Cross-Platform Development using the Software Matrix Model

BY

Vijay Appadurai

Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer and Information Sciences

ADVISOR:

Dr. James Fawcett

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

SYRACUSE UNIVERSITY

MARCH 2007
Syracuse, New York

Abstract:

In This Thesis, we develop an architecture for a repository server model based on the Software Matrix [1], to support development of software that is intended to run on multiple platforms. The Software Matrix is a framework that supports and promotes the reuse of components. In particular it is a runtime infrastructure into which individual pieces of applications can plug. A repository server is one which supports the storage and retrieval of files such as production code and documentation which are used as part of the software development lifecycle. The repository server model is intended to be used to manage the software resources of a large corporation, engaged in building cross platform systems, using collaboration of remotely distributed development teams. The system architecture also provides support for teams engaged in test-driven development.

Many specialized server models have been developed previously. The model to be developed in this research is unique in several respects. First, its structure is based on a comprehensive component dependency model. We intend to show that this repository server model has some compelling advantages compared to other recently developed models.

Table of Contents

1. Chapter 1 RESEARCH GOALS AND RELATED WORK

1.1 Introduction……………………………………………………....7

1.2 The Problem……………………………………………………....7

1.3 Prior Approaches………………………………………………9

1.3.1 Boost Build System…………………………………………..9

1.3.2 GNU Build System……………………………………………10

1.3.3 Intermediate Code Generation……………………………11

1.3.4 Web Services Portability…………………………………...11

1.4 Our Approach ………………………………………………...12

2. Chapter 2 SOFTWARE MATRIX TECHNOLOGIES

Tools and Architectures Used...…………………………… 14

2.1.1 Software Matrix…...……………………….………….. .14

2.1.1.1 Matrix Cell….……………………………………………...16

2.1.1.2 Cell ID……………………………………………………....17

2.1.1.3 Capability List……………………………………………...17

2.1.1.4 Message Queues…………………………………………….17

2.1.2 Functionality………………………………………………..17

2.2 Design……………………………………………………….18

2.2.1 Loader……………………………………………………….19

2.2.2 Matrix……………………………………………………….20

2.2.3 ICell………………………………………………………….20

2.2.4 Serialization/Deserialization…………………………………23

2.3 Extensions to the Software Matrix…………………………...24

2.3.1 Improving Efficiency…………………………………………24

3. Chapter 3 REPOSITORY TESTBED SYSTEM

Repository Testbed System ……………………………………25

Network Structure…………………………………………………28

Repository Testbed System with the Software Matrix……..30

Repository Server…………………………………………………32

Definitions………………………………………………………..32

Version……………………………………………………...…32

Item……………………………………………………….....…32

XML Manifest……………………………………………...33

Component…………………………………………………………33

Check-in…………………………………………………………33

Check-out……………………………………………………..34

Extraction…………………………………………………….34

Browsing…………………………………………………...34

3.4.2 Component Representation…………………………………35

3.4.3 Version Control…………………………………………….…37

3.4.4 File System Layout……………………………………….....42

3.4.5 XML Metadata of an item in the Repository………....….43

3.4.6 Repository Partitions………………………………...…44

3.4.6.1 RepositoryExec……………………………………….…45

3.4.6.2 Version Control……………………………………………46

3.4.6.3 SuperComponent Manager………………………………47

3.4.7 Check-In………………………………………………………48

3.4.8 Check-Out…………………………………………………….51

3.4.9 Insert………………………………………………………..53

3.4.10 Extract…………………………………………………………55

4. Chapter 4 TESTBED SERVER

TestBed Server ……….…………………………………..56

4.1.1 Builder……………………………………………….58

4.1.1.1 Builder…………………………………………......59

4.1.1.2 Build Parameters…………………………………….59

4.1.2 Test Harness……………………………………..…60

5. Chapter 5 SUPPORT FOR CROSS-PLATFORM BUILDS USING THE SOFTWARE MATRIX

Repository and Cross-Platform Development…………….62

Name and Platform…………………………………….64

Source Files and Documentation Files……………………..64

References…………………………………………………64

Description and Keywords………………………………….65

Output Type, Compiler and Linker Options………………65

5.2 Sample Check-in using Repository and RTClient…………65

5.2.1 Check-in of FileIO item……………………………………...66

5.2.2 Check-in of xmlParser item…………………………………70

5.3 Setting up Testbed Server…………………………………...72

5.3.1 Role of Software Matrix…………………………………….72

6. Chapter 6 DEMONSTRATION – Design, Implementation and Results

Process and TCP Connection Explorer……………………73

Sniffer Server………………………………………………74

Process Sniffer……………………………………………75

Port Sniffer……………………………………………………75

Threads and Locks…………………………………………75

Windows Build of Sniffer Server…………………………76

6.2.1 Sniffer Application on Windows……………………….....78

Linux Build of Sniffer Server…………………………..…79

6.3.1 Sniffer Application on Linux…………………………..… 80

7. Chapter 7 CONCLUSION

Addressing Cross-Platform Development……………….81

Contributions of this Thesis………………………………81

Future Work…………………………………………….…83

7.3.1 Distributed Software Matrix……………………………...83

7.3.1.1 Disadvantages……………………………………………...83

7.4 Peer-to-Peer Software Matrix…………………………….84

7.5 Service Discovery…………………………………………….85

7.5.1 Fault Tolerance……………………………………………85

7.6 Load Balancing……………………………………………86

Chapter 1: Research Goals and Related Work

1.1 Introduction:

Cross platform [16] software consists of applications which work on multiple system platforms (e.g. Linux/Unix [17], Mac OS and Windows [18]). This may mean supporting all common platforms or simply more than one. Multi-platform software is important because of its ability to run everywhere and it can be easily tested and proposed as a standard [2]. Examples of multiplatform software include game development, standard libraries, and developer toolkits. A traditional approach to multiplatform software development follows the approach of serial development [3]. The application is initially developed on one platform of interest and ported to other platforms.

1.2 Problem:

"Develop and test the C++ [14] application on one platform and it will work as expected on other platforms without any change". [Unfortunately this is not the case with the software industry which works in an environment with many operating systems, processors and tools] Environment and interface differences between operating systems, compilers, and code management tools make this difficult.[3] Risks with cross-platform development include portability problems such as the differences in memory layout, dependencies on non-portable libraries and tools, performance, hardware differences and internationalization. There is a need for cross platform development because of the existence of more than one platform the application has to run on, e.g. popularity of Windows, Linux, Mac OSX, as well as Unix and a variety of real-time and near real-time

operating systems for embedded development. The application has to be built, tested and deployed for all the platforms of interest.

In this research, our goal is to develop a framework that supports development of applications which are intended to run on various platforms. We demonstrate how meta-data stored in a code repository is useful in providing a single code base, regardless of the number of the target platforms in which the software under source code control is to be tested and deployed. We intend to show how the Software Matrix can be used for development of such frameworks providing seamless integration of systems running on disparate platforms. We also show how a simple XML [13] based messaging using Software Matrix between heterogeneous systems is superior to Remote Procedure Calls (which are bound to the stack frame implementation of a particular platform) in the development of distributed [15] cross platform applications.

1.3 Prior Approaches:

1.3.1 Boost Build System:

This is one of the Open Source Build systems available which can be used to compile code in multiple platforms. Boost Build is a make replacement with a simple and high-level target language. It supports build variants, and several different compilers and tools. It is also possible to add new tools easily with the help of configuration files [4].

The multi-platform build process in a Boost Build System works as follows. The user specifies the toolsets which should be used to compile the application using a configuration file. The targets to be built are specified using jam files [5]. The most important thing to note is that in Boost.Build, unlike other build tools, the targets you declare do not correspond to specific files. What you declare in a Jam file is more like a "metatarget." Depending on the properties you specify on the command line, each metatarget will produce a set of real targets corresponding to the requested properties. It is quite possible that the same metatarget is built several times with different properties, producing different files.

So the user specifies all the build variants and the target platforms in which the code could be executed. The system chooses the variant and the target platform depending on the command line input given to it. So the tool could be installed on all platforms of interest and then the command line can be modified based on the platform in which it is supposed to be built and deployed.

However, it is independent from the code repository and so the source code has to be manually installed in each target platform. Any change in the source code requires an update in all platforms of interest. Also there is no build manager which keeps track of the builds in the various platforms. We intend to show that our approach facilitates easy tracking of the builds and their history using a unified user interface for all the platforms.

1.3.2 GNU Build System:

The GNU build system is a suite of tools produced by the GNU project that assist in making packages portable to many UNIX-like systems. It is part of GNU tool chain. It comprises GNU AutoConf, GNU Automake and GNU Libtool. [6]

Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of UNIX-like systems. AutoConf transforms a user written configuration file to a portable shell script which can be used in all platforms of interest.

GNU Automake is a programming tool that produces portable make files for use by the make program, used in compiling software. GNU Libtool is a software tool for creating portable shared libraries across all UNIX variants.

The GNU Build System follows an approach of “write first, configure for everywhere”. However it is useful only for UNIX variants and so it is not truly cross platform. Our

system is designed to support applications which are intended to be built and tested on any kind of operating system

1.3.3 Intermediate Code Generation:

Cross-platform [16] development can also be done by depending on pre-existing software that hides the difference between the various platforms. Java programming language [11] is an example of cross-platform software. Java code is compiled into an intermediate code which runs on a virtual machine called JVM (Java Virtual Machine) which interprets and executes Java bytecode. The JVM is available for various platforms of interest. The use of the same bytecode for all platforms allow Java to be described as "Compile once, run anywhere"[7].

Our approach does not depend on any existing software. We simply create multiple versions of the same program for different platforms – The Windows version of the program might have one set of source code files and the Linux version could have another. The architecture of the Repository and Test bed system helps in the coordination of the source code files for various platforms and reduces the application development time considerably.

1.3.4 Web Services Portability:

XML Web Services follow a service-oriented architecture using a messaging protocol called SOAP (Simple Object Access Protocol) [11]. Web Services written on various platforms can communicate with each other. The role of Web services is to make it easier

to tie applications running on heterogeneous platforms together; to help them overcome the communication gaps that arise from decisions to use one development environment over another; and to help abstract such choices so that developers no longer have to keep track of what operating system or what development environment or what technology decisions have been made [8].

The application is built on different platforms and made interoperable using XML Web Services. Web services have a fairly high overhead associated with the standard XML-based Simple Object Access Protocol (SOAP) that has been the standardized basis for web service communication. Our approach using the Software Matrix also uses an XML based message passing protocol but we focus on building complete applications using the Matrix platform rather than interoperating the various parts of the system, and there is no need for a SOAP envelop wrapper to support the communication infrastructure.

1.4 Our Approach:

We propose a cross-platform framework for building heterogeneous systems which span multiple platforms using the software matrix. The framework consists of a repository for smart storage of software products, a Testbed for each platform which is a regression testing framework and one or more RTClients which is a unified interface for controlling the Repository and the Testbed.

The framework provides a single code base repository for all the platforms of interest. We use meta-data stored in the repository to configure source code for the different

platforms of interest. This ensures that configuration need not be done in all the machines every time the application is tested. The framework maintains the source code, builds and tests them and logs results into the repository.

The developers have the opportunity to continuously build and test each module from the ground up in all the platforms. Building a module automatically builds all other modules this module is dependent on. This is facilitated by using repository meta-data that describes dependency relationships between code modules, and is the primary mechanism of indexing code components in the Repository. The dependency-based meta-data drive processing reduces risks that might arise if we develop in one platform first and port it to others. All documentation, test results, dependencies, specifications and source code for each module are stored along with its metadata using XML manifests. The framework we build will also demonstrate how the Software Matrix is useful in developing such distributed systems which span multiple platforms.

In this research, we build the Software Matrix framework using the C++ programming language. We use the Software Matrix to develop the Repository Testbed system. This system will support the development of cross-platform applications using C/C++, but could easily be extended to support software written in other languages as well. We demonstrate cross-platform development by building an application which uses the Repository Testbed system for source-code control and demonstrate building of the application for various platforms of interest.

Chapter 2 – Software Matrix Technologies

This chapter focuses on the technologies used in this research. We describe the Software Matrix technology, as developed in recent research activities at Syracuse University, and extensions specific to this research.

2.1 Tools and Architectures Used:

The framework for cross platform development is developed as a distributed system with a repository server, a testbed server and RT clients to use the system. The various components of the system are developed using the Software Matrix technology. This chapter explains the extensions made to the Software Matrix technology through this research.

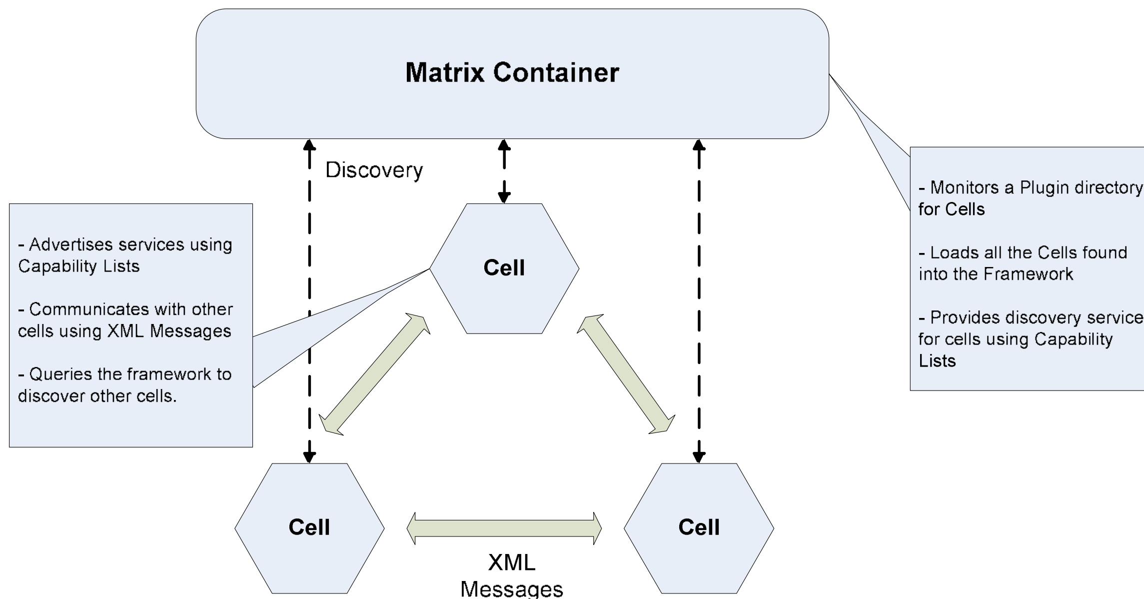
2.1.1 Software Matrix:

The Software Matrix framework is an architecture which was developed by previous research work by Riddhiman Ghosh [1] and Anirudha Krishna [10]. The technology is used for building components that can be reused with almost no transformation cost. The Matrix is a collection of code blocks called cells which are the building blocks out of which applications are built. The cells interact with each other using XML Message Passing. The interaction between the cells is governed by a mediator.

The Framework is composed of well defined components called Cells and a Mediator based Communication Framework that allows interaction between the cells. Such a configuration allows loose coupling of cells to such an extent that the cells can be placed

in a predefined folder. The Matrix has three important components – The Cell, Message Passing Infrastructure and dynamic construction.

Figure 2.1 Software Matrix

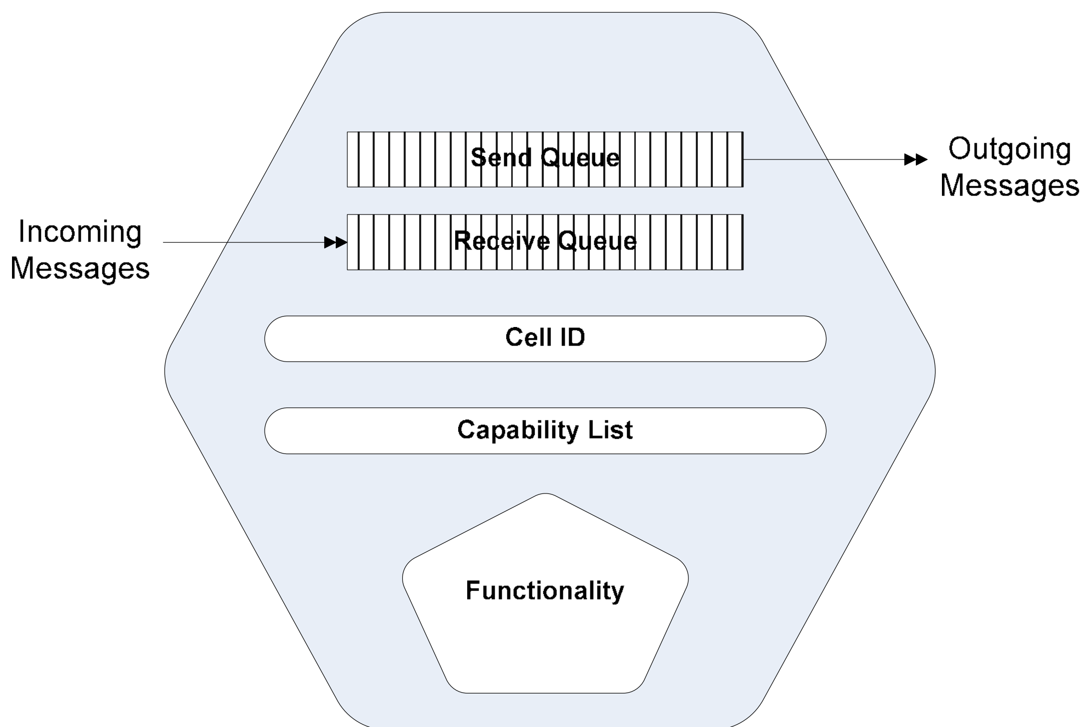


The Matrix Framework contains a Loader which monitors a Plug-in directory. The Plug-in directory contains all the cells that need to be loaded into the Matrix. The Loader loads all the cells in the Plug-in directory into the Software Matrix. It registers all the cells into the Matrix. Each cell contains a capability list describing the services it provides. The mediator contains information about the capabilities of each cell in the Matrix. The mediator provides service discovery to the cells requesting a service using this information. All communication between the cells is in the form of XML Messages.

