



DECEMBER 6, 2017


CSE-681 Software Modelling and Analysis

Instructor - Dr. Jim Fawcett

REMOTE BUILD SERVER - OCD

PROJECT 4#- OPERATIONAL CONCEPT DOCUMENT

RAMA TEJA REPAKA
SUID 264100460



Contents

1. Executive Summary	4
2. Introduction.....	7
2.1 Concept and Key Architectural Ideas.....	7
2.2 Required Functionalities.....	8
2.2.1 Functional Requirements.....	8
2.2.2 Non-Functional Requirements.....	9
2.2.3 High Level Architecture Design.....	10
2.3 Organizing Principles	10
3. Uses.....	10
3.1 Users and Impact on Design.....	11
3.1.1 Course Instructor and Teaching Assistants.....	11
3.1.2 Developers.....	11
3.1.3 QA's	12
3.1.4 Manager.....	12
3.2 Possible Extensions.....	12
3.2.1 Authentication.....	12
3.2.2 Application Scope logging.....	12
3.2.3 Visualize threads.....	12
3.2.4 Process parallelization.....	12
4. Module Structure.....	13
4.1 Mock Repository.....	14
4.2 File Manager.....	14
4.3 Blocking Queue.....	15
4.4 Build Config.....	15
4.5 Test Harness.....	15
4.6 MpCommService.....	15
4.7 IMPCommService.....	16
4.8 Logger.....	16
4.9 ILogger.....	16
4.10 Build Server Messages.....	16
4.11 Test Harness messages.....	16
4.12 Build Request Parser.....	16
4.13 Child Builder.....	16
4.14 Mother Builder.....	17
4.15 Mock Client.....	17
4.16 App Domain Manager.....	18
5. Key Application Activities.....	19
5.1 General activities at Mock client.....	20
5.1.1 Build Request Generation.....	20
5.1.2 Handling generated Build Request.....	21

5.2 General activities at Mock Repository.....	24
5.2.1 Validating build request file.....	24
5.2.2 Processing Build request.....	25
5.2.3 Processing file Requests.....	25
5.3 General activities at Build server (Mother builder).....	26
5.3.1 Process Pool Creation.....	25
5.3.2 Handling Blocking Queue Messages.....	25
5.4 General activities at Child Builder.....	27
5.4.1 Handling Build Request messages.....	27
5.4.2 Handling File request messages.....	27
5.5 General Activities at Test harness.....	27
5.5.1 Parsing of test Request.....	28
5.5.2 Loading Dynamic link libraries.....	28
5.5.3 Deletion of old Dynamic link libraries.....	28
6. Class diagrams.....	32
6.1 Mock client.....	32
6.2 Mock Repository.....	33
6.3 Process pool.....	34
6.4 Mock Test Harness.....	35
7. Views.....	36
7.1 Generate Build Requests.....	36
7.2 Handle Build Requests.....	37
7.3 View Logs.....	38
8. Critical Issues.....	41
8.1 Ease of Use.....	41
8.2 Building Different Sources.....	41
8.3 Throughput.....	41
8.4 Malformed Code.....	41
8.5 Exceptions.....	42
8.6 Performance.....	42
8.7 Versioning Control.....	42
8.8 Inconsistency in Input Request.....	42
8.9 Requirements Demonstration.....	42
8.10 Managing End Point information.....	43
8.11 Client crashes after sending a request.....	43
8.12 Embed application logic in Client.....	43
8.13 Message passing using Blocking queue.....	44
8.14 Defining a Single Message Structure.....	44
8.15 Testing both C# and C++ code.....	44
8.16 Client sends same Request Multiple times.....	44
9. Deficiencies.....	45

9.1 Authorization and Authentication.....	45.
9.2 Building Only C# source files.....	45
9.3 Implemented message dispatcher only for Mock client.....	45
9.4 Not Added Build Request declarations in Comm Message.....	45.
9.5 Bad Version management in mock repository.....	45
9.6 Local File storage in Mock repository.....	45
10. Conclusion.....	45
11. References.....	46
12 . List of some important diagrams	
12.1 Figure 1: High level architecture diagram for Remote Build Server.....	10
12.2 Figure 2: Package diagram for Entire Application.....	13
12.3 Figure 3: package diagram for Build server.....	15
12.4 Figure 4: Process pool.....	17
12.5 Figure 5: Remote build Server message flow.....	20
12.6 Figure 6: Activity diagram for Mock client.....	21
12.7 Figure 9: Sample Test log	23
12.8 Figure 10: Activity diagram for Mock Repository.....	24
12.9 Figure 11: Activity diagram for Build server.....	26
12.10 Figure 12: Activity diagram for Child Builder.....	27
12.11 Figure 13: Activity diagram for Mock Test Harness	29
12.12 Figure 14: Message dispatcher diagram	31
12.13 Figure 15: class diagram for Mock client.....	32
12.14 Figure 16: class diagram for Mock repository.....	33
12.15 Figure 17: class diagram for Build Server.....	34
12.16 Figure 18 : class diagram for test harness.....	35

1. Executive Summary

Continuous integration of big software systems is one of the most common challenge that we face today in real world. Big software systems contain thousands of packages and perhaps several million lines of code. To successfully implement big software systems, we need to partition code into relatively small parts and thoroughly test each of the parts before inserting them into the software baseline. As new parts are added to the baseline and as we make changes to fix latent errors or performance problems we will re-run test sequences for those parts and, perhaps, for the entire baseline. As in the real-world changes occur frequently through testing of complete software baseline becomes overhead. So best way to intensify this testing practical is to automate the process.

Remote Build Server supports continuous integration that is when new code is created for the system we build it we test it in the context of other code which calls it and as soon as the test passes we check in the code and it becomes part of the current baseline. We make use of dedicated servers like Mock-Client, Mock-Repository, Build Server, Mock-Test-Harness. Each of them will be having unique responsibilities which are necessary to support continuous integration of software baseline. Each of the dedicated server's responsibilities will be known shortly as we go through the document.

This development will create a Build Server, capable of building C#, using a process pool to conduct multiple builds in parallel. The implementation is accomplished in three stages.

The first, Project #2, implements a local Build Server that communicates with a mock Repository, mock Client, and mock Test Harness, all residing in the same process. Its purpose is to allow the developer to decide how to implement the core Builder functionality, without the distractions of a communication channel and process pool. The second, Project #3, develops prototypes for a message-passing communication channel, a process pool, that uses the channel to communicate between child and parent Builders, and a WPF client that supports creation of build request messages. Finally, the third stage, Project #4, completes the build server, which communicates with mock Repository, mock Client, and mock Test Harness, to thoroughly demonstrate Build Server Operation.

The final product consists of a relatively small number of packages. For most packages there already exists prototype code that show how the parts can be built. For this reason, there is very little risk associated with the Build Server development.

The intended users of this project are Developer, Teaching Assistants and Instructor. The developer will use this document for guidance and implementation in later phases. The TA's and instructor shall use this to check whether all requirements are met all critical issues related to the project are addressed.

Below are some of the critical issues associated with this project and solutions are discussed in detail in further sections.

User-Interface:

The Remote Build Server should be easy to use and thus providing a simple easy understandable user interface is crucial.

Building Different Source Files:

Building different types of source code files like C++, Java, C # etc. If we use MS Build library then it should be able to build only C# programs but not like Java, C++. So, we need to create our own build infrastructure that uses the compilers and tool chains for the target platform

Heavy-workloads:

The Build Server may have heavy workloads just before the releases, so we should make the through put for building code as high as is reasonably possible. To handle this critical issue, we need to implement the process pool.

Malformed code:

There may be Malformed code which overflow the process stack this should not stop the build server. It should create a new process replacement and reports the build errors to the repository.

Performance:

The Build Server will be one of the busiest servers in the project. One of the most critical issues is its performance. The time required to complete all the build tasks increases if there are more build requests and more source code files coming from the repository. As a result, there will be overhead on entire system because Mock test harness also takes lot of time to test and prepare reports and send back to the repository and it also should notify the author. This can be partially managed by multitasking and manage of resources.

Versioning Control:

Proper versioning control system should exist repository because there may be chances of build failure at such crucial times versioning control helps to resolve needs and make systems work with the available latest versions of source code.

Runtime exceptions

There may be also chance of runtime exceptions and federation servers also may crash sometimes

Client crashes:

Client crashes after sending a request, there can be a scenario where a client can crash after it has sent the request but not received response.

Embedding application logic in client:

Embed application logic in Client GUI, the temptation of dumping all the code behind each dialog window control to the respective button handler. This is a malpractice and even though it might sound easier to do, it usually causes the system to slow down considerably.

Requirements Demonstration:

Demonstrating to the graders that all the requirements have been met without having them to manually enter the data.

Inconsistencies in Build Request:

May be due to inconsistency in the Input build request there may be also chance of runtime exceptions which may crash the entire application.

Message passing using Blocking queue:

Situations when there are an overwhelming number of messages in the blocking queue, more than what the Server has been designed to handle.

