prime. Note that the operator used for modulo division in Figure 27 is `%`, the modulo operator from the C programming language. The number of division operations required to test whether a number is prime grows with the square root of the number being tested. Therefore, the time complexity of this algorithm expressed in big-O notation is $O(\sqrt{n})$. Since the work units generated by the work unit generator all contain ranges of the same length, as more work units are generated the number of operations required to compute their results will grow. However, $O(\sqrt{n})$ growth is fairly slow, so even using the primitive algorithm from Figure 27, distributed computing is an effective method for finding prime numbers.

The result validator for the example project is very simple. It simply marks all results as valid and selects the first result that was returned for a work unit as the canonical result for that work unit. The reason our validator is so simple is that we did not need any validation during the testing of SLINC. For the usability tests, it sufficed to show that the XML-RPC interface that the validator used was working correctly. We were in complete control of the performance tests, which were running on our private cluster, so we did not need to validate any of the work units. For demonstration purposes, this simple example validator shows enough about the structure of the validator to be useful as a reference for those developing validators.

References

[1] Foster, Ian, C. Kesselman, S. Tuecke. "What is the Grid? A Three Point Checklist," *Grid Today*, (2002).
[2] "Berkeley Open Infrastructure for Network Computing," http://boinc.berkeley.edu, (accessed September 5, 2005).
[3] Anderson, David P., Eric Korpela, and Rom Walton. "High-Performance Task Distribution for Volunteer Computing," *Proceedings of the First IEEE International Conference on e-Science and Grid Technologies*, (December 5-8, 2005, Melbourne, Australia).
[4] Sarmenta, Louis F. G. "Volunteer Computing," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, (June 2001).
[5] "SETI@home," http://setiathome.berkeley.edu, University of California, (accessed September 9, 2005).
[6] "distributed.net," http://distributed.net, (accessed September 9, 2005).
[7] "Applets," http://java.sun.com/applets, Sun Developer Network, (accessed October 8, 2005).
[8] "Applet Security FAQ," http://java.sun.com/sfaq/, Sun Developer Network, (accessed September 9, 2005).
[9] Sarmenta, Louis F.G. "Studying Sabotage-Tolerance Mechanisms through Web-based Parallel Parametric Analysis and Monte Carlo Simulation," *2nd International Conference on Internet Computing, in conjunction with PDPTA 2001*, (Las Vegas, Nevada, June 25-28, 2001).
[10] Jamie Carlson, David Esposito, and Nate Springer. "Java Distriblets." Technical Report MQP-DXF-9802, Worcester Polytechnic Institute, Spring 1999.
[11] Li La Moon, Yue Shen, and Keiji Oenoki. "Distriblet Applications." Technical Report MQP-CEW-9901,Worcester Polytechnic Institute, Spring 2000.
[12] Gabriel R. Boys, Michael DiCicco, and Thomas A. Plunkett. "Distriblet IV." Technical Report MQP-DXF-0001, Worcester Polytechnic Institute, Spring 2001.
[13] Anderson, David P. "BOINC: A System for Public-Resource Computing and Storage," *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, (November 8, 2004, Pittsburgh, USA).
[14] Buck, Paul D. "Unofficial BOINC Wiki: Overview of Daemons," http://boinc-doc.net/boinc-wiki/index.php?title=Overview_of_Daemons, (accessed October 9, 2005).
[15] "MySQL AB: The World's Most Popular Open Source Database," http://www.mysql.com, MySQL AB, (accessed October 8, 2005).
[16] "XML-RPC Home Page," http://www.xmlrpc.org, UserLand Software, (accessed October 8, 2005).
[17] "World Wide Web Consortium," http://www.w3c.org, W3C, (accessed January 7, 2006).
[18] "HSQLDB," http://www.hsqldb.org, The hsqldb Development Group, (accessed January 7, 2005).
[19] "PostgreSQL: The world's most advanced open source database," http://www.postgresql.org, PostgreSQL Global Development Group, (accessed January 7, 2006).
[20] "Oracle Corporation," http://www.oracle.com, Oracle, (accessed January 8, 2006).
[21] "Microsoft SQL Server Home," http://www.microsoft.com/sql, Microsoft Corporation, (accessed January 8, 2005).
[22] "SourceForge.net: Welcome to SourceForge.net," http://www.sourceforge.net, Open Source Technology Group (OSTG), (accessed January 14, 2006).
[23] "Eclipse.org home," http://www.eclipse.org, The Eclipse Foundation, (accessed January 14, 2006).
[24] "JUnit, Testing Resources for Extreme Programming," http://www.junit.org, Object Mentor, Incorporated, (accessed January 14, 2006).
[25] "abeille: Abeille Forms Designer," https://abeille.dev.java.net, (accessed January 19, 2006).
[26] "JGoodies: Java User Interface Design", http://www.jgoodies.com, JGoodies, (accessed January 19, 2006).
[27] "Apache Ant – Welcome," http://ant.apache.org, The Apache Software Foundation, (accessed January 21, 2006).
[28] "Apache XML-RPC," http://ws.apache.org/xmlrpc, The Apache Software Foundation, (accessed October 8, 2005).
[29] "XML-RPC for C and C++: Overview," http://xmlrpc-c.sourceforge.net, Eric Kidd, (accessed October 8, 2005).
[30] "hibernate.org – Hibernate," http://hibernate.org, JBoss, Inc, (accessed October 22, 2005).

[31] Grecu, Dan L. and Lee A. Becker. "Coactive Learning for Distributed Data." Computer Science Department, Worcester Polytechnic Institute. Worcester, MA, 1998.
[32] Sarmenta, Luis F. G. "Studying Sabotage-Tolerance Mechanisms through Web-based Parallel Parametric Analysis and Monte Carlo Simulation." Massachusetts Institute of Technology. Cambridge, MA, 2001.
[33] "OSCAR: Open Source Cluster Application Resources," http://oscar.openclustergroup.org, Open Cluster Group, (accessed February 3, 2006).
[34] Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne. "Operating System Concepts." 6th ed. Hoboken, NJ: John Wiley & Sons, Inc., 2003.
[35] "Mersenne Prime Search," http://www.mersenne.org, (accessed February 25, 2006).