2.1.1.1 Matrix Cell:

The Cell is the basic building block of the Software Matrix. It is similar to classes in object oriented design but is larger in scope, usually demonstrating component level functionality. Applications are built by combining cells that possess the desired characteristics.

Figure 2.2 Matrix Cell Structure



Since all the components in the Software Matrix are cells, they can either be servers or executive cells. The server cells provide services that can be used by other cells. The executive cells delegate responsibilities to the server cells and control the flow of the

application. All UI components are generally built as executive cells. Each Matrix Cells consists of certain common components which are discussed below:

2.1.1.2 Cell ID:

The Cell ID is a unique ID assigned to every cell instance when it is created at runtime in the Software Matrix. Each cell instance is discovered by the Matrix using this unique ID.

2.1.1.3 Capability List:

Each cell advertises the services provided by it using a Capability List. Each service provided by a cell is given a name called a Message Type. The Capability List of each cell is the collection of all the Message Types supported by the cell. The services of a cell are consumed by other cells using the Message Types in the Capability List.

2.1.1.4 Message Queues:

All communication between the cells is through message passing. Cells can send and receive messages from multiple locations at different times. A send queue is used to buffer outgoing messages and a receive queue is used to buffer the incoming messages so that none will be lost when a cell is busy processing a previous request.

2.1.2 Functionality:

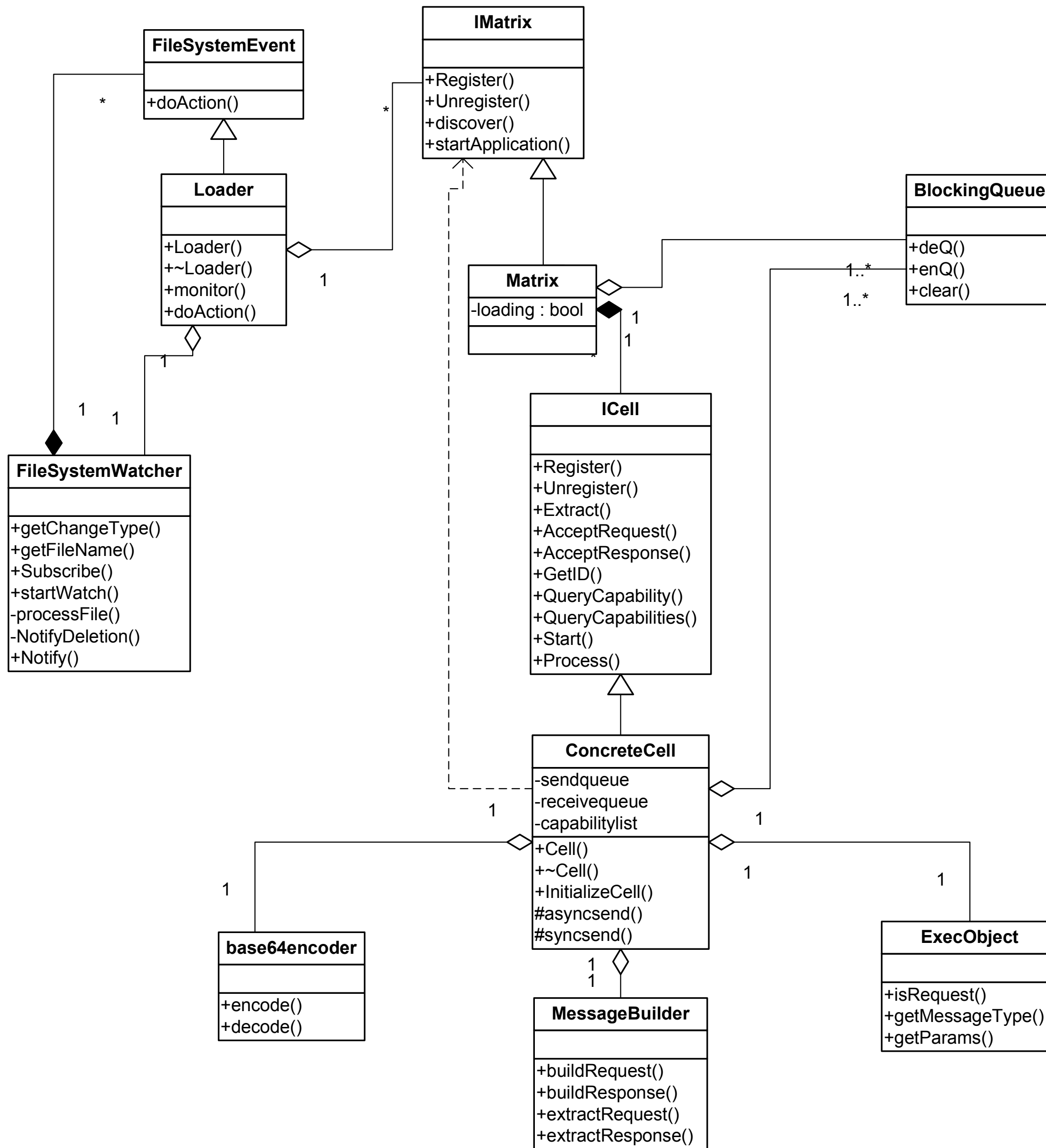
Functionality is the processing that is performed when a cell receives a service request. The parameters contained in a request message are used to process the request and generate a response.

All Cells follow a common protocol that specifies how to register and unregister with the Matrix, advertise their capabilities, send and accept messages and communicate with other Cells (communication could be synchronous, asynchronous or one-way). Also, every Cell has an entry-point, and is given a chance to execute on being plugged in. This decides whether a Cell will be a passive server or itself actively seek collaboration from other peers, depending upon whether the entry-point is empty or not.

2.2 Design:

We shall now discuss the implementation details of the Software Matrix. The Software Matrix was developed for this research using C++. The cells were written as dynamically loaded libraries (dll). The class diagram of the Software Matrix, below, shows the various classes and interfaces and the relationships between them.

Figure 2.3 Software Matrix – Class Diagram



2.2.1 Loader:

The Loader is responsible for identifying and loading all the valid cells into the Software Matrix. The Loader module uses a FileSystemWatcher to periodically monitor a "plug-in" directory for cells. The FileSystemWatcher notifies the Loader whenever a new cell is

dropped into the "plug-in" directory. The Loader then loads and registers the new cell into the Software Matrix.

2.2.2 Matrix:

The Matrix consists of the Capability Lists of all the cells registered into the Software Matrix. It provides the discovery service for the cells in the Software Matrix. Given a Capability, it returns the address of the cell providing the service. The Matrix also ensures that cell execution begins only after all the cells are loaded into the Matrix. This is accomplished by using a Blocking Queue which queues up the messages sent before all the cells are loaded into the Matrix.

2.2.3 ICell:

The ICell interface is one from which all valid cells in the system must derive. It defines the protocol that should be followed by all the cells in the Software Matrix. All concrete cells that are derived from the ICell interface are provided default implementations for the functions in the ICell interface. Some important functions which perform message-passing and so aid in cross-platform development are discussed below:

- **AcceptRequest(std::string& message)** – This method is used to enqueue the request XML message from other cells into the queue for processing.
- **AcceptResponse(std::string& message)** – This method is used to enqueue the response XML message from the server cell into the queue.
- **syncsend(std::string& capability, std::string& message, long WAITTIME) –** This method is used to send execution request message to

another cell. Blocks efficiently for the WAITTIME or till the corresponding response message from the server cell have arrived.

- **asyncsend(std::string& capability, std::string& message) –** This is the asynchronous version of the syncsend operation. The client cell does not block for response messages. This functionality can be used when the architecture of the application is intended to be asynchronous.
- **Process(ExecObject& obj) –** This method specifies what is to be done in response to the various messages that arrive to this cell. It appropriately delegates calls to the code that actually implements the functionality required to handle a particular message type.

Cells use the ExecObject class to assist in packaging of arguments, return values and other information regarding requests and responses between cooperating Cells in the Software Matrix.

Sample Request Message:


```
<?xml version="1.0" encoding="utf-8" ?>
<Message>
  <Type>Request</Type>
  <NetworkSend>false</NetworkSend>
  <RequestID>bfe8d2cf-2dde-4703-ad0c-4f7cfb30f8fa</RequestID>
  <Name>Matrix.RepositoryExec.GetTopItems</Name>
 <Params count="1">
  <Param>
     <ParamName>bWVzc2FnZQ==</ParamName>
     <ParamType>c3RkOjpzdHJpbmc=</ParamType>

<ParamValue>PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0idXRmLTgiPz4
8bWVzc2FnZT48Q29tbWFuZCB4bWxucz0iNyIgLz48L21lc3NhZ2U+</ParamVa
lue>
  </Param>
 </Params>
</Message>
```


Sample Response Message:


```
<Message>
  <Type>Response</Type>
  <NetworkSend>false</NetworkSend>
  <Source>0</Source>
  <Destination>0</Destination>
  <Name>Matrix.RepositoryExec.GetTopItems</Name>
  <RequestID>bfe8d2cf-2dde-4703-ad0c-4f7cfb30f8fa</RequestID>
 <Params count="1">
  <Param>
    <ParamName>cmV0dXJuVmFsdWU=</ParamName>
    <ParamType>c3RkOjpzdHJpbmc=</ParamType>

<ParamValue>PEl0ZW1zPjxJdGVtIHZlcnNpb249IjEiIHBsYXRmb3JtPSJ3aW4z
MiI+RmlsZUlPPC9JdGVtPjxJdGVtIHZlcnNpb249IjEiIHBsYXRmb3JtPSJsaW51
eCI+RmlsZUlPPC9JdGVtPjxJdGVtIHZlcnNpb249IjEiIHBsYXRmb3JtPSJ3aW4z
MiI+VGhyZWFkczwvSXRlbT48SXRlbSB2ZXJzaW9uPSIxIiBwbGF0Zm9ybT0id
2luMzIiPnhtbFBhcnNlcjwvSXRlbT48SXRlbSB2ZXJzaW9uPSIxIiBwbGF0Zm9y
bT0ibGludXgiPnhtbFBhcnNlcjwvSXRlbT48L0l0ZW1zPg==</ParamValue>
  </Param>
 </Params>
</Message>
```

2.2.4 **Serialization/Deserialization:**

The data such as arguments in request messages and return values in response messages should be serialized and deserialized so that the resulting persistent data can be encapsulated in XML messages to be passed across various platforms. A base64 encoder/decoder is used to serialize/deserialize the data in the request and response messages.

The Software Matrix was extended to support distributed Self Healing Systems by Anirudha Krishna [6]. The Self Healing Architecture provides fault tolerance using a Repository of Cells. All the cells which are a part of the distributed system are hosted in the central repository. Failure of any node in the distributed system results in the cells being downloaded from the Repository and used in place of the failed node.

Service discovery for the distributed software matrix was provided by a centralized addressing server which holds information such as the IP address of the location of the cells in the distributed system. The combination of the addressing server and the repository provides a framework which supports fault tolerant distributed development.

Failure of the Addressing server and the repository can be handled by using mirror servers. In case of failure of any of the servers, the mirror still provides the service with no interruption.

2.3 Extensions to the Software Matrix:

We will discuss the various improvements to the Software Matrix technology through this research.

2.3.1 Improving Efficiency:

In the Software Matrix, all communication between the cells is through a Mediator. This implies that the messages are passed to the Mediator every time when a communication to a different cell is needed. The Mediator then parses the message and discovers the cell providing the requested service and forwards the message to the discovered cell. We improve the efficiency of the Matrix by adding “Matrix Query” functionality to the Software Matrix technology. The matrix query is used to query the matrix for the cell providing a particular service. If the service is present in the local machine, it returns a pointer to the cell providing the service. Now the message is directly sent to the destination cell using the pointer returned by the Matrix Query. If the service is present over the network, it returns a network cell which searches the Matrix network for the cell providing the service. Hence instead of passing the whole message to the Mediator every time, we improve efficiency by using a “Matrix Query” to discover the destination cell and pass the message to the discovered cell directly.

In this research, we improved the Software Matrix framework to make it efficient by enabling communication between the cells without passing through the Mediator. We also added a lightweight Query Service to discover the cells in the Matrix.

Chapter 3 - Repository Testbed System

In this chapter, we discuss the development of the Repository component of our cross-platform build system.

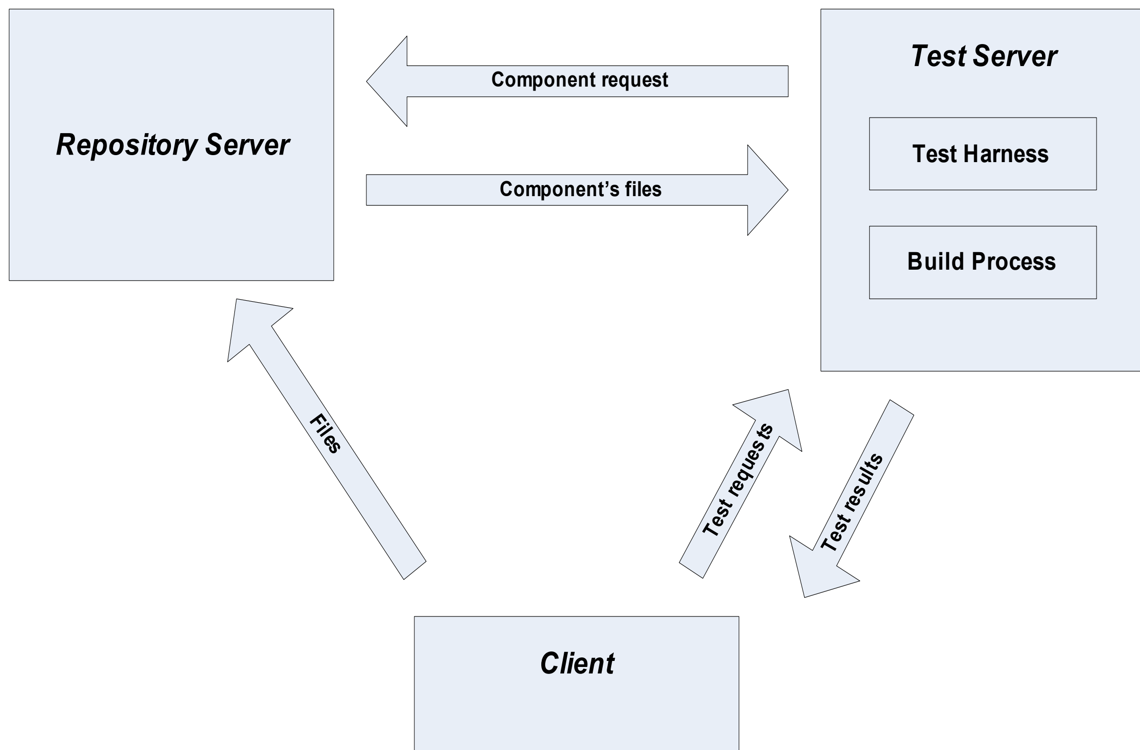
3.1 Repository Testbed System:

The Repository Testbed system is designed to help manage and test code produced in a large software development project. The Repository and Testbed system supports cross-platform development by providing facilities for source code check-in, check-out and versioning, defining building and executing test configurations remotely in each platform and reporting test results for that platform.

The Repository Testbed system is distributed with an underlying transport mechanism that uses message-passing through Software Matrix technology. Consequently the system can be composed of parts that reside on multiple machines in any set of locations accessible from the web.

The Repository and Testbed Server is composed of a Software Repository Server, a Testbed Server and an arbitrary number of RT clients. The Software Repository Server provides the maintenance and tracking of software files and their configurations for various platforms. The Testbed provides support for building libraries and execution images on the target platform of interest from the source code obtained from the Repository Server.

Figure 3.1 - Repository- Testbed Structure



The Repository Server uses XML-based meta-data to describe items and components and their relationships. An item refers to one or more source code files and their associated

documentation. A component consists of a root item and all the items on which it depends. Using a single component name, all of the source and/or documentation necessary to build or describe an entire component, at any level, may be extracted from the repository. The repository also has a versioning scheme designed to support cross platform development and collaboration of many teams across a large project.