Defining a single Message Structure

In the federation we need to define a single message structure that works for all the messages

Testing C++ and C# code

Testing different source code is one of main issue. We need to trap exceptions on loading native code libraries in the C# Mock Test Harness and direct to C++ Mock Test Harness.

Managing End point information

Managing End Point information for Mock Repository, Build Server and child builders, and Mock Test Harness is difficult. So, store End point information in XML file and load at start up.

Client sends same Request Multiple times

When users tend to send the same request multiple times in the test harness there may be exceptions because we need to delete the previous dll's send by the child builders. If we don't use the concept of app domain then we will face exceptions because dll's are still loaded in to the current app domain unless unloaded cant delete them. This problem can be solved by App domain.

This document further provides discussions on interactions between modules of the application, flow of activities to achieve tasks with support of diagrams, use cases of this application, Solutions to the critical issues associated with the application

2. Introduction

CONCEPT AND KEY ARCHITECTURAL IDEAS:

For supporting continuous integration that is when a new code is created for the system, we build and test it in the context of other code which it calls and as soon as the test case passes we check in the code and it becomes part of the current baseline, we make use of

Mock-Client which have graphical user interface that provide mechanisms to get file lists from the Repository, and select files for packaging into a test library¹, adding to a build request structure. It also provides the capability of repeating that process to add other test libraries to the build request structure. It is implemented using Windows Presentation Foundation (WPF). It uses Windows Communication Foundation (WCF) to communicate with other servers. Mock Client which also has the interface to submit the code and tests requests to the repository and later it also has the capability to view the results stored in the repository.

Mock-Repository that holds all code and documents for the current baseline, along with their dependency relationships. It supports client browsing to find files to build, builds an XML build request string and sends that and the cited files to the Build Server. It also holds build logs, test results sent by Build Server and Mock-Test-Harness. It uses Windows Communication Foundation (WCF) to communicate with other servers. It may also cache build images.

Most Important part that is, Build Server based on build requests and code sent from the Repository, the Build Server builds test libraries for submission to the Test Harness. Build server internally use a process pool to support heavy workloads which is necessary during customer demos and releases. Process pool lets you start limited set of processes spawned at startup. Then build server provides a queue of build requests, and each pooled process retrieves a request, processes it, sends the build log and, if successful, libraries to the test harness, then retrieves another request. During crashes simply creates a new process replacement, and reports the build error to the Mock Repository. The communication between pool process and Build Server's Mother builder is implemented using Windows Communication Foundation (WCF). It uses Windows Communication Foundation (WCF) to communicate with other servers like Mock Test Harness and Mock Repository.

Mock-Test-Harness based on the test requests and libraries sent from the build server executes tests, logs results, and submits results to the Repository. It also notifies the author of the tests of the results. It uses Windows Communication Foundation (WCF) to communicate with other servers.

We can demonstrate we are meeting all the requirements to the Graders and TA's by making a logger to console as well. The logger would thus write status to console as well as log file which gives us in sight in to project execution.

REQUIRED FUNCTIONALITIES:

The Remote Build Server shall implement the following the functional and non-functional requirements.

Functional Requirements:

Following are some of the functionalities that will be implemented in the project

- 1) Message passing communication service should be implemented using Windows Communication Foundation.
- 2) Mock Client implemented using Windows Presentation Foundation (WPF) and message-passing communication shall provide mechanisms to get file lists from the Repository, and select files for packaging into a test library¹, adding to a build request structure. It shall provide the capability of repeating that process to add other test libraries to the build request structure. Mock Client should be able to request the repository to send a build request in its storage to the Build Server for build processing.

- 3) Mock Client should send build request structures to the repository for storage and transmission to the Build Server.
- 4) Repository server should support client browsing to find files to build, builds an XML build request string and sends that and the cited files to the Build Server. Proper versioning control must be implemented by Repository server.
- 5) Build Server should receive all the source code and the build requests sent by the repository. It includes a Process Pool component that creates a specified number of processes on command.
- 6) The Message passing Communication Service shall support accessing build requests by Pool Processes from the mother Builder process, sending and receiving build requests, and sending and receiving files.
- 7) Pool Processes should use message-passing communication to access messages from the mother Builder process. Each Pool Process shall attempt to build each library, cited in a retrieved build request, logging warnings and errors. If the build succeeds, shall send a test request and libraries to the Test Harness for execution, and shall send the build log to the repository.
- 8) The Mock Test Harness shall attempt to load each test library it receives and execute it. It shall submit the results of testing to the Repository.

Non-Functional Requirements

The Remote Build Server project shall be implement,

- i. C# using the facilities of the .Net Framework Class Library and Visual Studio 2017.
- ii. Mock Client shall be implemented in the form of Windows Presentation Foundation (WPF).
- iii. A test executive that clearly demonstrates meeting of all functional requirements.

High Level Architecture Design:

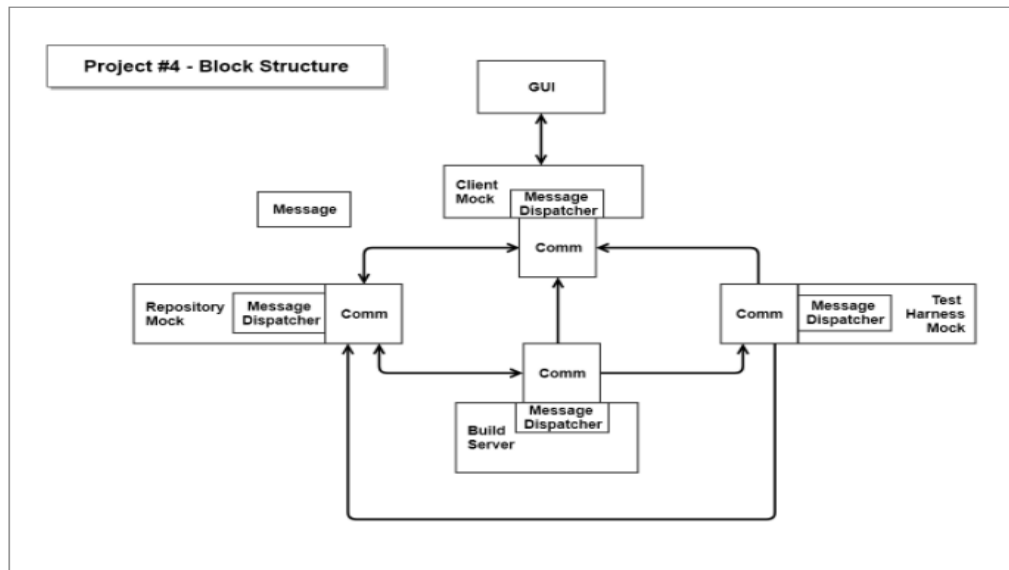


Figure 1: High level architecture diagram for Remote Build Server

The architecture diagram clearly shows how Mock Repository, Mock client, Mock Test harness and Build server are connected using Comm service which is used for communication built using Windows communication foundation(WCF).

ORGANISING PRINCIPLES

Most of the functionalities mentioned above will be achieved by splitting them into smaller tasks. Each task, or a group of related tasks, will be put into separate packages that make it easier to understand the operation of the overall application. The various packages that make up the application will include packages that control the program flow, test the various functionalities, process build requests, generate dynamic link libraries and log the results and execution.

3. USES

The current project mainly focuses on designing and implementing Remote build server and such as its primary uses will be generating the build request, processing the build requests, building libraries, notifying the client with the build logs, submitting libraries to test harness on successful builds, and commanding test harness to execute, and notifying the client with the test results.

USERS AND IMPACT ON DESIGN

Course Instructor and Teaching Assistants:

These users will focus on testing the implemented build server. This includes checking the various functionalities and evaluating the overall performance of the application.

Impact on Design:

Ideally testing should be designed such that it takes minimum input from the user, and displays the results of the build test effectively in a lucid concise manner.

Keeping this in mind program should include a Test package that will be responsible for demonstrating each of the requirements by running step-by-step through a series of tests. The results should be displayed concisely to facilitate easy comprehension.

Developers:

Developers are the primary users of this Remote Build Server. They work closely with the testers because they should not be wide gap between what developers has produced and what tester needs. They start the coding process and prepare unit test case documents to test their own modules.

Impact on Design:

Ease of use is the major design impact for the Remote Build Server. we need to provide user friendly graphical user interface to the developer along with the remote access facilities.

QA'S

The QA'S need to run several builds to demonstrate that the project meets the requirements. They may build the entire baseline and then later check the logs. As this is kind of automated process it saves lot of time. They also need to have the capability to understand issues and generate test cases. They mainly should have knowledge of execution methods.

Impact on Design:

Performance may be the major concern for them because there may be several build requests. So, build server should be fast enough to process all request and generate libraries.

Managers:

Manager is the main coordinator and work delegator in the system. They make sure that all resources are available to the team and they also prepare overall report.

They make sure that processes are carried out in strict order and check constantly whether meeting deadlines or not.

They also check on going project activities like check test case count, logs etc.

Impact on Design:

Managers need the test results data. Thus, the build server should provide excellent logging facilities with proper time and date stamp. It should also provide facilities to retrieve graphs showing the amount of load on the build server. These logs should be named, identify the test developer, the code tested including version, and should be time date stamped for each test execution.