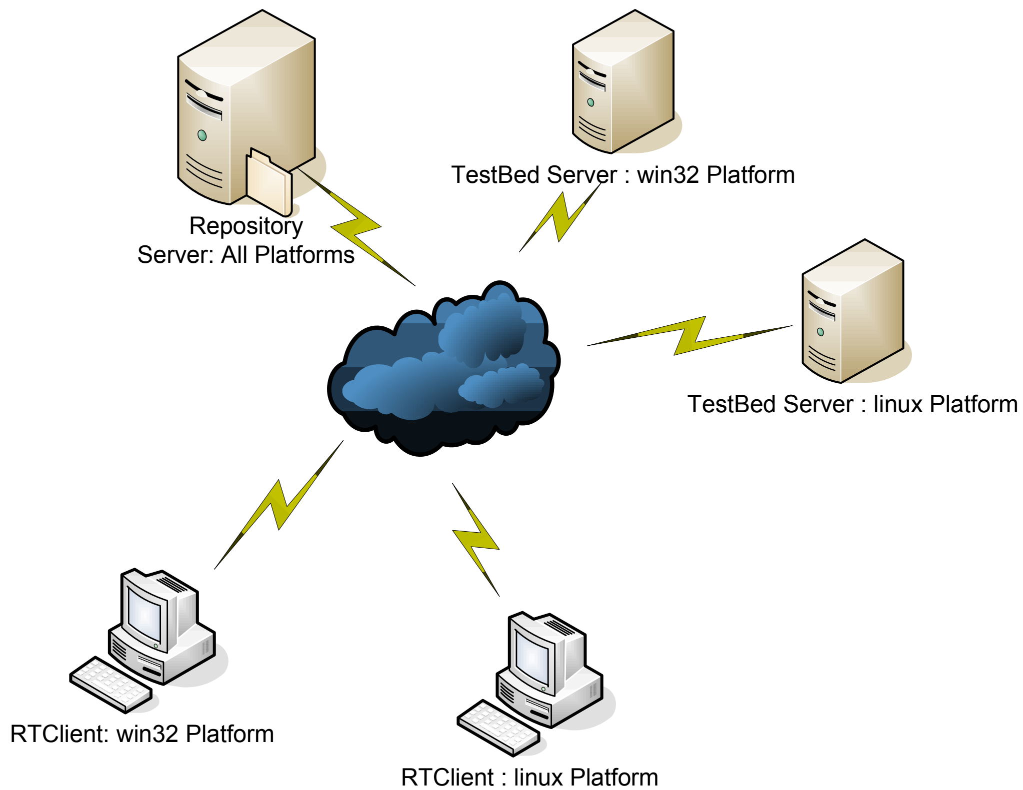
The Testbed Server conducts test sequences using a Test Harness structure. The Testbed Server can extract items and components from the Repository Server and use the Test Harness for testing them. The Testbed Server will run many concurrent test configurations for an evolving software development project. A test configuration consists of a series of test and tested code libraries which are dynamically loaded and executed in the test harness. It is intended that testing may be eventually become continuous, and configurations will grow to encompass the entire software development project. Individual clients will run scaled down versions of these test configurations, of interest to a specific team.

Clients will initiate the creation of software products, check them into the repository and then define and request execution of specific test configurations on the Testbed.

3.2 Network Structure:

The Repository and Testbed system supports cross-platform development. The Repository Server has a single code base which contains the source code for all the platforms in which the application has to be deployed. The source code should be built and tested in all the platforms in which the application has to be deployed.

Figure 3.2 Repository and Testbed – Network Diagram



The Repository Server organizes the source code for the various platforms using the XML based meta-data and a versioning scheme designed for supporting cross platform development, described above. It provides services for extracting the source code from any platform using the Software Matrix.

The cross platform application needs to be deployed on various platforms. Each platform consists of a Testbed Server with a Test Harness structure. The Testbed Server uses the services of the Repository Server to extract source code that belongs to the platform in which it runs. These components are then built using a build process in the Testbed Server. The Test Harness in the Testbed Server is used to test the compiled test libraries and report test results to the Clients requesting the test run.

The RTClients are present for each platform of interest. They are the operational centers of the Repository Testbed system. They are used to check-in and checkout source code files into the repository for their platform. They are also used to issue test requests to the Testbed Server for any platform of interest.

3.3 Repository Testbed system with the Software Matrix:

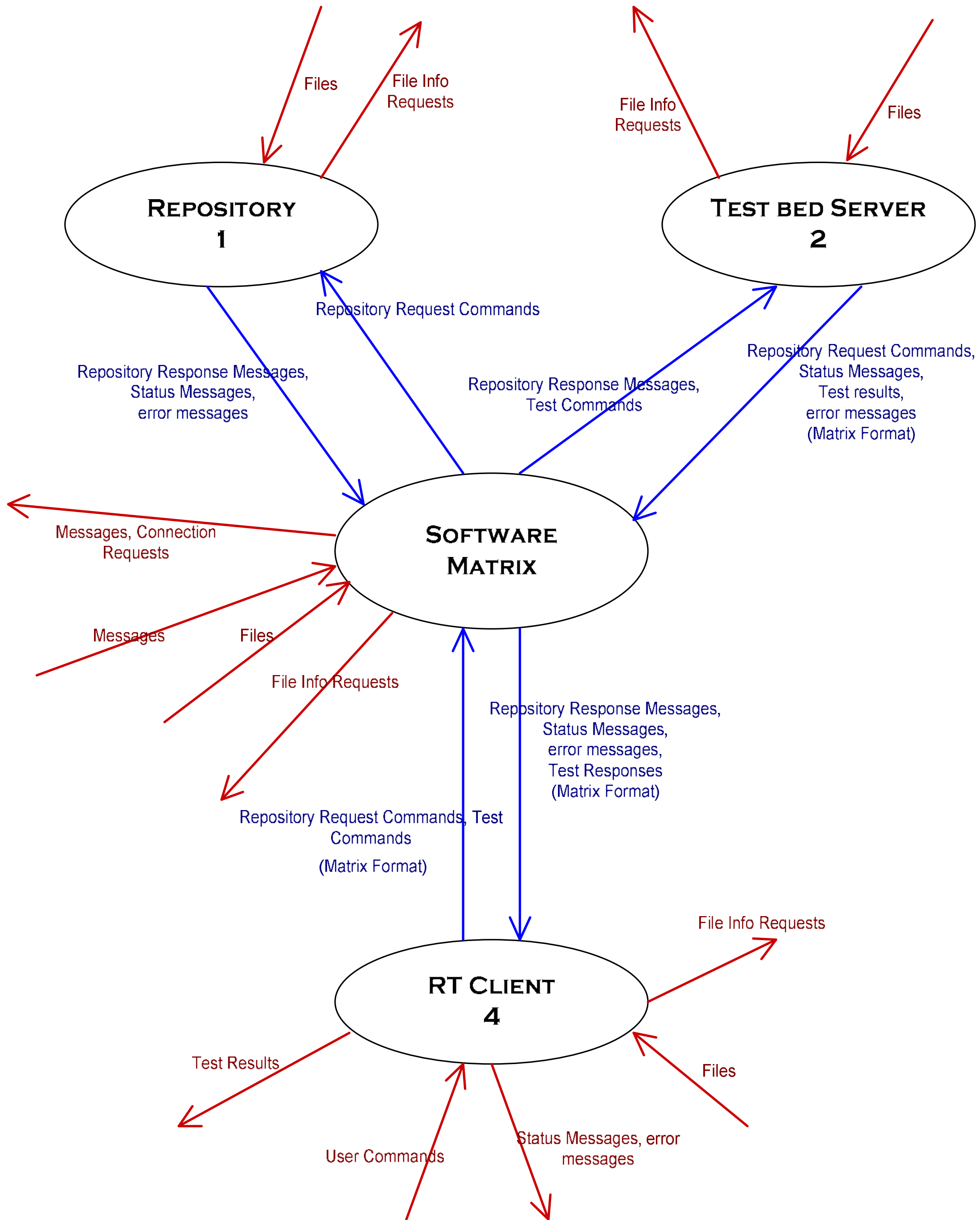
The Repository and Testbed system is composed of parts that reside on different machines running different platforms on a network. The Software Matrix technology is used for communication between the different parts that are distributed over the network. The Software Matrix technology uses an XML-based message passing protocol for communication. Since XML is cross-platform, the Software Matrix supports communication between systems that reside on different platforms as well.

The Repository Server and the Testbed Server are built using the Software Matrix technology. We have RTClients and Testbed Servers for every platform of interest. So communication between the servers and clients which reside on multiple platforms is accomplished using the Software Matrix technology.

The Repository Server and Testbed Server are composed of Software Matrix cells. The services provided by the Repository Server and the Testbed Server are advertised using the Capability List in the Software Matrix. The communication between the different cells in the Repository and Testbed Server is also through the Software Matrix.

The dataflow diagram, shown on the next page, shows the major message traffic between the software matrix, Repository, Testbed Server, and RTClients.

Figure 3.3 REPOSITORY AND TESTBED SERVER



3.4 Repository Server:

The Repository Server is a tool designed to manage code development, control, and testing. The Repository Server supports various services to support source code control and testing.

3.4.1 **Definitions:** The various definitions that will be used in the Repository Server are discussed below:

3.4.1.1 Version:

A version is a number generated in sequence by the Repository Server that is assigned to a file. This number is encoded in the file specification using the convention: filename.VersionNumber.Extension

Each checked-in modification of a file results in a new version number, generated sequentially. Should a version be removed from the Repository Server, versions of this file with higher version numbers will not be re-versioned. All files stored in the repository such as source code files, documentation files and the XML manifests will be versioned.

3.4.1.2 Item:

An item is a named, versioned XML metadata and all the files on which it holds references, excluding references to other items. Each item refers directly to one or more source code files and configuration files associated with the source code files. The XML

metadata of the item can also hold references to documentation files and other items on which it depends. The uniqueness of an item is determined both by the name of the item and the platform for which it is intended.

3.4.1.3 XML Manifest:

A named, versioned XML file that groups together items with the same name but which are intended for different platforms. All items within the manifest have the same name and the same version.

3.4.1.4 Component:

A component is one root item and all the items it references, either directly or indirectly. That is, a component is a top-level item and the closure of all its references. The name of a component is the name of its top-level item. Its version is the version of the top level item. Programs are items that refer only to module items, test source files and documentation. Systems are items that refer only to program items, test source files and documentation.

3.4.1.5 Check-in:

The process of storing all the files of an item in the repository and providing sequenced version numbers as described above is called a check-in. On check-in, the item is given a unique name for a given platform and a version number. Check-in does not replace files with earlier version numbers. Once an item is checked in, it is immutable.

3.4.1.6 Check-out:

The process of transferring files of an item from the Repository Server to the RTClient for the purpose of modification is called a check-out. Items checked out can be checked back in to create newer versions.

3.4.1.7 Extraction:

The process of transferring a component's files from the Repository Server to the RTClient and/or Testbed Server is called Extraction. No Extracted items may be checked back in. The RTClient uses the Extraction service to view the files whereas the Testbed Server uses it for building and testing on various platforms.

3.4.1.8 Browsing:

The process of viewing the items and components in the Repository Server and their dependencies is called Browsing. Browsing an item shows all the files it is dependent on and their relationships with other items.

3.4.2 Component Representation:

A key property of the Repository is that it manages source code as items and components. An item is a one or more production source code files, combined with documentation files, and related information that helps users and other parts of the system use the code. Each item may refer to lower-level items, and is bound to its constituent parts with an XML metadata. Each reference to source code, documentation, lower level items, or other information is encoded by XML tags and attributes within the XML metadata.

A Component is an item and all the lower level items it references. All code in the repository is accessed as either items or components. If we want to build a component we simply extract the component by name from the repository which yields the source code of the top-level item in the component and all items on which it depends.

Items define specific versions of modules, programs, or systems for a particular platform. Thus, if a client extracts a program, all the files required for the version of the program requested for a platform will be sent to the client, if it does not already have them with the correct version. Here we show how items and components are represented in the Repository for a single platform.

The Systems, Programs, and Items which have the same name but which are intended for different platforms are grouped together using the XML Manifest. The XML Manifest is an aggregation of the XML meta-data for all items with the same name but which run on different platforms.

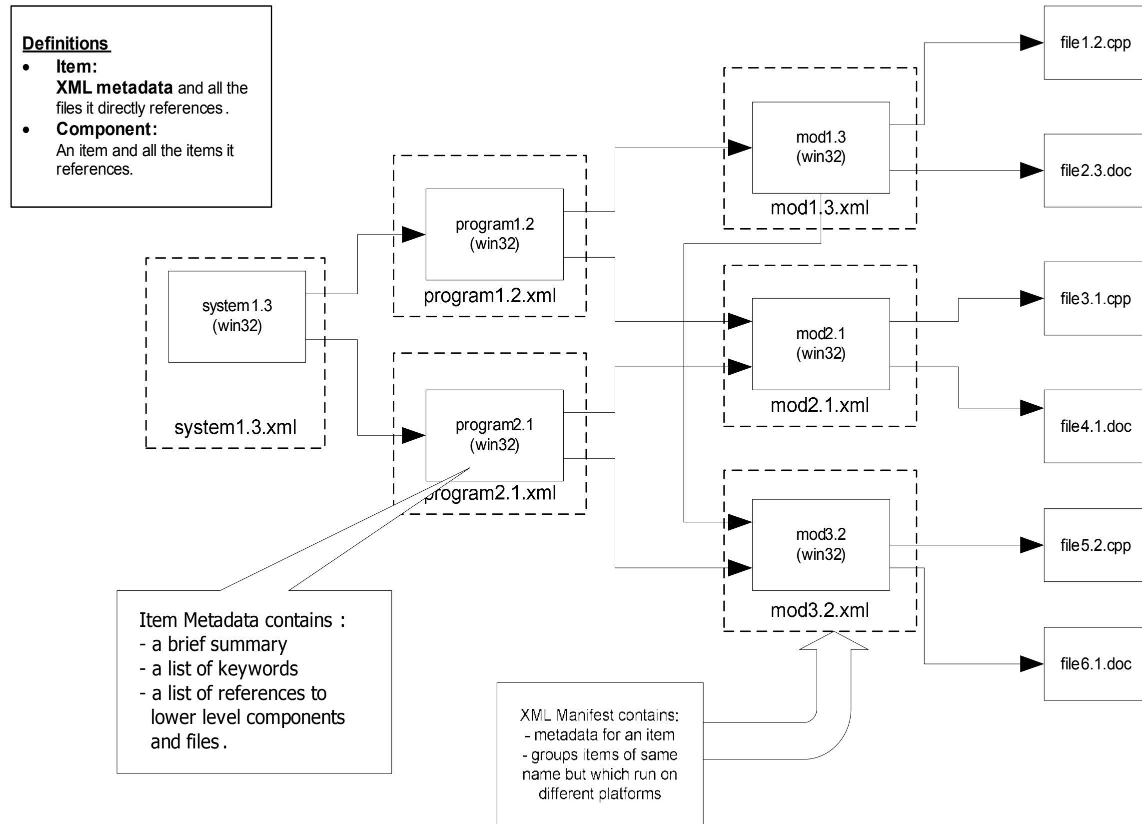
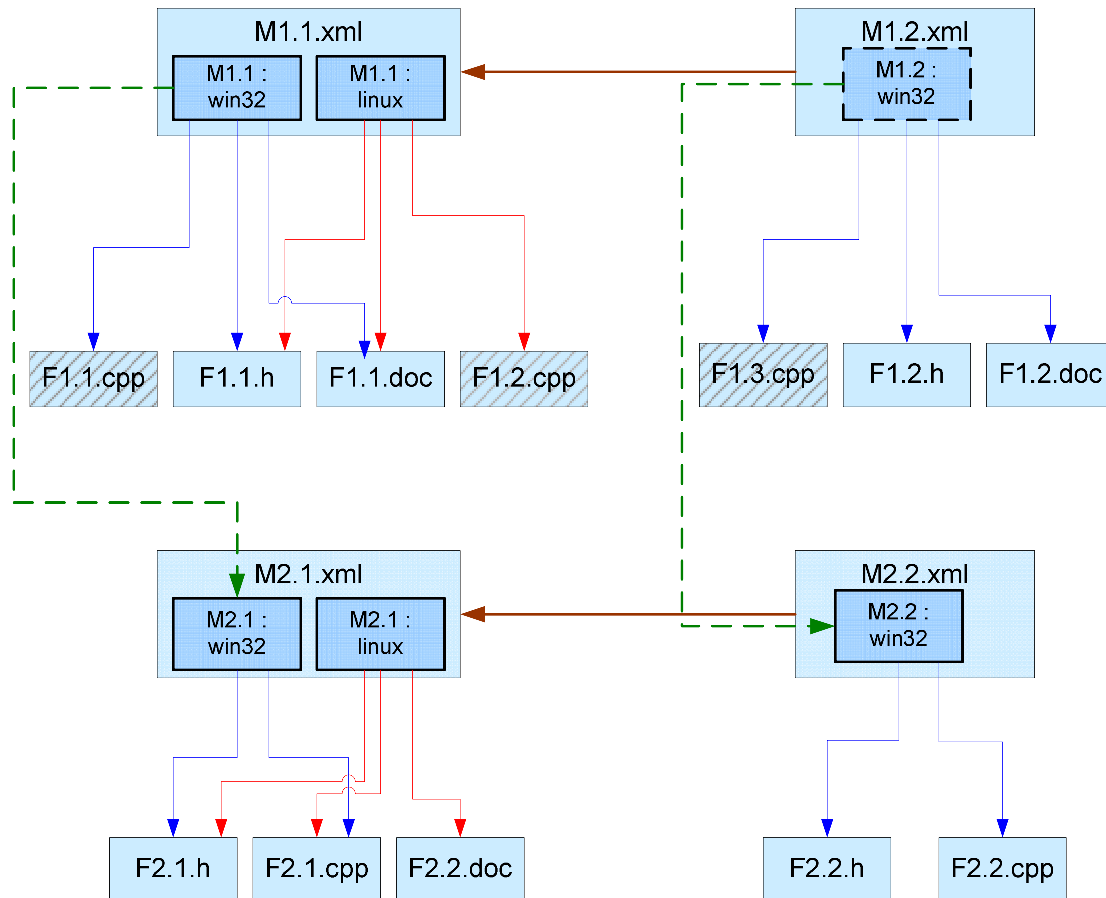


Figure 3.4 Component Representation

3.4.3 Version Control:

The Repository supports the organization of its items and components for various platforms. To enable the smooth collaboration of different teams working on the same project, we designed a version control scheme that also supports cross platform development. That is, various teams writing code for different platforms can collaborate during the development of a project using the Repository Testbed System.

Figure 3.5 CROSS-PLATFORM VERSION CONTROL



In the above diagram, M1.1 and M2.1 are two items in the repository. It is shown that M2.1 is dependent on M1.1. M1.1.xml and M2.1.xml are the XML Manifests that group M1.1 and M2.1 respectively for windows and linux platforms. The version numbers are assigned to the items sequentially. So M1.2.xml is the newer version of the XML Manifest M1.1.xml.

The Manifest M1.1.xml groups together M1.1 for the windows platform and M1.1 for the linux platform. In this case, the items for both the windows platform and the linux platform share the same header file and the documentation file but they have their own implementation file. Note that the version numbers of the files referenced by the item are not the same as the version number of the item itself. This is because, it is possible that two items can share the same file as discussed above.

The Manifest M2.1.xml groups together M2.1 for the windows platform and M2.1 for the linux platform. In this case, both the windows and the linux items share the same header file and the implementation file. This occurs in cases where the source code in the implementation file is portable across the various platforms.

Now, if we checkout M2.1 for the windows platform and make some modifications and check it back in, it results in the creation of a new item called M2.2 since each check in results in the creation of a new version number in sequence. This also results in the

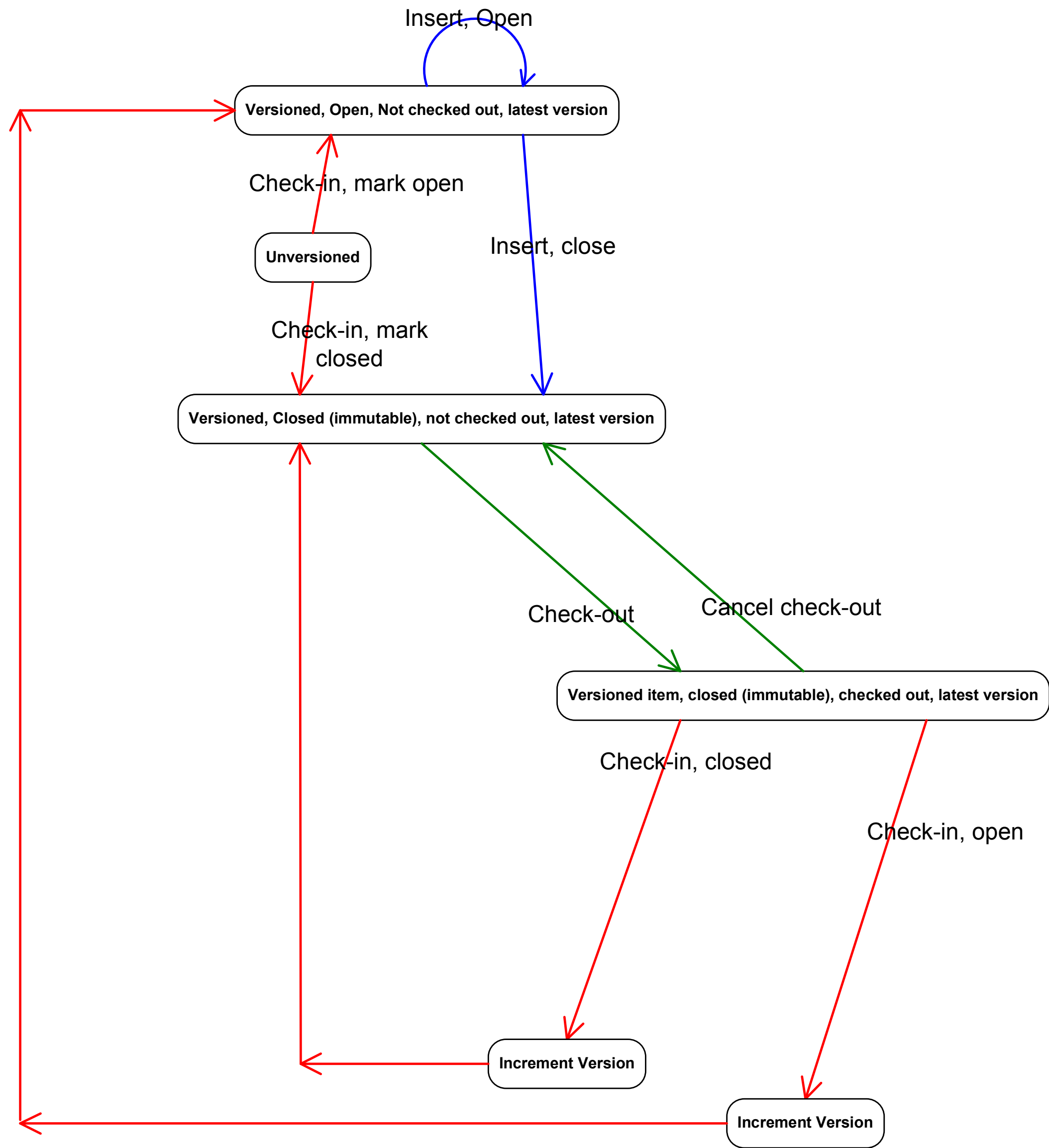
creation of a new XML Manifest called M2.2.xml to hold the XML metadata for item M2.2.

Checking out the item M2.1 for the linux platform and checking it back in will also create a new item called M2.2 for the linux platform. The metadata for M2.2 will be stored in the XML Manifest M2.2.xml which was created previously. So an XML Manifest is created only if it does not exist.

Item M1.1 : win32 is dependent on item M2.1 : win32 as shown above. Now we have created a new version for M1.1 : win32 called M1.2 : win32. The new version M1.2 : win32 now can point to the newer version M2.2 : win32 or it can choose to point to the older version. In this case we point it to the newer version.

If most of the source code is portable across various platforms, this version control scheme makes sure that the files are shared across the different items for the various platforms avoiding redundancy of the source code in the Repository Server. While testing an item for a particular platform, the Repository Server uses the platform information present in the XML metadata to route the correct source code to the correct platform, hence supporting cross-platform testing.

Figure 3.6 STATE DIAGRAM FOR REPOSITORY SERVER



The above diagram shows the different states in which an item is present in the Repository. Each check-in of an item results in the creation of new version of the item. This means that each and every modification of an item will result in the creation of the new version. This can result in the creation of large number of versions in the Repository. To avoid this, we introduce "Insertion" of an item into the Repository. We also introduce two ways in which an item can be checked-in to the Repository.

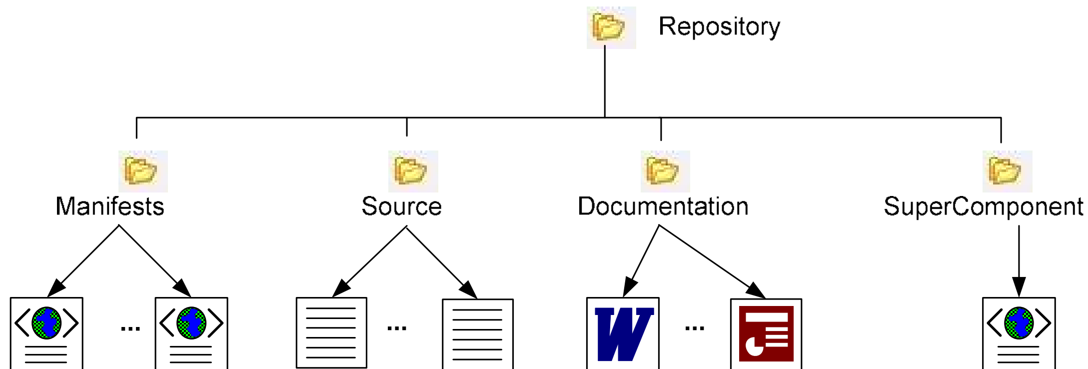
Every item in the Repository can be in any one of the two states Open or Closed. When an item is checked-in as "Open" in the Repository, checkout of the item will fail. In order to make changes to the "Open" item, we extract the item, modify the source code and then "insert" the item into the Repository. Insertion of an item into the Repository will result in overwriting the previous files present in the item with newer ones without incrementing the version number of the item. If we do not want to further modify the item, the item can be inserted as "Closed". This makes the item immutable.

When the item is checked-in as "Closed" in the Repository, checkout of the item will mark the item as checked out. Any further checkout requests will fail before checking in the item again. The checked-out item is modified and checked-in again, resulting in the creation of a new version number for the item.

3.4.4 File System Layout:

The organization of the contents of the Repository in the Operating System's File System is shown below:

Figure 3.7 File System Layout of Repository Server



The Repository File System consists of Manifests, Source, Documentation and a SuperComponent. The Manifests folder holds all the XML Manifests which are the containers of an item's metadata. The Source folder contains all the source code files present in the Repository. The Documentation folder contains all the documentation files supplied with check-in in the Repository. The meta-data in the XML Manifests contain references to the files in the Source and Documentation folders.

The SuperComponent folder contains a single XML file called super.xml. It contains the reference counts of all the items which are present in the Repository.

3.4.5 XML Meta-Data of an Item in the Repository:

This is an example of the meta-data of an item in the Repository. It represents the Threads item which consists of source files and depends on another item in the Repository – Lock version: 1, platform: win32.

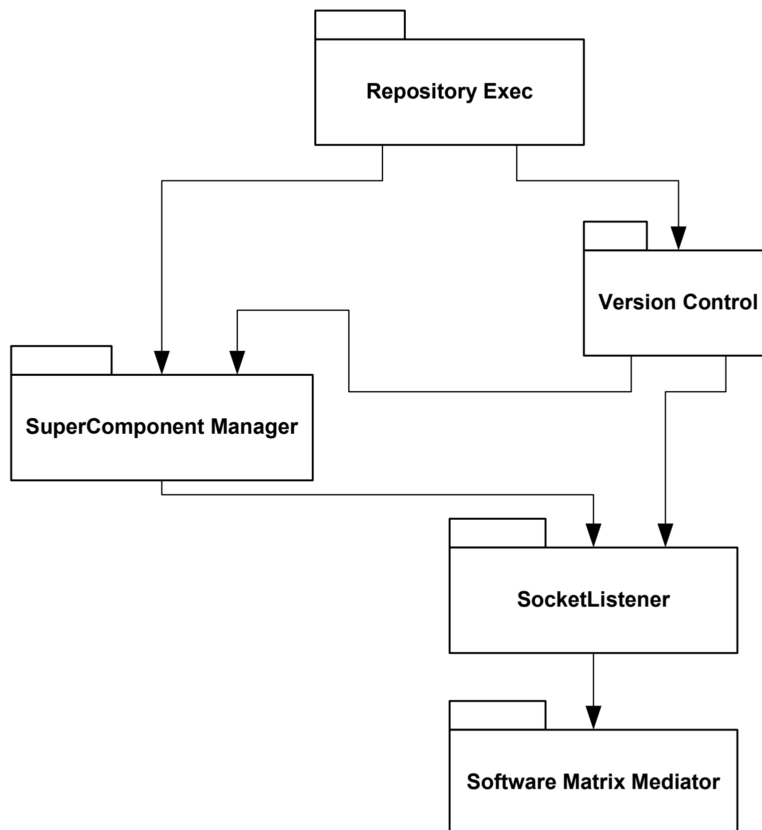
```
<manifest>
<platform>
<pinfo>
  <name>Threads</name>
  <version>1</version>
  <description>Thread class</description>
  <compiler>VC++</compiler>
  <status>closed</status>
  <pname>win32</pname>
  <outputType>exe</outputType>
  <outputFilename>threads.exe</outputFilename>
  <checkedout>no</checkedout>
  <miscCompilerOptions />
  <miscLinkerOptions />
  <references>Lock.1.xml:win32</references>
  <sourcefiles>threads.1.h</sourcefiles>
  <sourcefiles>threads.2.cpp</sourcefiles>
</pinfo>
</platform>
</manifest>
```

3.4.6 Repository Partitions:

The Repository Server is built using the Software Matrix technology. The Repository Server is partitioned into "cells" in the Software Matrix. The Repository Server consists of four cells:

- RepositoryExec
- VersionControl
- SuperComponent Manager
- SocketListener

Figure 3.8 REPOSITORY SERVER – PACKAGE DIAGRAM



3.4.6.1 RepositoryExec:

The RepositoryExec cell is the main executive which delegates responsibilities to all the other cells which compose the Repository Server. The RepositoryExec accepts all the service requests to the Repository Server, uses the other cells to execute the service requests and returns the result to the requesting client.

The RepositoryExec accepts the following services provided by the Repository Server:

- Check-In
- Check-Out
- Extract
- Insert
- Browse
- TopItems

In order to provide the above services, the RepositoryExec needs the services of the VersionControl and the SuperComponent Manager cells. The SocketListener cell is used to communicate with the RTClient and Testbed Server in different machines running different platforms. The SocketListener module is also a cell in the Software Matrix.

3.4.6.2 Version Control:

The Version Control cell manages the creation, updating and deletion of an item in the Repository. It is also used to calculate the next version of a given file in the Repository. The RepositoryExec cell uses the services of the VersionControl to Checkin, Checkout and Insert items into the Repository.

The Version Control cell provides the following services:

- AddItem
- UpdateItem
- ExtractItem
- DeleteItem
- getNextVersion

The VersionControl cell helps in finding the XML manifest in which the item's meta-data has to be added, calculation of the next version number of both the item and the files on which the item depends on. It also helps in extracting a named item from the Repository for a given platform.

The Version Control cell also uses the services of the SuperComponent Manager to validate the checkin and deletion of items from the Repository.

3.4.6.3 SuperComponent Manager:

When an item is checked-in into the Repository, no other item will be referencing this item. In order to allow other items in the Repository to find this item, we maintain references to all items in the Repository in the Super Component.

The Super Component Manager provides the following services:

- IncrementReferenceCount
- DecrementReferenceCount
- Backup
- AddItem
- RemoveItem

The Super Component maintains reference counts of all the items in the Repository. Only if the reference count of an item is "0", the item is allowed to be deleted from the Repository. On check-in the item is added to the Super Component and on deletion it is removed from the Super Component.

The Super Component can also be used to determine the number of items referencing a particular item. This will help the developer of this item in notifying the dependent items if he creates a new version. The dependent items can then link to the new version if necessary.

3.7 Check-In:

Check-in is the process of storing all the files of an item in the Repository and providing sequenced version numbers to the item. The Repository Server supports the check-in of items and test configurations from RTClients.

Each check-in results in the creation of a new version number for the checked-in item with a unique identifier for a given platform. Any source code and documentation files supplied for the checking process are given new version numbers. If the files referred by the checking item are already present in the Repository, they don't need to be supplied during check-in. All of the presented files will then be stored in the Repository and the can be accessed using the XML metadata of the checking item.

Check-in succeeds only if the check-in process presents to the Repository Server an XML metadata for the item and all the files to which it refers, if those files do not already exist on the Repository Server. The check-in process is validated according to the following rules:

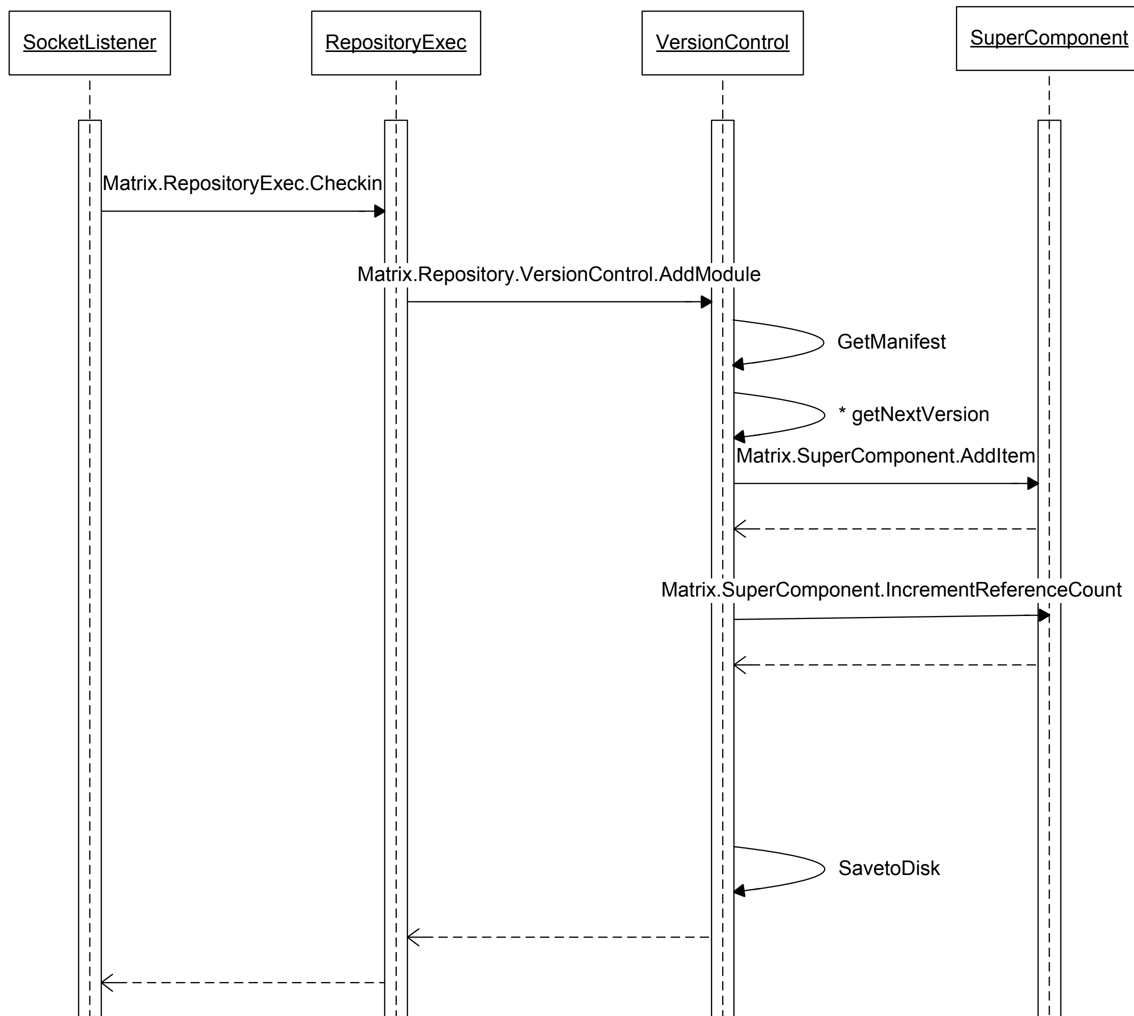
- If the item is un-versioned and has no unique identifier check-in succeeds if the Repository Server does not contain another item with the same name for a given platform.
- If the item file is versioned and has a unique identifier, the latest version of the item must be closed.