POSSIBLE EXTENSIONS:

Authentication

Implementing the authorization and authentication on the top of Remote build Server so that multiple users can login, generate build requests and in the background process generated build requests, build libraries, notify the client with the build logs, submitting libraries to test harness on successful builds, and commanding test harness to execute, and notifying the client simultaneously with the test results.

Application Scope logging

Event log processing and analysis play a major role in applications ranging from security management, IT trouble shooting, to user behavior analysis. At the same time, as logs are found to be a valuable information source, log analysis tasks have become more sophisticated demanding both interactive. This application can be extended to support log data for mock client, mock repo along with build server, Mock test harness stores to support log processing and analysis, exploiting the scalability, reliability, and efficiency commonly found in Remote Build Server application

Visualize threads

Analyzing application running in complex multithreading environment is difficult because of the behavior of the threads. In this scenario, Remote build Server is much useful to visualize the threads behavior.

Process parallelization

In multi-threaded application there are few issues like

No security between threads

One thread can stomp on another thread's data.

If one thread blocks, all threads in task block.

Processes are a useful choice in such cases for parallel programming with workloads where tasks take significant computing power, memory or both. Because process allows for better throughput due to increased independence and isolation between the tasks vs. using one process with multiple threads. In this scenario, Remote build Server federation server mother builder is much useful to visualize the process parallelization behavior because it is implemented using process parallelization.

4. Module Structure:

Modern software applications scope is so large, so we can include all the functionalities in the single package or single source file. Instead application is developed as a collection of packages. The high-level responsibilities of application are broken down into smaller tasks that must be carried out to implement those functionalities. Logically a similar set of functionalities are grouped together what we call packages. Then various packages interact among themselves to provide fully functional software. The advantages of such approach are easier development, testing and debugging.

The package structures of entire application and Build server is shown in the below figures. The package diagrams are followed by a brief description of each individual package and its interactions with the other packages.

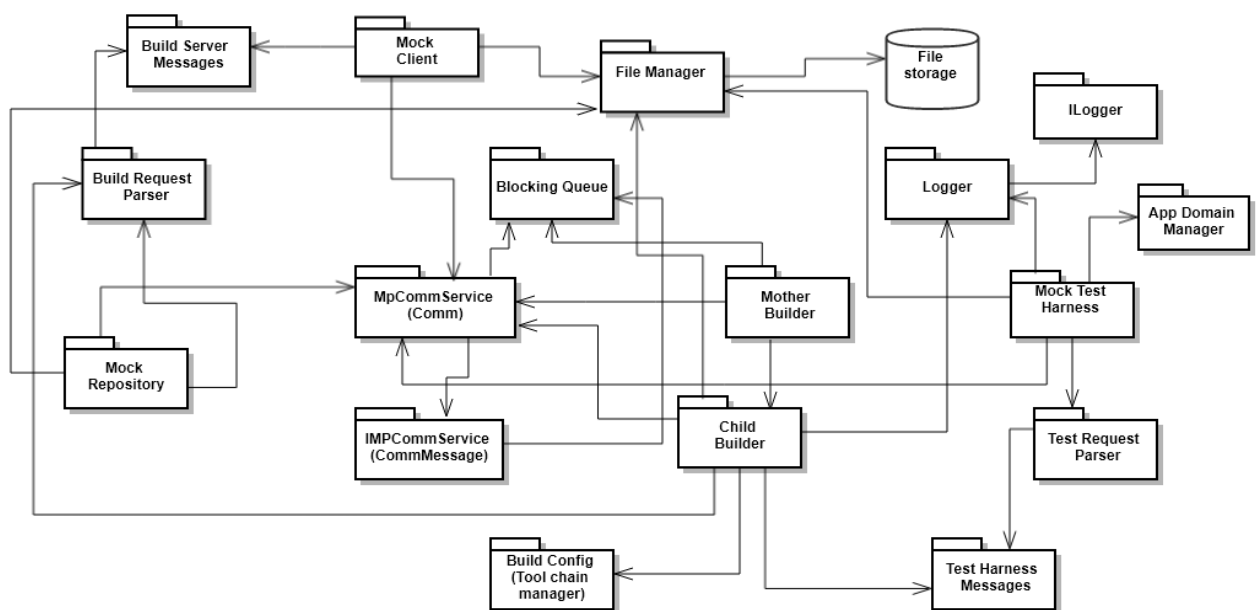


Figure 2: Package diagram for Entire Application

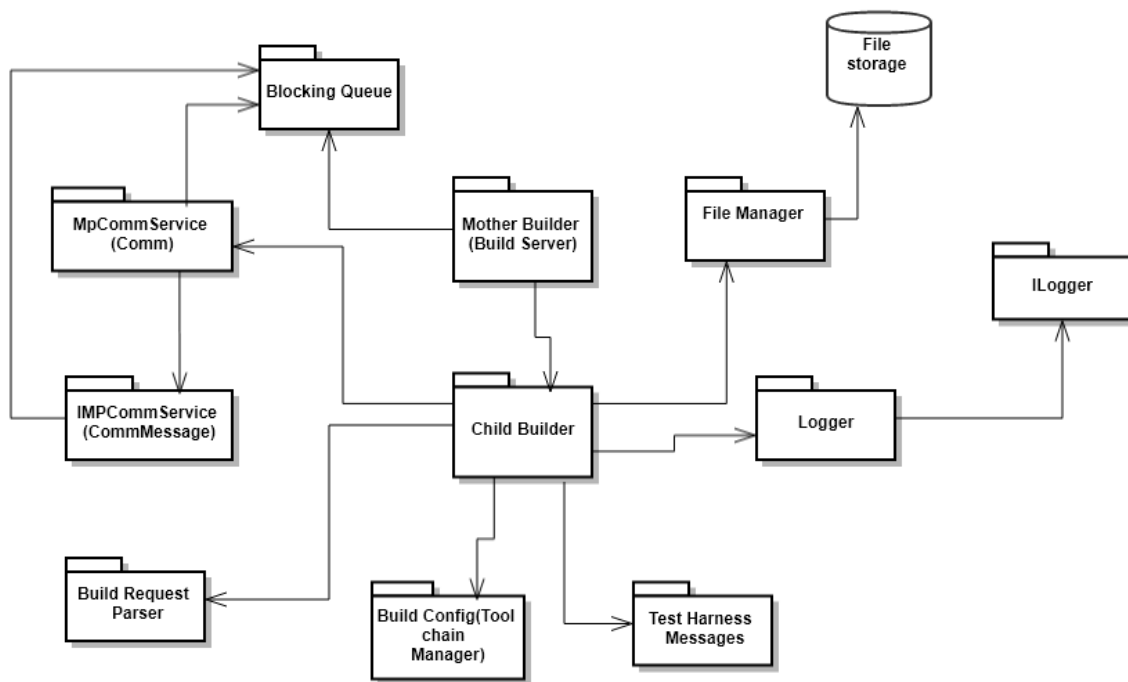


Figure 3: package diagram for Build server

Mock Repository:

Mock repository holds all code and documents for the current baseline, along with their dependency relationships. It uses message passing communication service for all communication with federation servers.

Some of the responsibilities of this package are:

validating the xml request sent by the client and storing them in the repository if it is a valid xml request and notify the client regarding the status of validation of xml request, Whenever client requests the repository to send a build request in its storage to the Build Server for build processing it serves the request, Storing all the build logs sent by the builders, Storing all the test logs sent by the Mock Test harness and Sending all the logs to the mock client when ever client requests for logs.

File Manager:

This package provides a class, File Manager, and object factory function, create (...). Its mission is to search the directory tree, rooted at a specified path, looking for files that match a set of specified patterns. It provides virtual functions file (...), directory (...), and done(..), which an application can override to provide application specific processing for the events: file found,

directory found, and finished processing. The package also provides interface hooks that serve the same purpose but allow multiple receivers for those events.

Blocking Queue:

This package implements a generic blocking queue and demonstrates communication between two threads using an instance of the queue. If the queue is empty when a reader attempts to deQ an item, the reader will block until the writing thread enQs an item. Thus, waiting is efficient. This blocking queue is implemented using a Monitor and lock, which is equivalent to using a condition variable with a lock.

This package is used by the Mother builder to insert all the build requests and ready requests into blocking queue later they will be dequeued and sent to the child builders based on corresponding ready requests. This is also used by message passing communication service to store all the messages that it received and to act in a first in first out manner.

Build Config:

This package is mainly responsible for maintaining all the configuration files and tool chains of target platforms which will be used while translating builder commands in to those needed by the specific tool chains. This build config infrastructure package will be used by the Child builders package where each builder will carry out the corresponding separate build process by using the concept of process parallelism.

Test Harness:

This package uses message passing communication service to receive all the libraries and test requests from the builders. It parses the test request and starts loading the libraries by using the concept of Application domain. It executes all the tests and sends all the test logs to Mock repository and notify the client regarding test status. It uses blocking queue which servers the builders in first in first out manner.