- Check-in supports both an “Open” and a “Closed” status. When an item is checked-in as “Open”, the item may be modified without changing its version by accepting new references to items and files or changing an existing reference.
- When an item is checked-in as “Closed”, the item becomes immutable. No changes can be made to the item once it is “Closed”. Once “Closed” the status cannot be changed to “Open”.
- If any of the items referenced by this item is “Open” then check-in fails.

The XML Manifests are the containers of metadata for an item. The manifest aggregates the meta-data of items which have the same name and version for various platforms. While checking in an item, the Repository Server checks for the existence of an XML Manifest to hold the item’s metadata for a given platform. If the XML Manifest is not present in the Repository, it is created and the item’s metadata is added to it.

For example, let us consider an item with a name called “Tokenizer”. The item can be developed for various platforms such as win32 and linux. While checking in Tokenizer for the win32 platform for the first time, the Repository Server creates a new XML Manifest with the name Tokenizer.1.xml and stores the meta-data for Tokenizer (win32 platform). Now if we check-in “Tokenizer” for the linux platform, it reuses the Tokenizer.1.xml to store its metadata.

Figure 3.9 Check-in Sequence Diagram



The sequence diagram shows the interaction between the various cells in the Repository during the check-in process. The RepositoryExec cell delegates responsibility to the VersionControl cell. The VersionControl cell manages the creation of the XML Manifests and adds the item's metadata to it. It also creates new version numbers for the source code and documentation files supplied with the check-in command. The SuperComponent Manager increments the reference counts of the items on which the

checking item is dependent on. It also adds the checking item into the SuperComponent with a reference count of “0”.

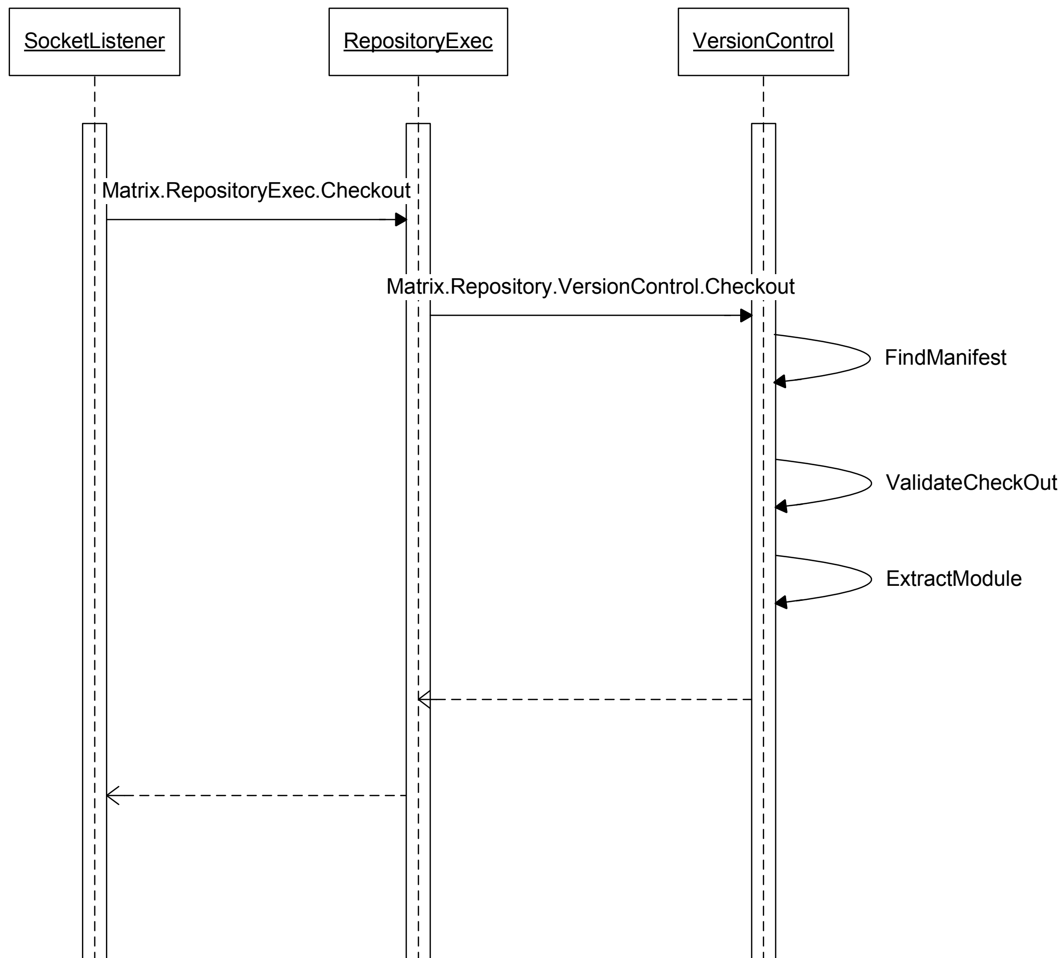
3.4.8 Check-Out:

Check-out is the process of transferring the files of an item to an RTClient for the express purpose of modification. The modified files are then checked into the Repository to create newer versions of items. The Repository Server supports check-out of items and test configurations from RTClients. Once checked out, the Repository Server supports the cancellation of the check-out if requested by the user.

The check-out process is validated according to the following rules:

- Check-out of an item which has previously been checked-out results in a failure of the check-out process.
- Only the latest version of an item for a given platform can be checked-out for modification.
- Only “Closed” items in the Repository can be checked-out by the user. Any attempt to check-out an “Open” item from the Repository results in the failure of check-out.

Figure 3.10 Check-out Sequence Diagram



The VersionControl cell first finds the item which needs to be checked-out from the Repository. Using the metadata, all the files contained in the item are extracted from the Repository and returned to the requesting RTClient. The status of the item is marked as "checked-out" in the Repository. Any subsequent "check-out" requests for this item will fail unless "check-out" is cancelled by the user.

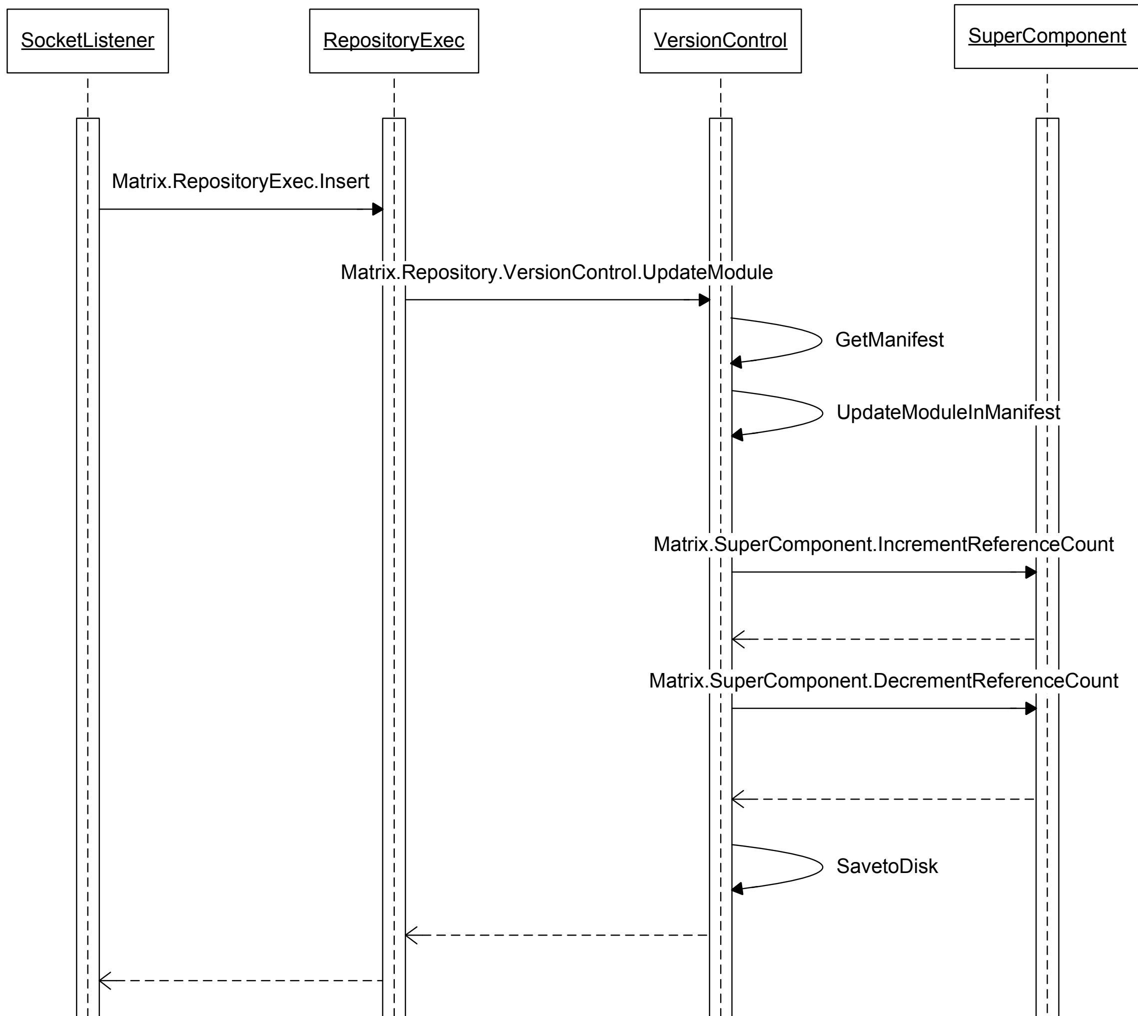
3.4.9 Insert:

Insertion is the process of modifying an open-versioned item in the Repository. Finally the item's version is closed after modification. The Repository Server supports the insertion of items into the Repository from the RTClient.

To avoid the creation of large number of versions for items with a particular name, we check-in the item as "Open". To modify the items without creating a newer version we use the "insert" functionality. Insertion of items which were previously "Open" results in overwriting the previous files with newer ones. When we decide that the item no longer needs to be modified for a particular version, the previously "Open" item is inserted as "Closed". This results in overwriting the previous files with newer ones and marking the item as "Closed".

The sequence diagram shows the interactions between the "cells" during the "Insertion" process. The VersionControl cell finds the XML Manifest which contains the item and updates it. If any of the references of this item is changed, the SuperComponent Manager is notified of the change in the reference count.

Figure 3.11 Insert Sequence Diagram

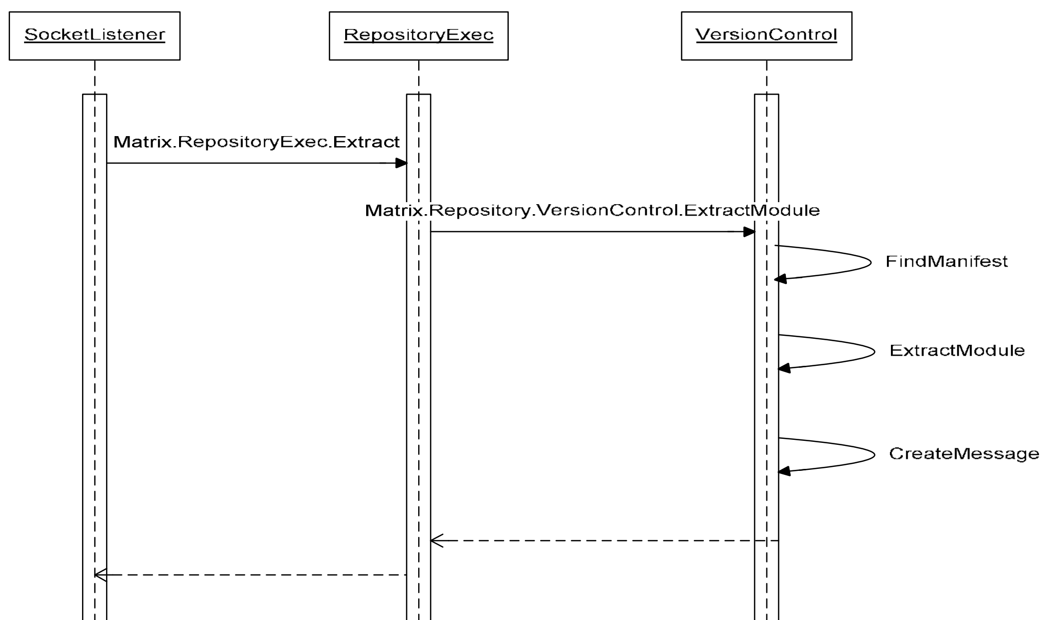


3.4.10 Extract:

Extraction is the process of transferring the component's files from the Repository Server to the RTClient. No items which were extracted can be checked back in to the Repository. The Extraction operation is expected to be used by the RTClient for viewing the item's files and the Testbed Server for building and testing the component.

Extraction does not change the state of the item in the Repository. Extraction of a component is a recursive operation which first yields the top-level item. Using the meta-data of the top-level item, the client requests for other lower-level items recursively thus extracting the whole component. This is handled by a file manager on the target machine, so the RTClient operator simply has to select the top-level component. Extraction of a component thus yields the top-level item and the closure of all its dependencies.

Figure 3.12 Extract Sequence Diagram



Chapter 4 – Testbed Server

This chapter discusses the Test Bed Server Component of this Software Matrix-based research system.

4.1 Test Bed Server:

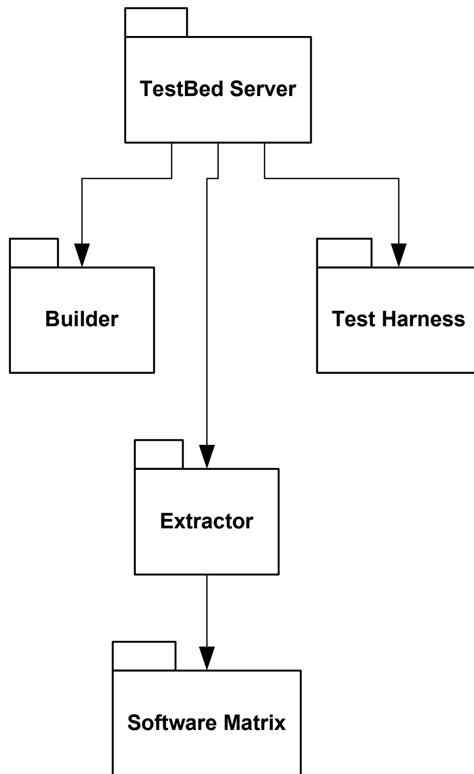
The Test Bed Server is a dedicated server running in each platform in which the source code has to be tested and deployed in. The Test Bed Server consists of a Builder and a Test Harness. The Test Bed Server provides the service of building and testing a component in the Repository. The Test Bed Server accepts test requests from the RTClient to test a component in the Repository Server.

The Test Bed Server uses the services of the Repository Server to extract the components from the Repository. Each item checked into the repository consists of meta-data defining the item. This meta-data consists of the specification for building the item. The Build Process in the Test Bed Server uses this specification to build the item. Since the item's metadata contains references to other items it depends on, the builder uses this information to build the lower-level items first hence building the entire component. The builder thus compiles the entire components and produces test libraries as output which can be consumed by the Test Harness.

The Test Harness is a facility for executing test suites. It loads all the test libraries which are output by the Build Process and executes all the tests that have been made part of the

test suite and announces the number of tests run and the number of tests passed. Test Harness will generate a resulting test report which is sent back to the RTClient requesting the test run.