Mp Comm Service:

This package defines mainly three classes sender, receiver and COMM which implements using sender and receiver. Sender contains methods to connect to a channel, post a message or file and closing the channel. Similarly receiver contains methods to connect to a channel, post a message, receive the files through bytes and closing the channel. COMM has methods post message that sends message to receiver instance, get message that retrieves a message from sender instance and finally post file called by sender to transfer a file. It is used by all the federation servers for communication between them.

IMP Comm Service:

This package is a service interface for message passing communication service where it declares all the operations of communication service.

Logger:

The logger package will be used by child builder to log all the build activities, it is used by the mock test harness to log all the automated testing done by mock test harness. It is also used by Mock repo to save as logs in mock repository because test harness or builders sent logs in the form of comm messages. It contains build status, errors and warnings if any. It contains information regarding time stamps of build activities and test activities. federation servers uses communication service to send these logs.

Logger:

This package declares all the necessary operation for the logger package

Build Server Messages:

This package is mainly used for Serialization of Build Server Data structures. This can be used to create a build request. This package will be used by mock client to generate a build request. Child builders will be using build request parser which in turn uses build server messages.

Test Harness messages:

This package is mainly used for serializing and de serializing complex data structures used in Test Harness. It can create a Test Request message and a Test Results message. It also can parse a Test Request message and a Test Results message. It is used by Builders to create a Test Request message and It is used by test harness to parse that test request message. Test harness creates results of tests by using Test result message.

Build Request Parser:

This package is used by child builder to parse the build request. It uses reference of Build server messages which contains all the definitions of build server messages because to parse the request we need to know the structure of the request.

Child Builder:

This package is mainly used for building the libraries of different kinds of source files like C++, Java, C#. To build libraries it uses build config package which contains all the configurations and tool chains of target platforms. It starts building libraries after receiving build request messages from the Mother builder. It uses message passing communication channel to access messages from the mother builder. If the build succeeds, sends a test request and libraries to the Test

Harness for execution, and sends the build log to the repository. It uses Logger package to internally store all the logs at build server.

Mother Builder (Build Server):

The build server may have very heavy workloads just before customer demos and releases. We want to make the throughput for building code as high as is reasonably possible. To do that the build server will use a "Process Pool". This package is mainly responsible for creation of process pool. That is, a limited set of processes spawned at startup. The build server provides a queue of build requests, and each pooled process retrieves a request, processes it, sends the build log and, if successful, libraries to the test harness, then retrieves another request.

Ready queue will be holding all the ready messages and build requests queue will be holding all the build requests. Each time mother builder will dequeue both queues and send build request to the child builder whose address is present in the dequeued ready message. After child builder processing is done then it again sends back a ready message which will be inserted in a ready queue. In this way we are using the process again. Pool Processes uses Communication service to access messages from the mother Builder process. Below figure clearly illustrates Build server and child builder's interaction through COMM which is message passing communication service.

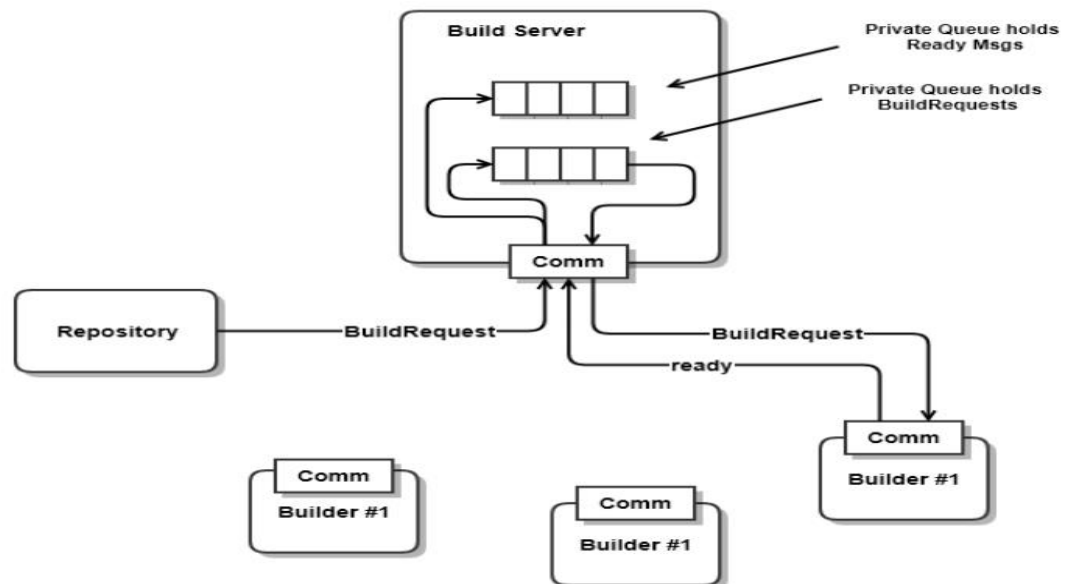


Figure 4: Process pool

Mock Client:

This package will be mainly responsible for providing the interface to the users. This is mainly developed by using windows presentation foundation and using message passing

communication service. Users can generate build requests which can contain multiple build items. Users can send build request structures to the repository for storage and transmission to the Build Server. Users further able to request the repository to send a build request in its storage to the Build Server for build processing. Users can also be able to view the build logs and test logs.

App Domain Manager:

There may be cases when users send the same request multiple number of times. so test harness has to delete the previous dll's files delivered by child builders if we don't use the concept of app domain test harness cannot delete the previous dll's files because they are loaded into the current app domain. It gives you exception when you try to delete them because it is already loaded in memory. To delete them we must use the concept of app domain.

This package is mainly responsible for creation of child app domain which will be used by test harness to load the libraries (dll's). After the processing test harness will be using app domain manager to unload the child domain that is created. This could be implemented in C# as follows

```
AppDomain main = AppDomain.CurrentDomain;

// Create application domain setup information for new AppDomain
AppDomainSetup domaininfo = new AppDomainSetup();
domaininfo.ApplicationBase
    = "file:/// " + System.Environment.CurrentDirectory; // defines search path for
assemblies

//Create evidence for the new AppDomain from evidence of current
Evidence adevidence = AppDomain.CurrentDomain.Evidence;

// Create Child AppDomain
AppDomain ad
    = AppDomain.CreateDomain("ChildDomain", adevidence, domaininfo);

////////////////////////////////////
// Way to create ChildDomain using default evidence and domaininfo
// AppDomain ad = AppDomain.CreateDomain("ChildDomain", null);

ad.Load("DemoClassLibrary");
//showAssemblies(ad);
Console.WriteLine("\n\n");

ObjectHandle oh
    = ad.CreateInstance("DemoClassLibrary","AppDomainDemo.hello");
object ob = oh.Unwrap(); // unwrap creates proxy to ChildDomain
Console.WriteLine("\n {0}",ob);

//_
////////////////////////////////////
// Alternate way to create instance, providing more information
// BindingFlags flags
//     = (BindingFlags.Public |
//     BindingFlags.Instance |
//     BindingFlags.CreateInstance);
// ObjectHandle oh = ad.CreateInstance(
//     "DemoClassLibrary","AppDomainDemo.hello",
//     false,flags,null,new string[0],null,null,null);

// using object in ChildDomain through IHello interface
AppDomainDemo.IHello h = (AppDomainDemo.IHello)ob;
h.say();

// unloading ChildDomain, and so unloading the library
AppDomain.Unload(ad);
```

5. KEY APPLICATION ACTIVITIES

The overall flow of the application could be clearly illustrated by the below diagram

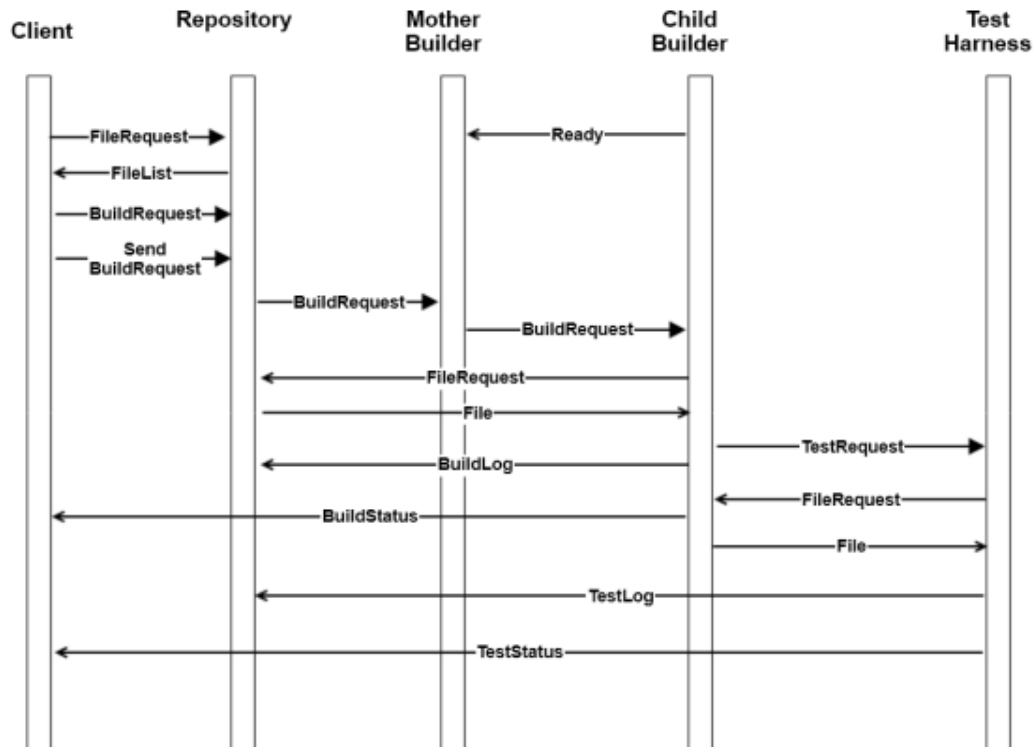


Figure 5: Remote build Server message flow

Key tasks of the application would be building a process pool component by using message passing communication service because build server may have very heavy workloads just before customer demos and releases. We want to make the throughput for building code as high as is reasonably possible. To do that the build server will use a "Process Pool". That is, a limited set of processes spawned at startup. The build server provides a queue of build requests, and each pooled process retrieves a request, processes it, sends the build log and, if successful, libraries to the test harness, then retrieves another request.

The overall flow of activities for this application is explained by the below activity diagrams. The activities involved in the application are described in a sequence.

General activities at Mock client:

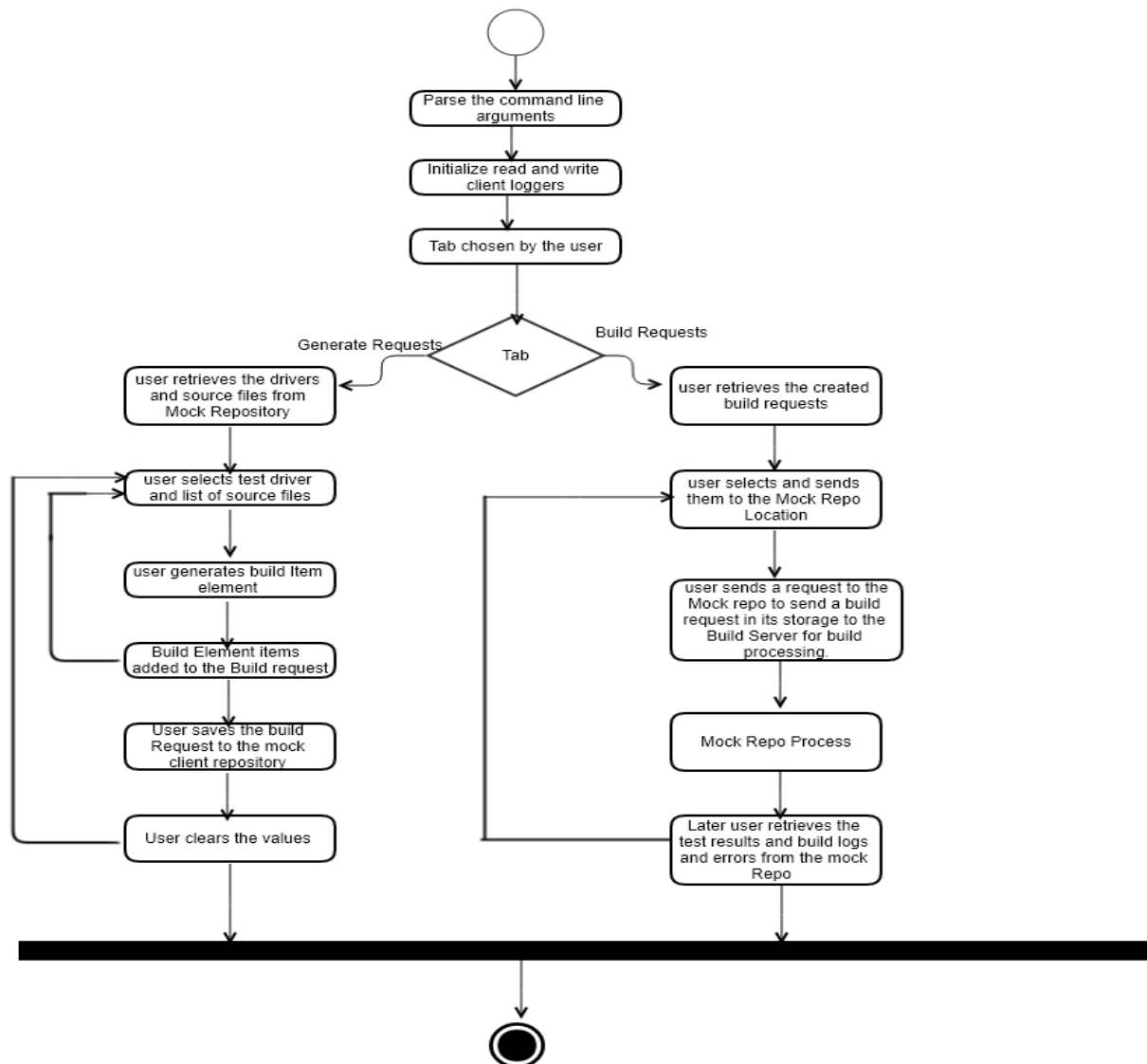


Figure 6: Activity diagram for Mock client

The above figure shows the general activity flow of the Mock client side and is explained below:

Build Request Generation

If user chooses generate build request tab, then he would be having capabilities to generate a build request. For generating build request, he would be getting all the repository contents

from the mock repository by clicking get repo contents button. Then user selects test driver and list of source files and click on generate button to generate build item which is added to build request. User can repeat this process and add as many as build items to the build request and later they can click on save button to store the xml build request at client local repository.

Handling generated Build Request

If user chooses handle build requests, then he would be having capabilities to send the generated build request to mock repository for validation where mock repo process will validate the request and reply validation status to the client. User also can request the repository to send a build request in its storage to the Build Server for build processing where processing of federation servers like build server and test harness starts and after the process is finished they will be sending build logs and test logs to the mock repo. Later client switching to logs view can retrieve all the logs by from the mock repository by clicking on get remote logs button and download button. Mock client can load all the logs by clicking on load downloaded logs and double click to open any log in a separate pop up.

Below Figure represents sample build request that is generated through mock client user interface

```
<?xml version="1.0" encoding="utf-16"?>
<BuildRequest>
  <Builds>
    <BuildItem>
      <driver>
        <file>
          <name>SixTestDriver.cs</name>
        </file>
      </driver>
      <sourcefiles>
        <file>
          <name>Interfaces.cs</name>
        </file>
        <file>
          <name>TestedLib.cs</name>
        </file>
        <file>
          <name>TestedLibDependency.cs</name>
        </file>
      </sourcefiles>
    </BuildItem>
  </Builds>
</BuildRequest>
```

Figure 7: Sample Build Request

Below is the sample build log received by the mock client after a successful build

```
##### start of the Build request #####

##### log start for the Test driver #####

the command and output that is executed is shown below
the time stamp is 131569245865830912

csc /target:library SixTestDriver.cs Interfaces.cs TestedLib.cs TestedLibDependency.cs

Generated----->SixTestDriver.dll in ../../CoreBuilder/Builderstorage/Builder8091

##### log end for the Test driver #####

##### end of the Build request #####
```

Figure 8: Sample Build log

Below is the sample test log received by the Mock Test harness after testing libraries

```
##### Starting of a test #####

The time when Test harness executed these logs is131569245870362422

start of log for TestRequest1.xml

loaded {0} SixTestDriver.dll

test {0} failed to run

attempting to create instance of {0} ,TestBuild.Test1

##### Test status #####

test {0} passed

##### Ending of a test #####

##### Starting of a test #####

The time when Test harness executed these logs is131569245870518657

test {0} failed to run

attempting to create instance of {0} ,TestBuild.Test2
```

Figure 9: Sample Test log

[General activities at Mock Repository](#)