Figure 4.1 Testbed Server – Package Diagram



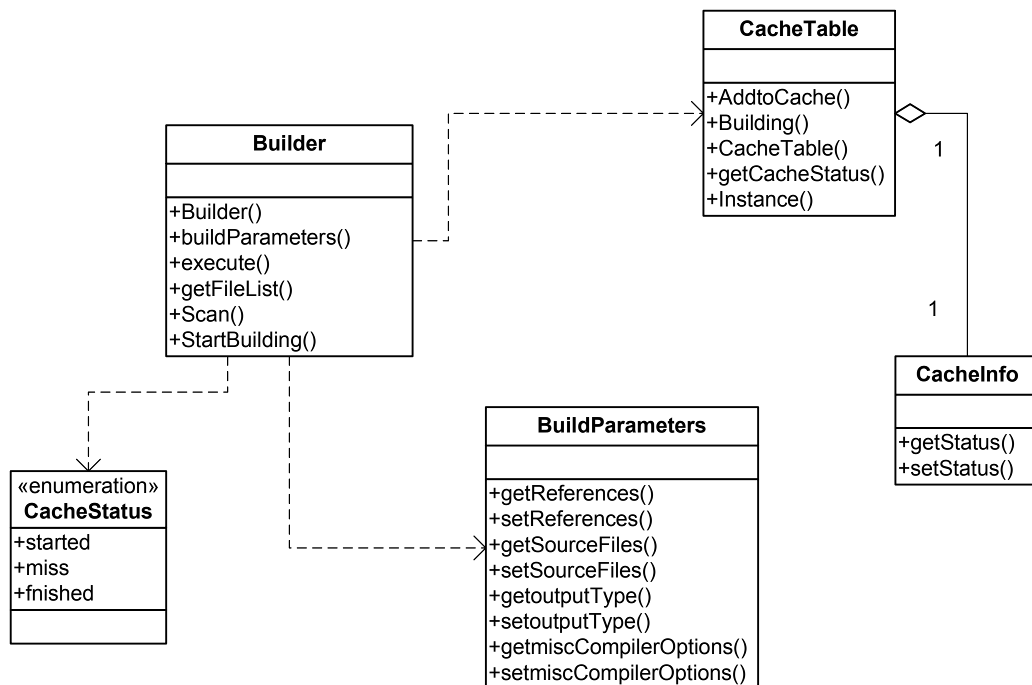
The Test Bed Server is composed of the Builder, Test Harness and an Extractor. The Test Bed Server accepts test requests from the RTClients. It uses the services of the Extractor to extract the components from the Repository. The Builder module compiles the source files extracted by the Extractor into compiled test libraries. The test libraries are then consumed by the Test Harness which loads and executes all the tests, produces a test report and reports it back to the RTClient. The Software Matrix is used for

communication between the Test Bed Server and other parts of the Repository Testbed System.

4.1.1 Builder:

The class diagram of the Builder module is shown below. The Builder module is responsible for building test libraries from the source files in the repository. Based on the platform in which the Testbed Server is executing, the Builder needs to use the appropriate compiler to build the source files. For example, the VC++ compiler can be used for win32 platform and g++ compiler can be used for the linux platform. We shall discuss the implementation details of the Builder in the following section.

Figure 4.2 Builder – Class Diagram



4.1.1.1 Builder - The builder class is the executive module and co-ordinates and uses the services of other helper classes in the build process. The builder accepts a path to the extracted files from the repository. It scans all the extracted items from the repository and using the build-specification in the item's metadata it compiles the source code into test libraries. The list of test libraries is then returned for testing by the Test Harness.

4.1.1.2 Build Parameters – As mentioned above, each item has a specification for its build in its meta-data. The meta-data contains the source files, output type of the item, supported compiler, miscellaneous compiler and linker options and references to other items, if any. This meta-data is used by the Builder to generate the final compiler command on the fly for each extracted item. This compiler command is then passed to the appropriate compiler and produces the test libraries.

The Builder also uses the services of the CacheTable and the CacheInfo classes to avoid rebuilding the same modules again by maintaining a Build Cache on the Testbed Server. An item in the repository is immutable once it is closed. To avoid rebuilding the item every time a test request is executed for a higher level component, we maintain a Cache Table which has references to all the items in the Build Cache. Thus we can copy the item from the build cache instead of rebuilding it again.

4.1.2 Test Harness:

The test harness provides a testing service for both Testbed Server and each RT Client. The harness contains a test aggregator, called tester, that loads a specified set of test Dynamic Link Libraries (DLLs). Each test library is required to support the ITest interface, so the aggregator creates instances of each test it loads (one per library), bound to the ITest interface, and executes the test by invoking a test() method declared by the interface and implemented by the class that implements the interface.

The test class derives from the ITest interface and aggregates a default TestVectorGenerator class and set of Logger classes. The TestVectorGenerator provides facilities for generating test inputs that classes derived from test use while testing. An example of one such facility reads a specified file line by line, supplying a new line each time its GenerateNext() function is called.

The FileLogger, ConsoleLogger, and MemoryLogger each derive from the ILogger interface and provide default facilities for writing test output to a file, stream, or saved to memory, for use later in the test.

We expect that a class, aTest, derived from test will be created for each module of source code to be tested, which must implement the test() function, defining specific testing operations. The designer of the derived aTest class will often provide classes derived from TestVectorGenerator and one or more of the loggers to provide the test inputs and logging needed for this specific test.

Note that both the Testbed Server and RT Clients have Test Harnesses, Test Configurers, and Builders. The RT Client will use these facilities to develop a team's tested source code before checking in the code's component to the Repository Server. The Testbed Server uses these facilities to run, perhaps many, concurrent tests on code combined from two or more of the teams working on the project.

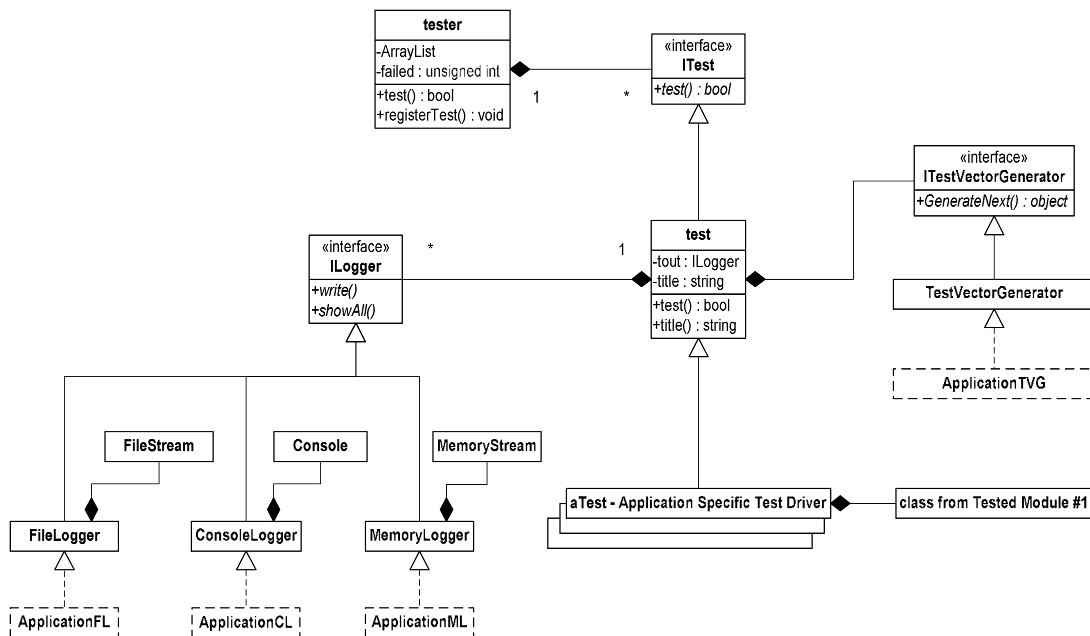


Figure 4.3 Test Harness Concept

Chapter 5 - Support for cross-platform builds using Software Matrix

The architecture of the Repository Testbed system supports development of applications which can run on various platforms. This is mainly possible because of the XML metadata used to describe each item in the repository. Using this XML metadata, we can control the platforms on which the item can be built and tested on. The Software Matrix framework is based on an XML based messaging protocol. Since XML is cross-platform, the Software Matrix can communicate with applications on any platform.

The cross-platform application is developed with the help of the Repository Testbed system. The source code for the various platforms of interest is checked-in to the Repository Server using the RTClient. We also setup a Testbed server on each platform of interest. Each item that is checked-in to the Repository can be built and tested using the Testbed server which is setup for the platform it has to be tested on.

5.1 Repository and Cross-Platform Development:

The cross-platform application consists of modules which run on a particular platform only as well as those which run on any platform. For example, it might consist of a C++ thread module which uses operating system specific features. Because of this, it will have different source files for different platforms. It might also consist of a C++ xmlParser module which uses standard C++ for all functionality needed to implement it. Because of

this it will have the same source file for different platforms. We will now discuss how the Repository organizes this information to support cross-platform development.

The cross-platform application to be developed is structured as items which are checked-in to the repository using the RTClient. The meta-data of each item is provided by the user while checking in the item into the Repository. The user specifies the following information to facilitate cross-platform development.

1) Name
2) Source Files
3) Documentation Files
4) References
5) Description
6) Keywords (optional)
7) Platform
8) Output Type
9) Miscellaneous Compiler Options (optional)
10) Miscellaneous Linker Options (optional)

5.1.1 Name and Platform:

Each item in the Repository is identified by a unique combination of name and platform. This name should be specified by the user while checking in the item into the Repository. Items for multiple platforms have the same name. For example, FileIO module for win32 and linux can have the same name as “FileIO” though they might refer to different source files. Each item is checked-in individually for every platform of interest.

5.1.2 Source Files and Documentation Files:

Each item is associated with header file(s), source file(s) and documentation file(s). The header files and source files for the item, particular to the platform to be checked-in are provided. For example, “FileIO” item for win32 will have the source file as FileIOwin32.cpp and “FileIO” item for linux will have the source file as FileIOLinux.cpp. Two items with same name but different platforms will point to the same source file if their implementation does not change across platforms.

5.1.3 References:

An item in the Repository can be dependent on another item for execution. In that case, we add a reference to the dependent item in the Repository. This helps in browsing the

source code tree according to dependencies and to extract all the code necessary for building automatically during testing.

5.1.4 Description and Keywords:

Each item in the Repository is given a description describing the purpose of the item in the Repository. The Keywords can be used to effectively locate items which conform to some specific criteria. For example, the keyword "ThreadSafe" can be added to all items which can operate in a multi-threaded environment.

5.1.5 Output Type, Compiler and Linker Options:

The output type of the item can be an executable, a dynamic link library or a shared library. Any extra compiler and linker options can be specified such as linking the wsock32.lib for sockets. These options are interpreted by the builder while building the item.

5.2 Sample Check-in using the Repository and RTClient:

We shall now demonstrate the check-in of items into the Repository. The first item to be checked in consists of different source files for different platforms and the second item to be checked in consists of the same source file across different platforms.

5.2.1 Check-in of FileIO Item:

The FileIO item uses operating system specific features for FileIO. So it will have different source files for different platforms. We now demonstrate the check-in of the item for windows and linux platforms.

The snapshot shows the checkin of the FileIO item for the win32 platform. Note that we have FileIO.h and FileIOWin32.cpp as source files and the user has entered the metadata about the item such as the item name, keywords, platform, output type etc.

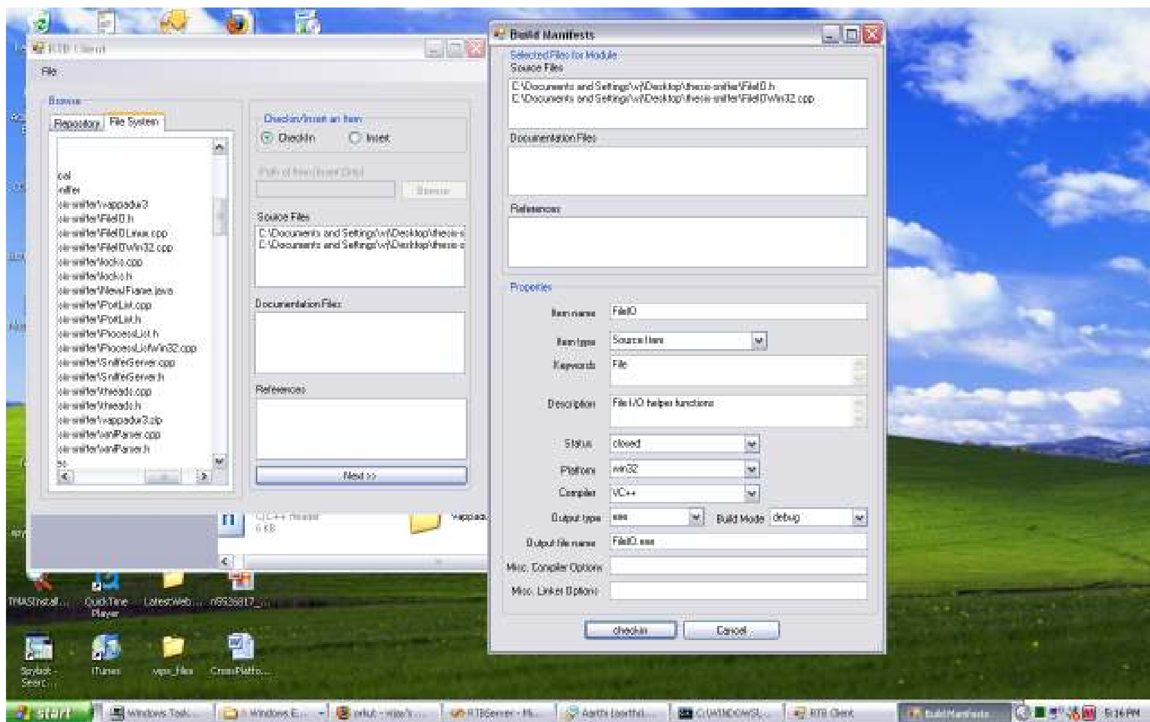


Figure 5.1 Check-in of FileIO item for win32 platform

The FileIO item was also written for the linux platform. From the Repository tab in the RTClient, we see that the FileIO for win32 platform was checked-in. Now in this case both the windows and the linux platform share the same header file. So when we check-in the item for the linux platform we make a reference to the header file which is already present in the Repository. In the "Build Manifests" form, we specify the implementation file as FileIOLinux.cpp and share the header file with the win32 item. We also specify metadata for the linux item and check it into the Repository.

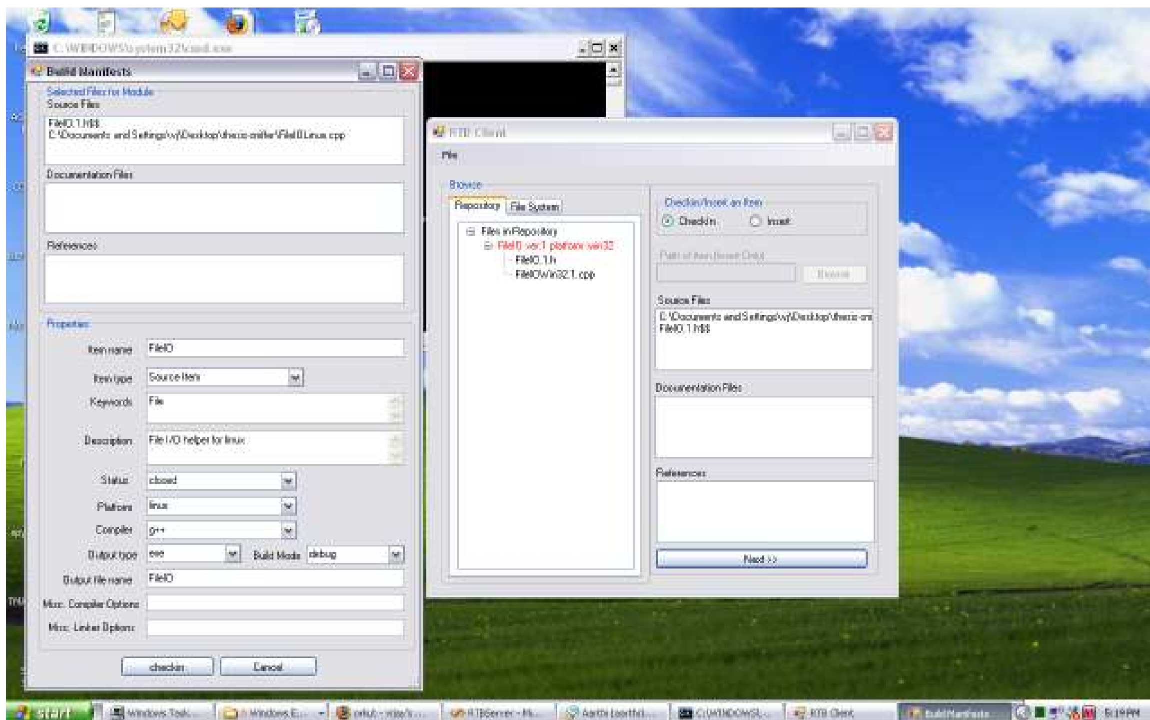


Figure 5.2 Check-in of FileIO item for linux platform

The following snapshot shows the item tree in the Repository Server through the RTBClient. The item colored in red is the win32 item and the item colored in blue is the linux item. We see that both of them share the same header file FileIO.1.h but have different implementation files.