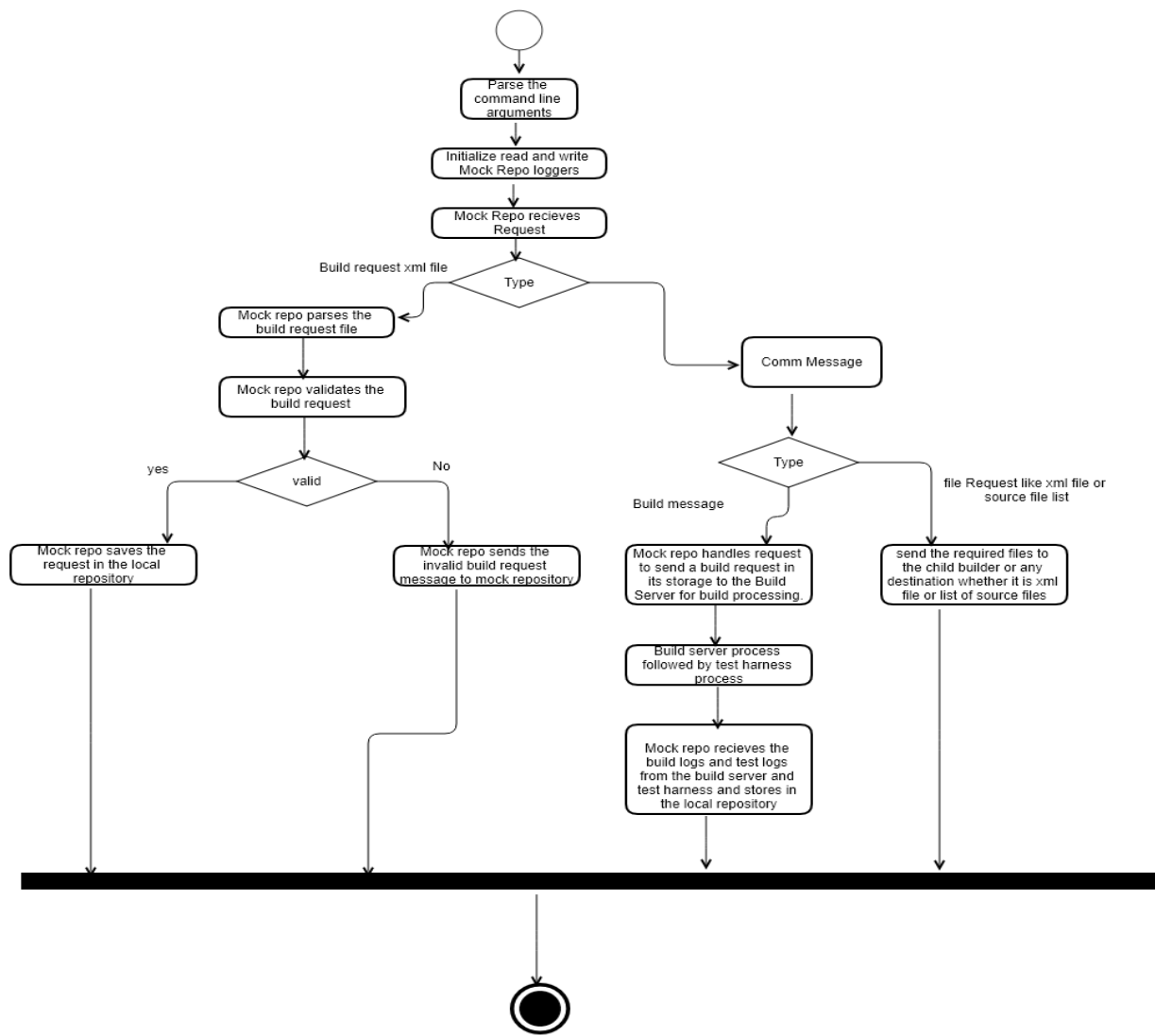


Figure 10: Activity diagram for Mock Repository

The above figure shows the general activity flow of the Mock Repository side and is explained below:

Validating build request file

If mock repo receives a request type build request xml file, then mock repo parses that build request xml file and validates that build request xml file if it is valid then saves that request in its storage and sends success validation message to the mock client if not it ignores the build request received and sends validation failed status to the mock client.

Processing Build request

If mock repo receives a COMM message type build message, then mock repo handles the request to send a build request in its storage to the Build Server for build processing. After forwarding the request to build server then processing starts like building source files by child builders and loading dll's by test harness. Then after this process happens it receives build logs as well as test logs from child builders and test harness.

Processing file Requests:

If mock repo receives a COMM message type file request, then it would all the corresponding files that are requested.

General activities at Build server (Mother builder)

Major Activities are

Process Pool Creation:

By using the command line arguments build server creates process pool where each process has the functionality of builder. It uses process class to achieve that where it can specify all the command-line arguments required by child builder in the process start info arguments. Each pooled process retrieves a request, processes it, sends the build log and, if successful, libraries to the test harness, then retrieves another request.

Handling Blocking Queue Messages:

Build Server contains a thread which will be keep on listening for messages if it receives a ready request from child builders then it inserts in to the ready blocking queue. If it receives build request, then it will be inserted in to the build request blocking queue.

Build server contains main thread which dequeues messages from the blocking queues and forward it to the child builders basing on ready message.

The below figure shows the general activity flow of the Build Server and is explained below:

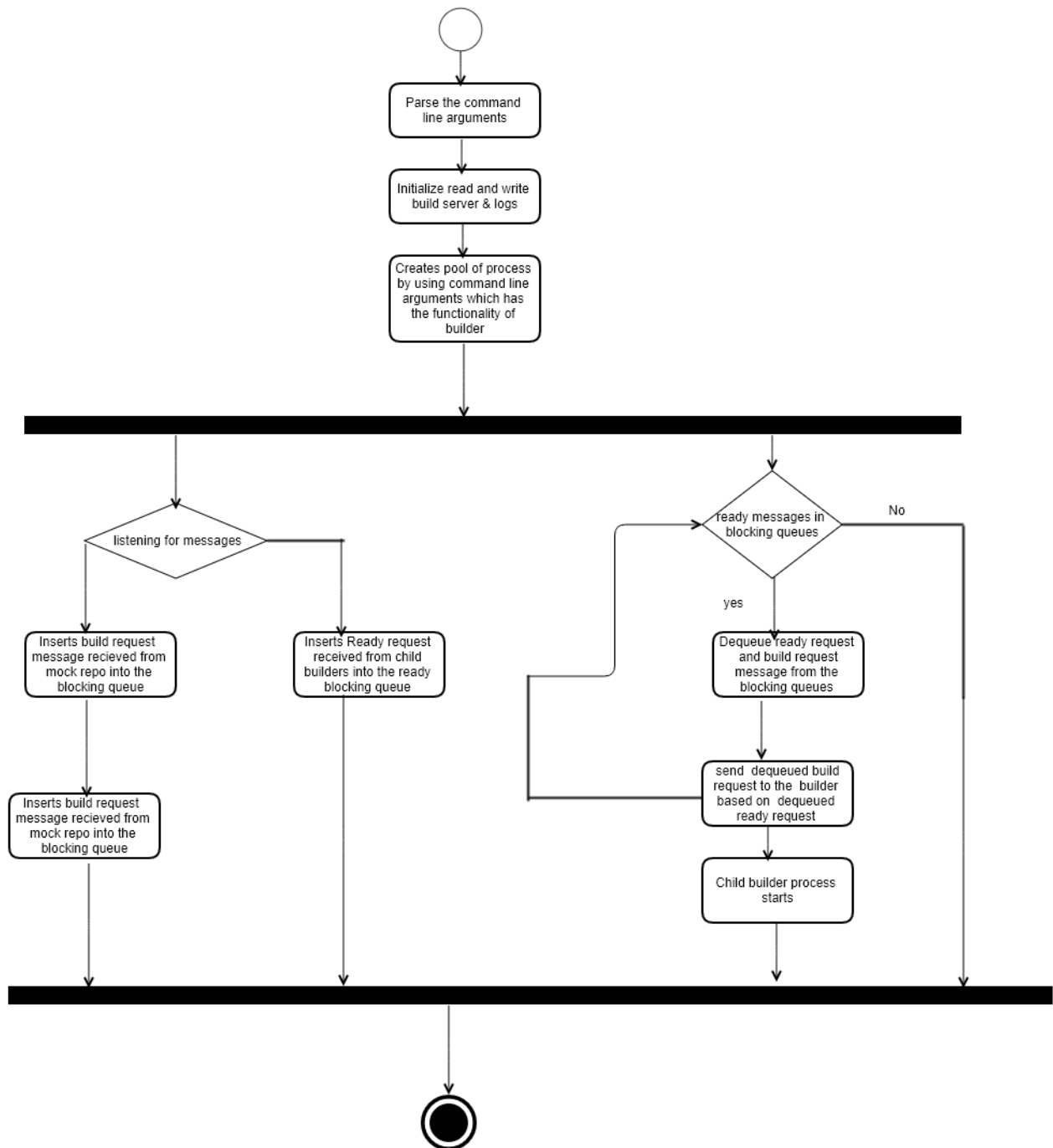


Figure 11: Activity diagram for Build server

General activities at Child Builder

The below figure shows the general activity flow of the Child Builder side and is explained below:

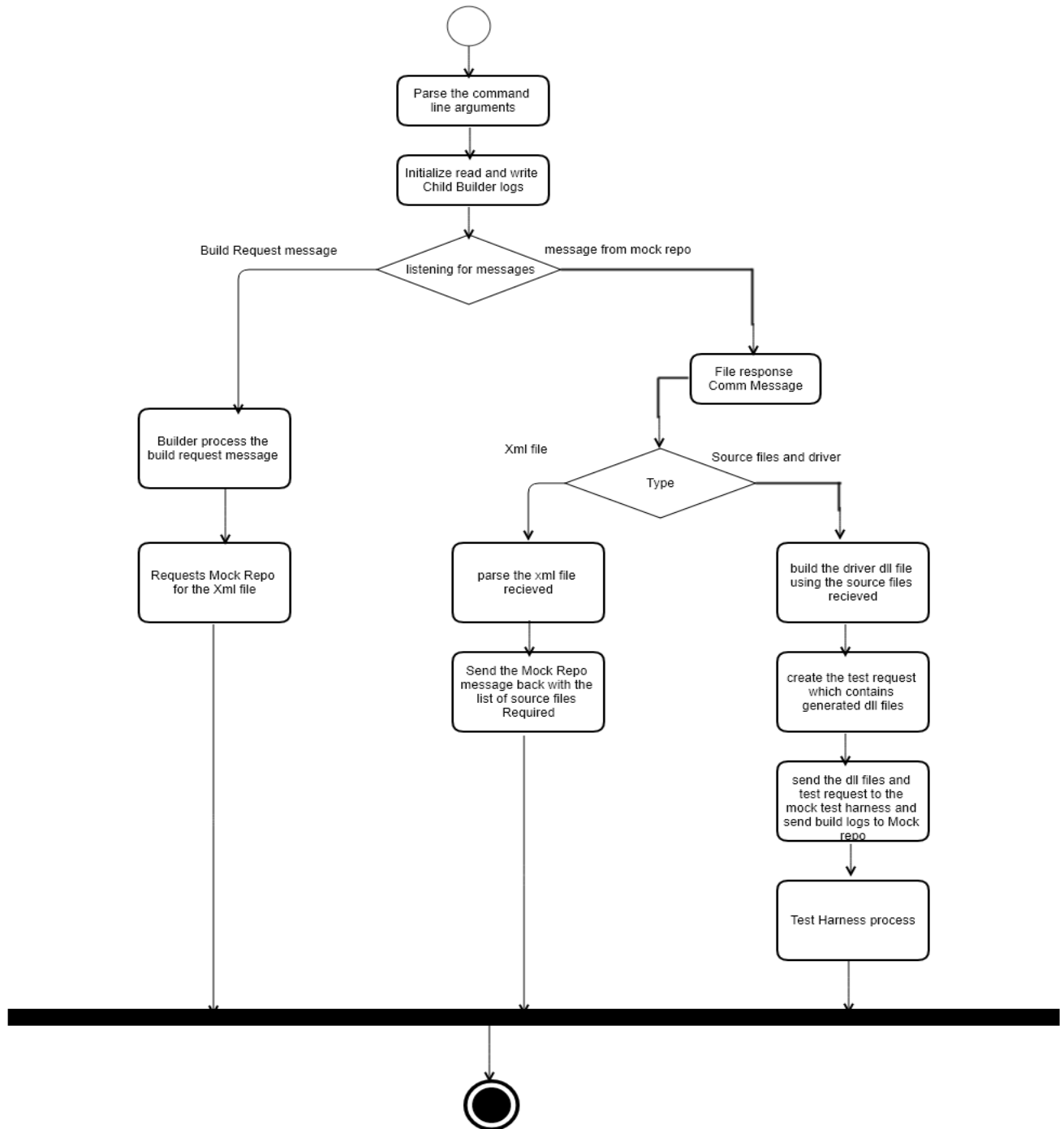


Figure 12: Activity diagram for Child Builder

Handling Build Request messages

In Builder there will be thread running which keeps on listening for messages if builder receives COMM message Build request message then builder process the build request message and requests mock repo for the xml file.

Handling File request messages

If builder receives COMM message xml file, then builder will parse the xml file received and send the mock repo list of source file that are required.

If builder receives COMM message source files and driver then builder will build test driver and generate test request and deliver the libraries and test request to the mock test harness. from where the Mock test harness process starts. Finally send logs to mock repository.

General Activities at Test harness

Major activities are

Parsing of test Request

In test harness a child thread will be keep on listening for messages if it receives the COMM message test request and dynamic link libraries files then test harness will parse that test request and sends the validation status back to the Child Builder.

Loading Dynamic link libraries

If the validation status is true, then test harness will create a child app domain then load all the libraries in that child app domain and run all the tests in that app domain and finally unload the child app domain and send test logs to mock repository.

Deletion of old Dynamic link libraries

During start up it deletes all the unused libraries and test requests. For each request test harness creates a temporary directory where it does all the processing and later delete that temporary directory which will be very useful for managing files.

Using the concept of App domain makes test harness work efficiently. It also leads to efficient management of resources.

The below figure shows the general activity flow of the mock Test harness side and is explained below:

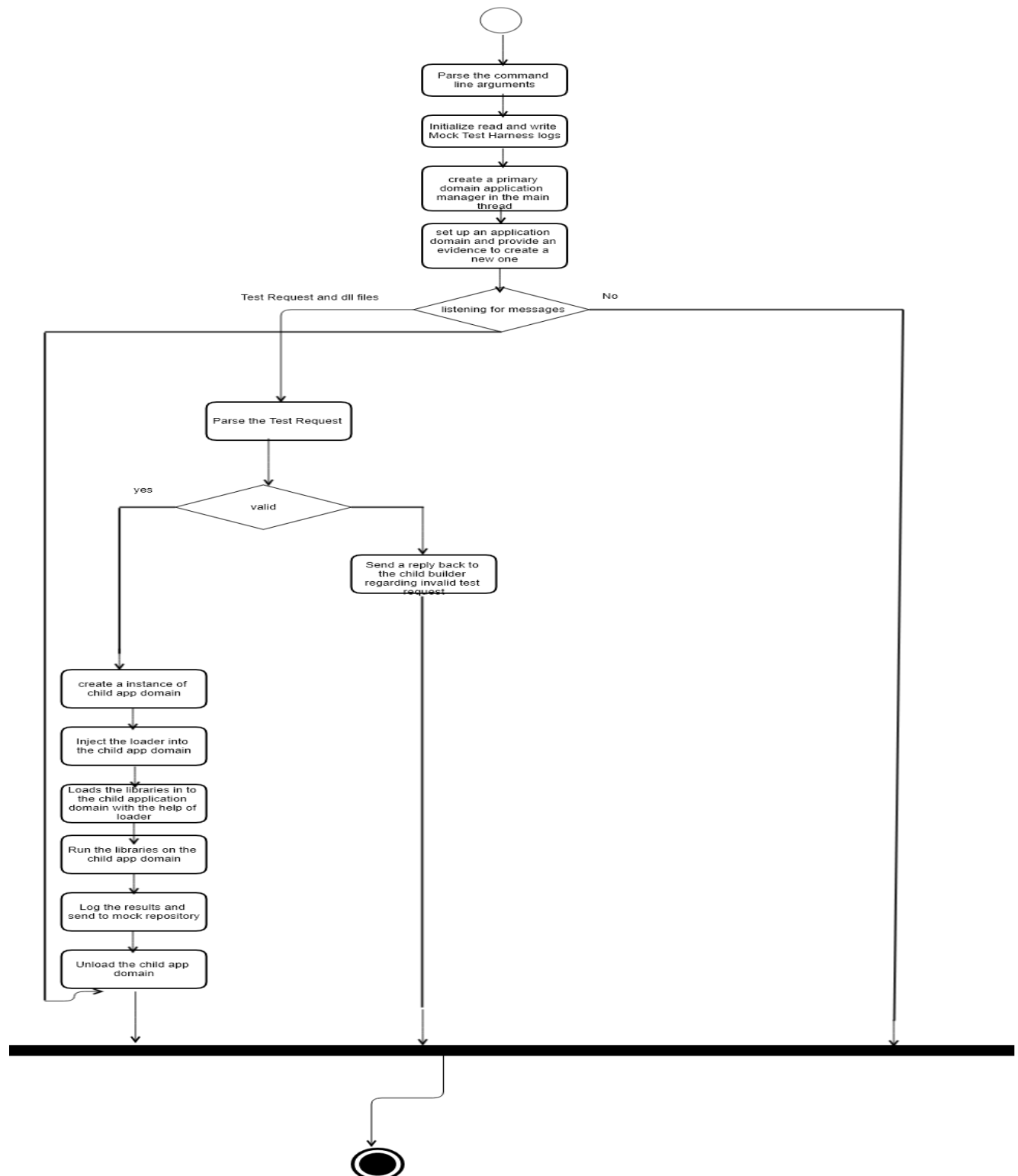
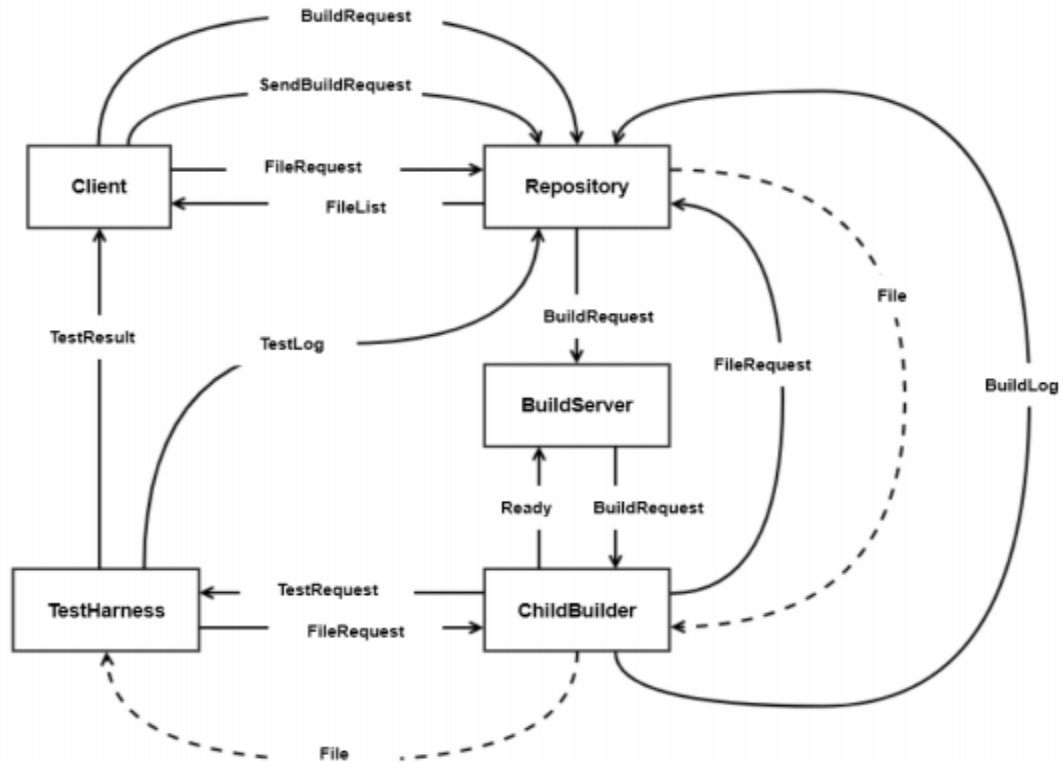