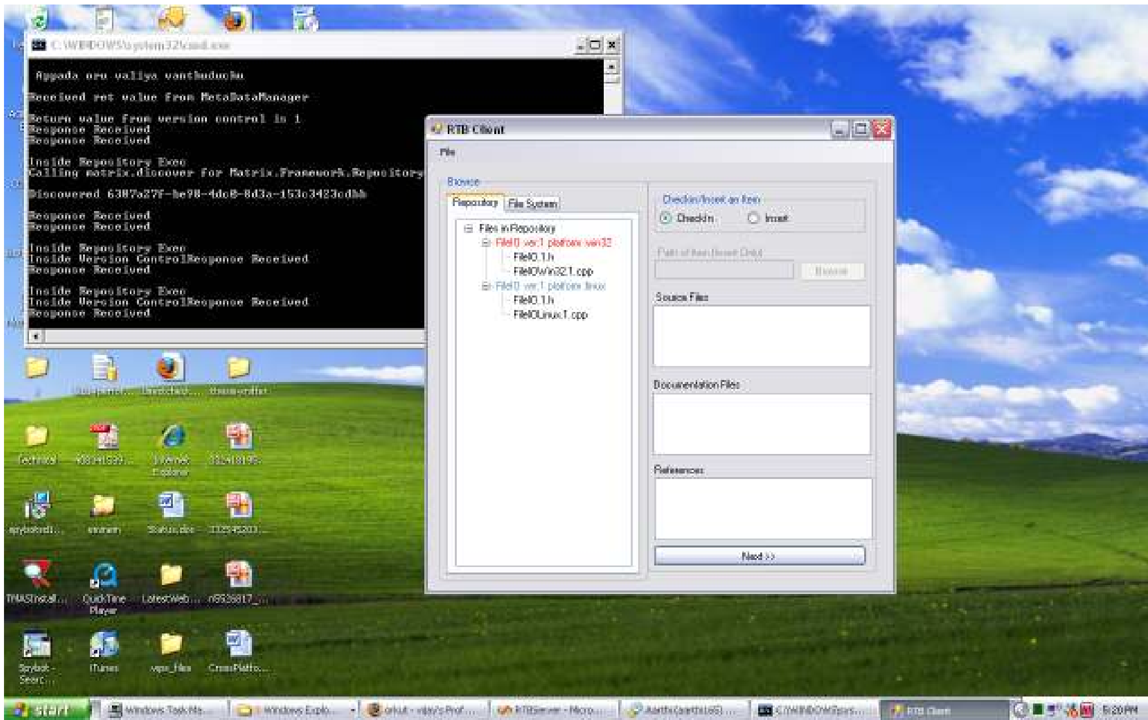


Figure 5.3 FileIO item in the Repository Tree View

The following XML data shows how the Repository stores the meta-data entered by the user to organize the items. Even though both the items refer to the same header file, the file is not replicated. The Repository just makes an extra reference to the file. Modification of the file by one item reflects in the other item since both of them refer to the same file.


```xml
<manifest>
<platform>
  <pinfo>
    <name>FileIO</name>
    <version>1</version>
    <description>File I/O helper functions</description>
    <compiler>VC++</compiler>
    <status>closed</status>
    <pname>win32</pname>
    <outputType>exe</outputType>
    <outputFilename>FileIO.exe</outputFilename>
    <checkedout>no</checkedout>
    <miscCompilerOptions />
    <miscLinkerOptions />
    <sourcefiles>FileIO.1.h</sourcefiles>
    <sourcefiles>FileIOWin32.1.cpp</sourcefiles>
    </pinfo>
  </platform>
<platform>
  <pinfo>
    <name>FileIO</name>
    <version>1</version>
    <description>File I/O helper for linux</description>
    <compiler>g++</compiler>
    <status>closed</status>
    <pname>linux</pname>
    <outputType>exe</outputType>
    <outputFilename>FileIO</outputFilename>
    <checkedout>no</checkedout>
    <miscCompilerOptions />
    <miscLinkerOptions />
    <sourcefiles>FileIO.1.h</sourcefiles>
    <sourcefiles>FileIOLinux.1.cpp</sourcefiles>
  </pinfo>
 </platform>
</manifest>
```

5.2.2 Check-in of xmlParser Item:

The xmlParser item was developed using standard C++ so it runs on both windows and linux without any modification. The xmlParser item therefore has the same source files across different platforms. We check-in the item individually for each platform as shown above. The following snapshot shows the xmlParser item in the Repository for both the windows platform and the linux platform. Note that both the items share the same source files xmlParser.1.h and xmlParser.2.cpp.

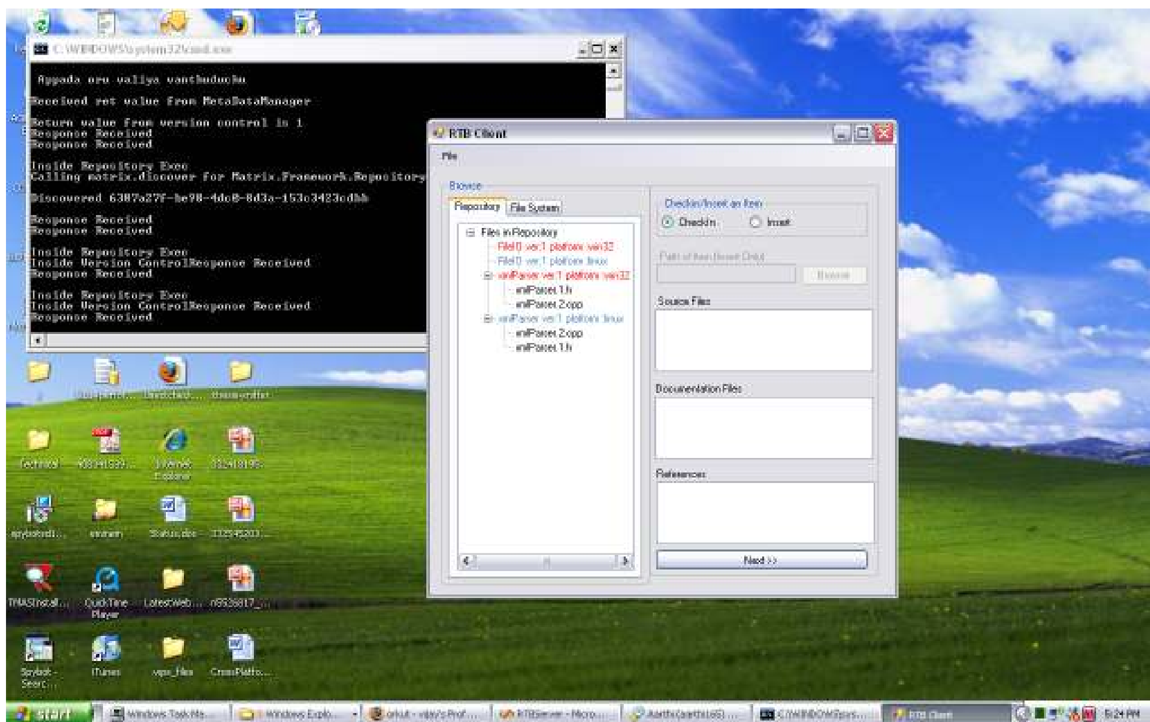


Figure 5.4 xmlParser item in the Repository Tree View

The following XML data shows how the Repository stores the meta-data entered by the user to organize the items. Even though both the items refer to the same source files, the files are not replicated. The Repository just makes an extra reference to the file. Modification of the files by one item reflects in the other item since both of them refer to the same file.


```
<manifest>
<platform>
  <pinfo>
    <name>xmlParser</name>
    <version>1</version>
    <description>helper class</description>
    <compiler>VC++</compiler>
    <status>closed</status>
    <pname>win32</pname>
    <outputType>exe</outputType>
    <outputFilename>xmlParser.exe</outputFilename>
    <checkedout>no</checkedout>
    <miscCompilerOptions />
    <miscLinkerOptions />
    <sourcefiles>xmlParser.1.h</sourcefiles>
    <sourcefiles>xmlParser.2.cpp</sourcefiles>
  </pinfo>
</platform>
<platform>
  <pinfo>
    <name>xmlParser</name>
    <version>1</version>
    <description>helper class</description>
    <compiler>g++</compiler>
    <status>closed</status>
    <pname>linux</pname>
    <outputType>exe</outputType>
    <outputFilename>xmlParser</outputFilename>
    <checkedout>no</checkedout>
    <miscCompilerOptions />
    <miscLinkerOptions />
    <sourcefiles>xmlParser.2.cpp</sourcefiles>
    <sourcefiles>xmlParser.1.h</sourcefiles>
  </pinfo>
</platform>
</manifest>
```

5.3 Setting up the Testbed Server:

The Testbed Server is setup for every platform in which the items are supposed to be built and tested. We have one testbed server for linux and one for windows. The Testbed Server listens for test requests from the RTClient. When the user requests a build of an item in the Repository using the RTClient, the client sends the request to the Testbed server configured for the platform in which the item is supposed to be built. The Testbed server issues extract requests to the Repository Server for the item and all its dependent items. Once the extract operation is complete, the Testbed consists of all the source files needed to build the item in the platform it is operating on. Now the builder is invoked to build the item.

5.3.1 Role of Software Matrix:

It should be noted that the Repository Server and the various Testbed Servers operate on different platforms. Furthermore, the RTClient was written with a different programming language (C#). The communication between the systems should be flexible so that it works both across programming languages and also across different platforms. So we need a platform agnostic communication mechanism. The Software Matrix is used to fulfill this requirement. All communication among the various components of the system is through the Software Matrix messaging protocol. This eliminates the necessity of using heavyweight protocols such as SOAP for cross-platform communication. The cross-platform, language independent nature of XML messaging helps us to tie heterogeneous systems written with different programming languages together.

Chapter 6: Demonstration – Design, Implementation, and Results

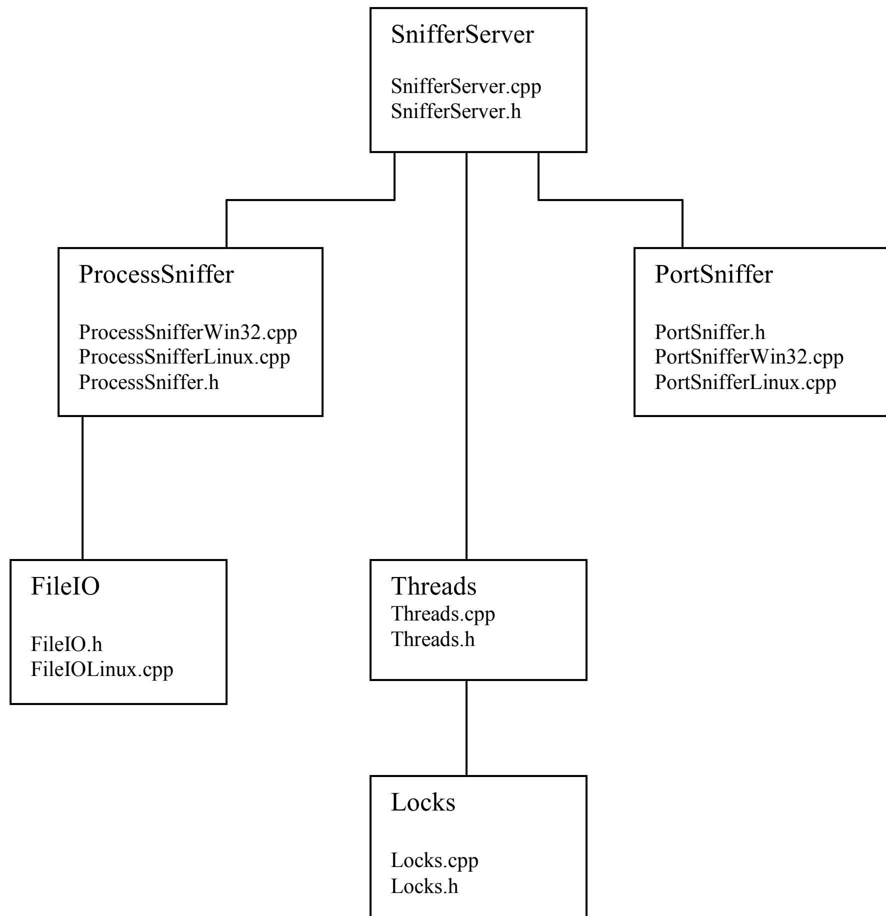
In this chapter we shall develop a cross-platform application using the Repository Testbed system and demonstrate building of the application in Windows and Linux platforms. We setup a testbed server in both windows and linux systems and configure the RTClient to use them. The application to be developed is structured as items and checked-in to the Repository using the RTClient. The items are built in various platforms by issuing requests to the Testbed server from the RTClient.

6.1 Process and TCP Connection Explorer:

The application that we develop should be truly cross-platform. So we decided to develop an application which uses OS specific features so that parts of the application are different for different platforms. Since the application was developed using C++, most of the application can run on various platforms without modifications to the source code.

The application is a client-server application with the server written in C++ and the client written using Java [12]. We will be demonstrating the development of the server side (C++) using the Repository Testbed system. The server module consists of a Process Sniffer which list all the processes currently running in the system and a Port Sniffer which lists all the ports in the machine which are open. Since we need to use operating system specific features to implement this functionality, the application is truly cross-platform and contains different source files for different platforms.

Figure 6.1 Module Diagram of Sniffer Server



6.1.2 SnifferServer:

The Sniffer Server is the executive module which listens on a socket for XML messages from the Sniffer Client written using Java. For each request from the client, it gets the information about the list of processes from the Process Sniffer and the list of open ports from the Port Sniffer in the form of XML. It then returns this XML to the Sniffer Client using the socket which displays the list of processes and ports on its User Interface. Since it uses sockets with the Berkeley sockets interface, the source file remains the same for all platforms with minimal use of #ifdefs.

6.1.3 ProcessSniffer:

This module retrieves the list of all processes running in the server machine. The process of getting this information is different for windows and linux. We have the same header file for both platforms, but the implementation files differ for windows and linux. The windows version uses EnumProcess API in the psapi.dll to get the information about the running processes in the system. The Linux version contains the /proc filesystem and gets the list of running processes from the file system. So it makes use of the FileIO module for the linux platform.

6.1.4 PortSniffer:

This module retrieves the list of all ports which are open in the server machine. We use different methods to obtain this information for windows and linux. So they have different implementation files. In the case of linux, we simply try to connect to each port and determine if it is open or not. In the case of windows we get the tcp ports and the udp ports which are open using the GetTcpTable API.

6.1.5 Threads and Locks:

These modules use operating system specific features to achieve multi-threading and synchronization. The windows version uses the win32 API and the linux version uses the pthreads API. However we have the same implementation file for both platforms with the use of pre-processor directives (#ifdefs). These modules are used by the SnifferServer while spawning a thread for each request and for synchronizing the access to the console.

6.2 Windows Build:

When a test request is issued by the RTClient to build the win32 item of SnifferServer, it makes a request to the TestBed Server. The Testbed Server extracts the item and all its dependencies to the target machine and builds the item

It should be noted that the building of the item is done automatically without any user configuration. Using the meta-data in the Repository, the Testbed Server extracts all the dependent items needed for compilation and linking. The user also does not need to configure the order in which the items are to be built. It is also taken care of automatically using the meta-data in the Repository. The items which don't depend on anything are built first followed by those which are dependent on them.

Determining the order of build automatically also enables the Testbed Server to test in that order. The item which is not dependent on anything should be tested first followed by those which are dependent on it. This makes it easier to locate the components which need to be tested again in the case of a failure. . The following snapshot shows the building of the SnifferServer item in the Windows machine by the Testbed Server present on that machine.

C:\WINDOWS\system32\cmd.exe

```
Compiler Command: cl /D "WIN32" /EHsc   PortList.cpp /c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

PortList.cpp

Adding to Cache:Finished

Adding to Cache: Started
Compiler Command: cl /D "WIN32" /EHsc   /I "C:/Repository/Extract\PortLister"  P
/c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ProcessListWin32.cpp

Adding to Cache:Finished

Adding to Cache: Started
Compiler Command: cl /D "WIN32" /EHsc   /I "C:/Repository/Extract\PortLister"  /
ract\ProcessLister"  locks.cpp /c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

locks.cpp

Adding to Cache:Finished

Adding to Cache: Started
Compiler Command: cl /D "WIN32" /EHsc   /I "C:/Repository/Extract\Lock"  /I "C:/
ortLister"  /I "C:/Repository/Extract\ProcessLister"  threads.cpp /c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

threads.cpp

Adding to Cache:Finished

Adding to Cache: Started
Compiler Command: cl /D "WIN32" /EHsc   /I "C:/Repository/Extract\Lock"  /I "C:/
ortLister"  /I "C:/Repository/Extract\ProcessLister"  /I "C:/Repository/Extract\
ver.cpp /c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

SnifferServer.cpp

Adding to Cache:Finished
Linker Command: link /OUT:SnifferServer.exe /INCREMENTAL /NOLOGO /DEBUG /SUBSYST
X86                                               /ERRORREPORT:PROMPT kernel32.l
lib winspool.lib                                                 comdlg32.lib a
lib ole32.lib oleaut32.lib uuid.lib
2.lib wsock32.lib  "C:/Repository/Extract\Lock\locks.obj"   "C:/Repository/Extra
st.obj"   "C:/Repository/Extract\ProcessLister\ProcessListWin32.obj"   "C:/Repos
rServer\SnifferServer.obj"   "C:/Repository/Extract\Threads\threads.obj"
Press any key to continue . . . _
```

Figure 6.2 Testbed Server Building the Sniffer Server Component in Windows

6.3 Sniffer Application on Windows:

The snapshot shows the execution of the Sniffer Server on the win32 platform using the Sniffer Client written using Java. The client and server communicate using XML messaging as shown in the server console. The Sniffer Client queries the Sniffer Server every 5 seconds for the list of running processes and open ports and lists them on the user interface. We developed the Sniffer Client using the NetBeans IDE. We did not use the Repository Testbed System for the development of the Java Client.

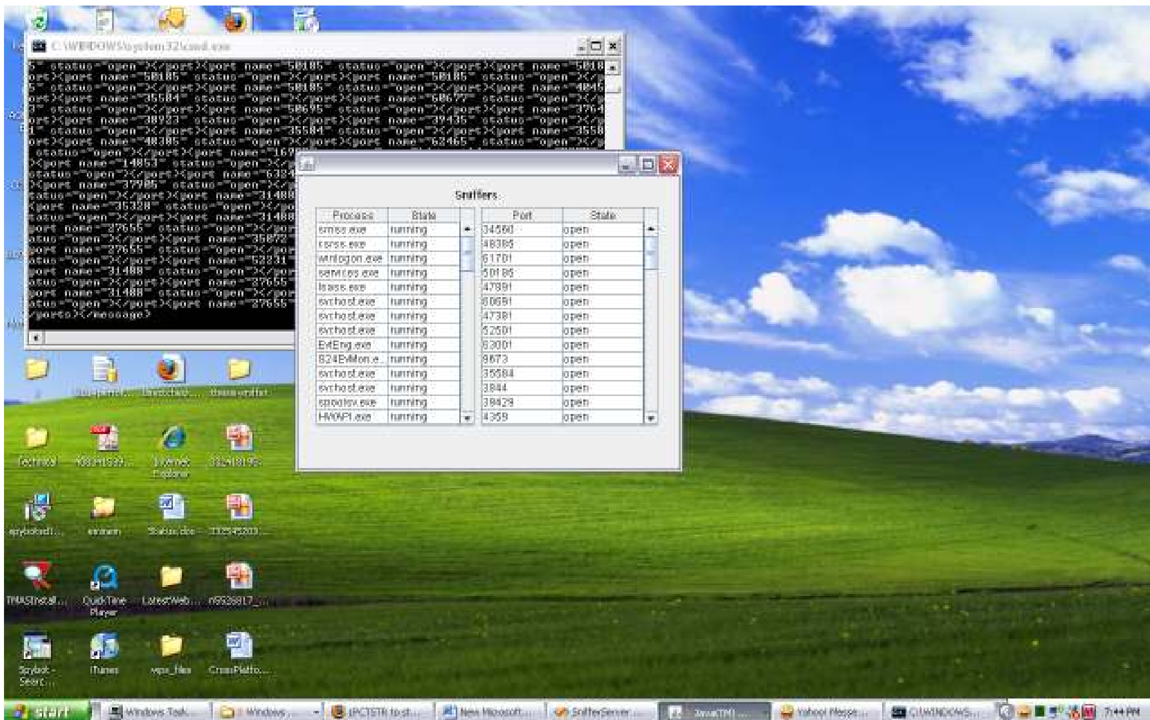


Figure 6.3 Execution of Sniffer Application on Windows Platform

6.4 Linux Build:

Similar to the windows build, when a test request is sent for the item which executes on linux platform, the testbed server for that platform extracts the item and all its dependencies and build the item in linux. It should be noted that there is no configuration needed at this point of time to perform the build. The following snapshot shows the building of the Sniffer Server item and all the items it depends on using a Testbed Server in Red Hat Enterprise Linux.

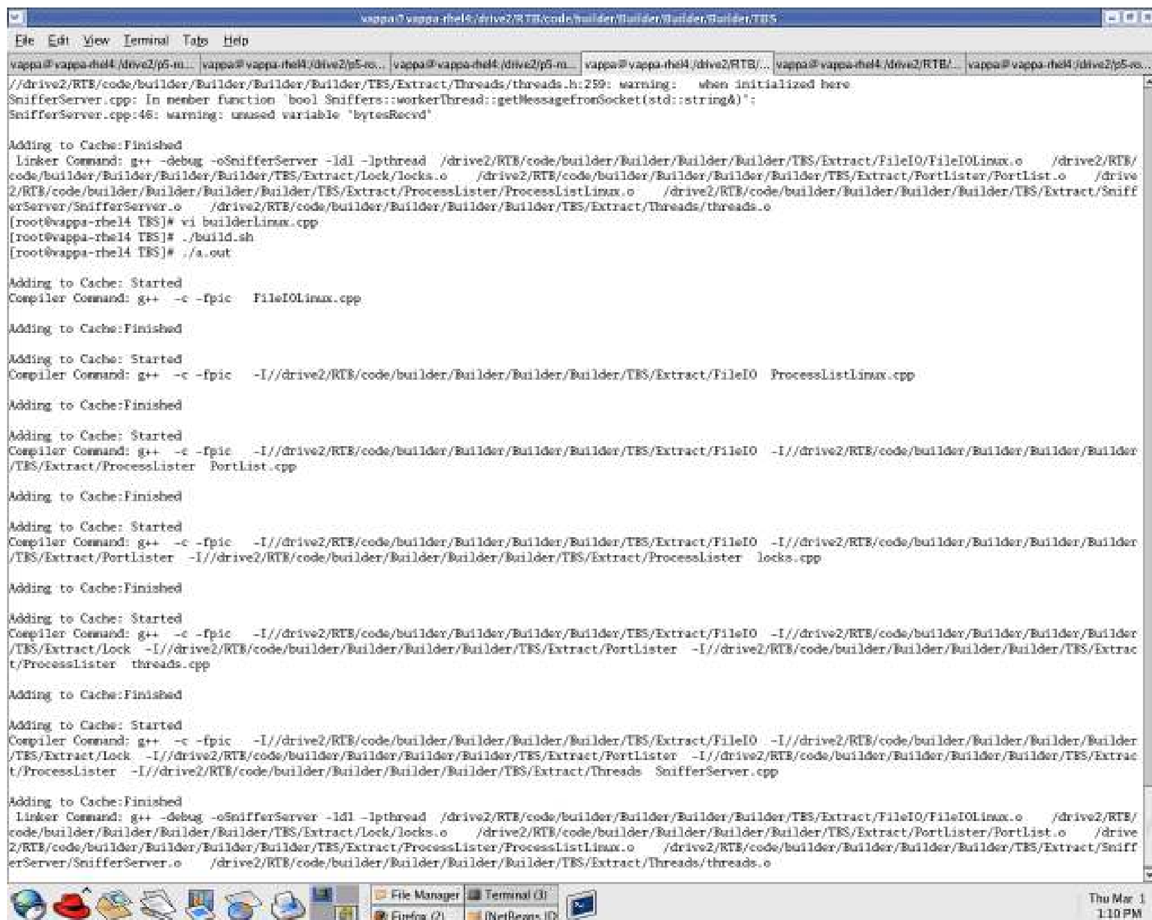


Figure 6.4 TestBed Server building the Sniffer Server component in Linux

6.5 Sniffer Application on Linux:

The snapshot shows the execution of the Sniffer Server on the RHEL (Red Hat Enterprise Linux) platform using the Sniffer Client written using Java. The client and server communicate using XML messaging as shown in the server console. The Sniffer Client queries the Sniffer Server every 5 seconds for the list of running processes and open ports and lists them on the user interface. We developed the Sniffer Client using the NetBeans IDE for Linux. We did not use the Repository Testbed System for the development of the Java Client.

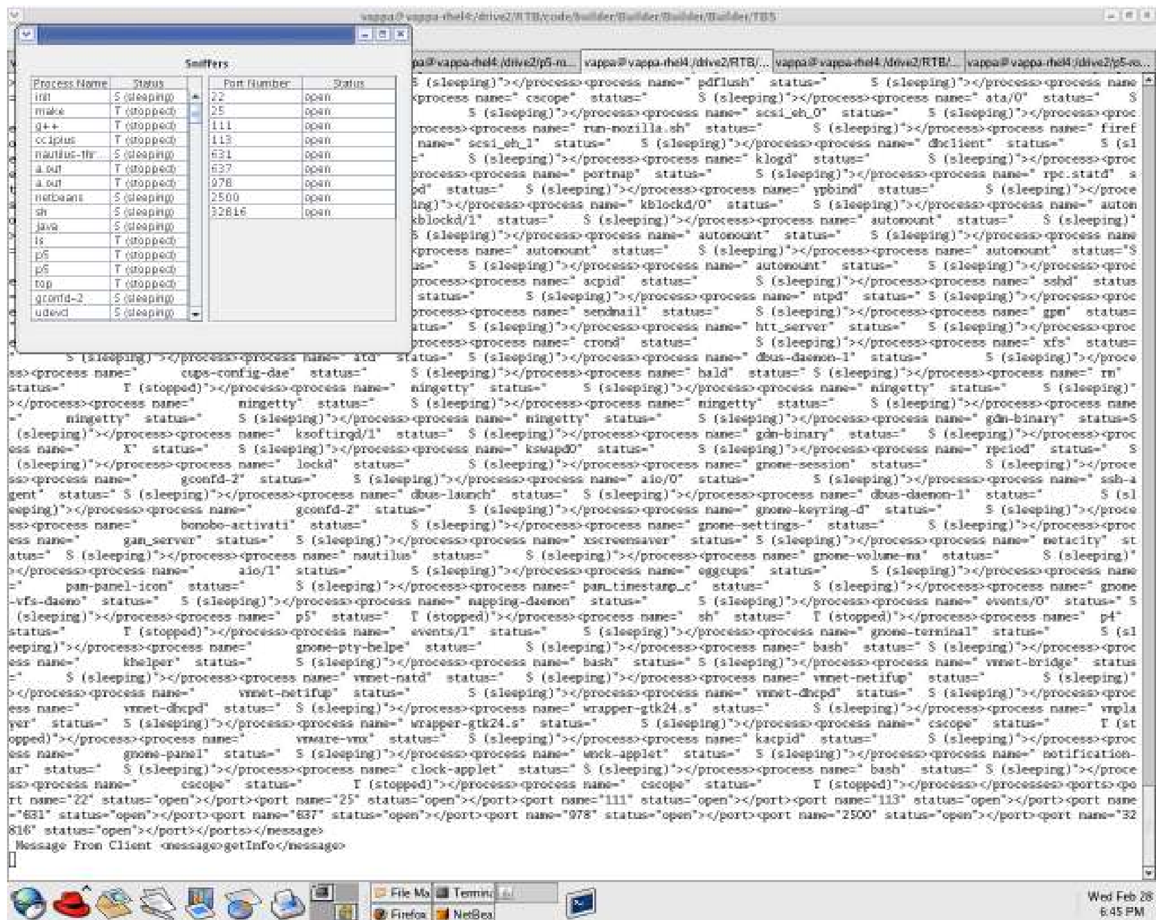


Figure 6.5 Execution of Sniffer Application on RHEL

Chapter 7 - CONCLUSIONS

This chapter summarizes the goals of the Repository Testbed System, our contributions towards simplifying cross platform development, and the future work.

7.1 Addressing Cross-Platform Development:

Cross-Platform development has been an important part of the software industry because of the existence of more than one platform the application has to run on. Any application developed has to be tested on all platforms they need to be deployed on. In this thesis our goal is to support cross-platform development through a framework which will help to reduce development and maintenance costs. We also aim to facilitate building and testing of the applications developed with no configuration required every time the application is built. We also aim to show how a light weight framework can be used for cross-platform development instead of using heavy-weight protocols such as SOAP.

7.2 Contributions of this Thesis:

Our contributions in this thesis are mainly on supporting cross-platform development.

1. We proposed a framework which supports cross-platform development using a source-code control and a test bed. We had a single code-base to store the source code files for all the platforms called the Repository Server and we had one Test bed Server for all the platforms the applications need to be deployed on. We also developed an RTClient which is the integrated user interface to control both the Repository and the Testbed Server.

2. We developed the framework supporting cross-platform development using the Software Matrix, a framework supporting Software salvage. So we demonstrated that the Software Matrix can be used in the development of large applications in which performance was crucial. We also demonstrated how the Software Matrix can be used to tie different platforms together using its cross-platform messaging protocol. The RTClient and the Repository Server were written using different programming languages but they work seamlessly using the Software Matrix. We also improved the performance of the Software Matrix by bypassing the Mediator and sending the messages directly to the intended cells.
3. We developed a sample Sniffer application which was truly cross-platform using the Repository Testbed system. The Sniffer application had parts of it working on any platform and the other parts working only on a specific platform. We demonstrated cross-platform building of the Sniffer application by extracting files needed for the target platform and demonstrated that cross-platform development works with no configuration at the time of building.

7.3 Future Work:

During the process of working on the Repository Testbed System, directions for future research and development were identified. Pursuing these areas of improvement will improve the Software Matrix to develop distributed applications with high tolerance and efficiency.

7.3.1 Distributed Software Matrix:

The Self Healing System developed by Anirudha Krishna [6] uses a centralized server model to provide service discovery and fault tolerance using a centralized addressing server and a repository to host the cells respectively. Since it is based on a centralized server model, it carries all the disadvantages of that model

7.3.1.1 Disadvantages:

- Requires use of high-performance specialized servers for hosting the Addressing Server and the Repository to handle large number of requests from the cells.
- Failure of the Addressing Server and Repository brings down the whole system.
- Additional cost of updating multiple servers will affect the speed of operation of the system as a whole.
- Cost of Configuration of the Addressing Server and the Repository Server

7.4 Peer-to-Peer Software Matrix:

We intend to develop a peer-to-peer lookup service for Software Matrix using Chord : A peer-to-peer Lookup Service for Internet Applications. [7] Chord is an implementation of a distributed hash table. All the nodes in the Chord network are arranged in a modulo ring 2^m where m is the number of bits in the hash function used to hash the IP Address of the nodes. The keys are distributed evenly over the network using a consistent hashing algorithm like SHA1. The keys are hashed and stored in the nodes whose id is greater than or equal to the key id in the modulo ring. We intend to implement the service discovery for the distributed software matrix using the distributed hash table. All nodes add the capabilities of their cells into the distributed hash table on joining the network and as well as when new cells are loaded into the matrices in the different nodes. Each node in the distributed hash table maintains information about a maximum of m nodes. The number of messages required for a lookup in the hashtable in O(log N) where N is the number of nodes in the network and requires O(log 2 N) messages for joining the network. The proofs of these can be found in Chord [7].

7.5 Service Discovery

When a cell queries the location of a cell for a particular service, the mediator first searches the local matrix for the cells. If no cells provide the capability, the request is forwarded to the chord cell which provides service discovery through the distributed hash table. The chord cell searches the distributed hash table for the location of the cell providing the service. Then the cell can send the message directly to the destination cell providing the service. On unloading a cell from the Matrix, the entry is deleted from the distributed hash table.

7.5.1 Fault Tolerance:

The fault tolerance for service discovery is provided by replicating the keys in the r successors, where r is the fault tolerant factor. Failure of any of the nodes results in the lookup in the next successor with the replicated key. Hence for the chord network to fail lookup there should be a simultaneous failure of r nodes. Hence there is no centralized failure. It requires failure of all the nodes to bring down the Matrix since there is no centralized point of failure.

Similarly, self healing can be provided by replicating the cells of the Matrix in several nodes. Whenever a failure of a node happens, the lookup will simply yield the address of a different node hosting the same replicated cell.

7.6 Load Balancing:

A peer-to-peer distributed system is one in which all the nodes of the network have equal importance. In this case, every node stores part of the addressing information and the repository information. Hence there is no load on a single server as such. This model hence provides load balancing without using high-performance servers.

References:

[1] The Software Matrix: An architecture for Software Salvage, Master's Thesis by Riddhiman Ghosh, 2004

[2] Why The Future Of Science Must Be In Free Software, Alessio Damato, 26th June 2005

[3] ACCU Spring 2003 Conference: Multiplatform Software Development, Beman Dewis

[4] http://www.boost.org/tools/build/v2/index.html Boost.Build System v2

[5] http://www.perforce.com/jam/jam.html Jam – Open Source Tool replacing make

[6] GNU AutoConf, Automake and Libtool, by Gary V. Vaughan, Ben Elliston, Tom Tromey and Ian Lance Taylor.

[7] http://en.wikipedia.org/wiki/Java_Virtual_Machine

[8] http://www-128.ibm.com/developerworks/webservices/library/ws-port/

[9] Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications.

[10] Self Healing Systems using the Software Matrix by Anirudh Krishna, Master's Thesis, 2005

[11] SOAP Messaging Framework http://www.w3.org/TR/soap12-part1/

[12] Java Programming Language http://java.sun.com/

[13] Extensible Markup Language http://www.w3.org/TR/xml/

[14] The C++ Programming Language http://www.research.att.com/~bs/C++.html

[15] Distributed Computing http://en.wikipedia.org/wiki/Distributed_computing

[16] Cross-platform http://en.wikipedia.org/wiki/Cross-Platform

[17] Linux Operating System http://www.sun.com/software/linux/

[18] Windows Operating System http://www.microsoft.com/windows/products/windowsxp/default.mspx