Figure 13: Activity diagram for Mock Test Harness

Below is the diagram summarizing all the possible message flows between the federation servers.



Below is the table representation of different type of messages used in Remote build server

BuildRequest – XML string	SendBuildRequest – command
FileRequest – command	FileList – list of strings
BuildLog – text string	File – binary
Ready – command (status)	TestRequest – XML string
TestResult – command (status)	TestLog – text string

For handling different messages, I have used message dispatcher in the federation servers.

Message dispatcher can be implemented with a Dictionary <Msg.command,Action<Msg>> . The Msg.command defines the type of processing needed for a message, and the Action defines the processing used to handle that message.

Message dispatcher could be explained clearly by the below diagram

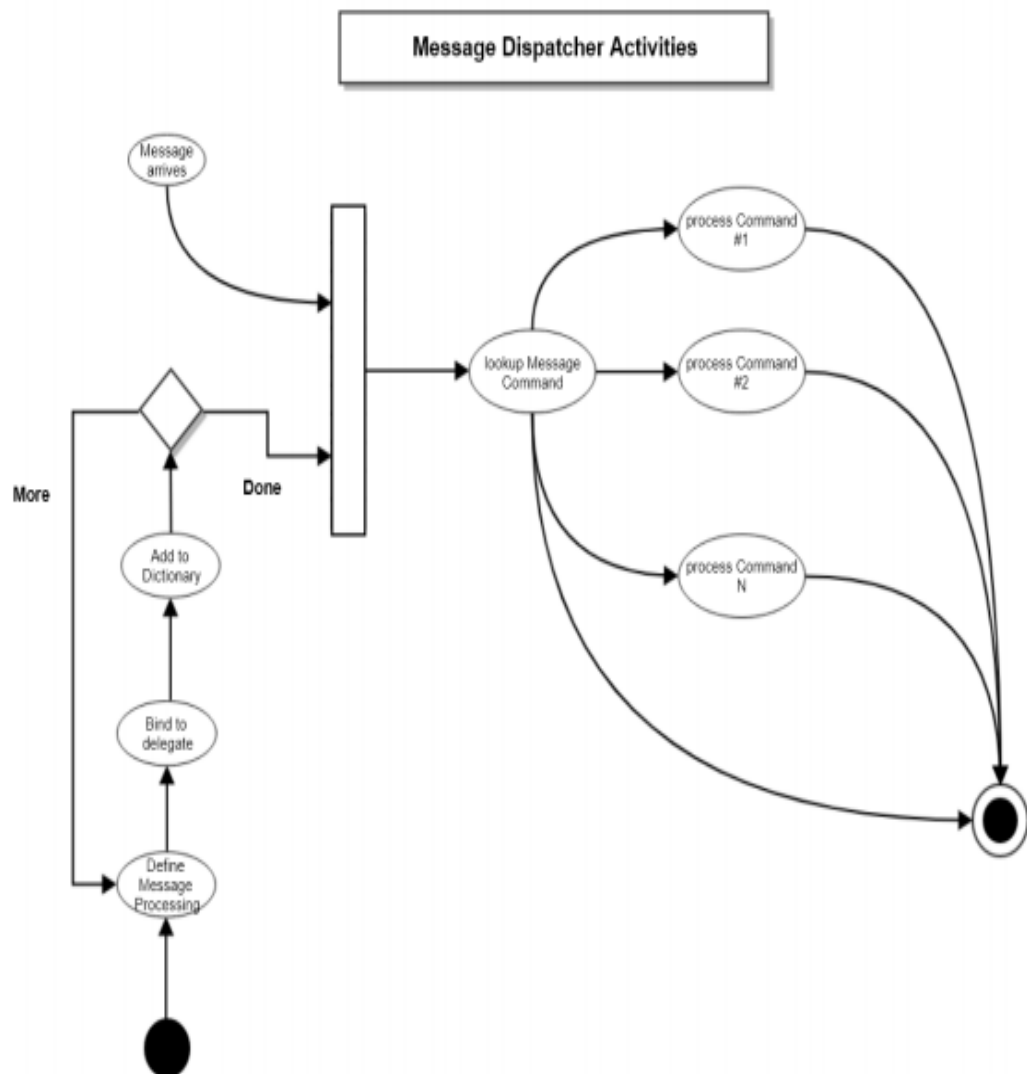


Figure 14: Message dispatcher diagram

6. Class diagrams

Mock client

Below class diagram shows the design of the mock client

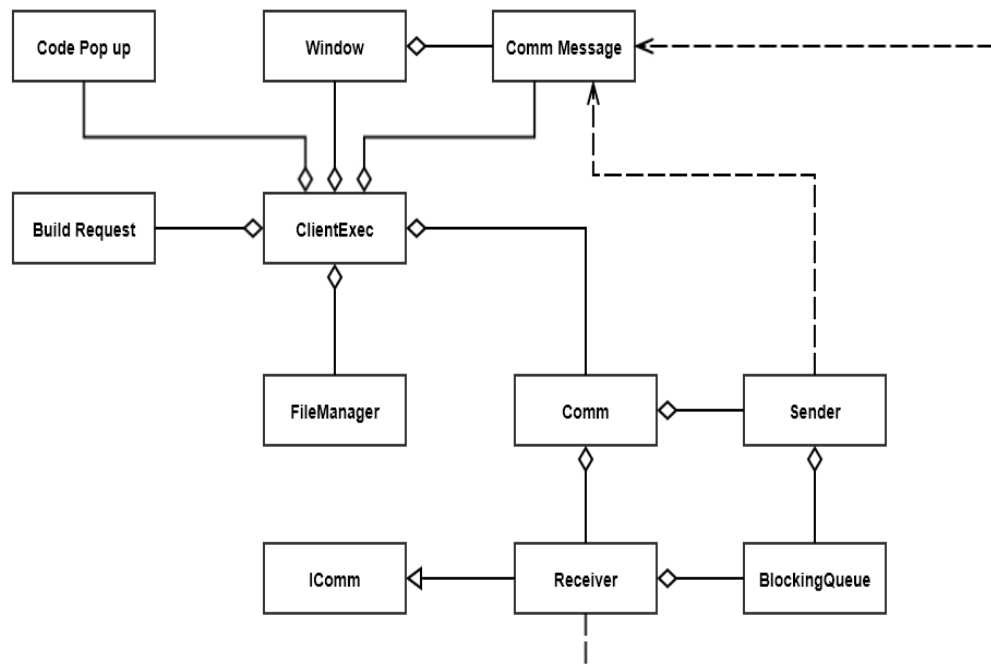


Figure 15: class diagram for Mock client

Build Request contains definitions of build request structure and has ability parse and get the source files.

Code pop up is used for displaying logs in a separate window

COMM and COMM Message support communication between Federation servers like Mock Repository

File Mgr is used to create temp directory, find the files for Comm.sendFile, and delete directory.

Sender can connect to a channel, post message, post file.

Receiver can start and close the service host, post message, get message, write file block, open files for write.

Blocking queue is thread safe and contains enqueue and dequeue methods.

Client Exec contains all the functionality like generating build requests and sending requests for storage as well as for processing for build server to Mock repository

Mock Repository

Below class diagram shows the design of Mock repository

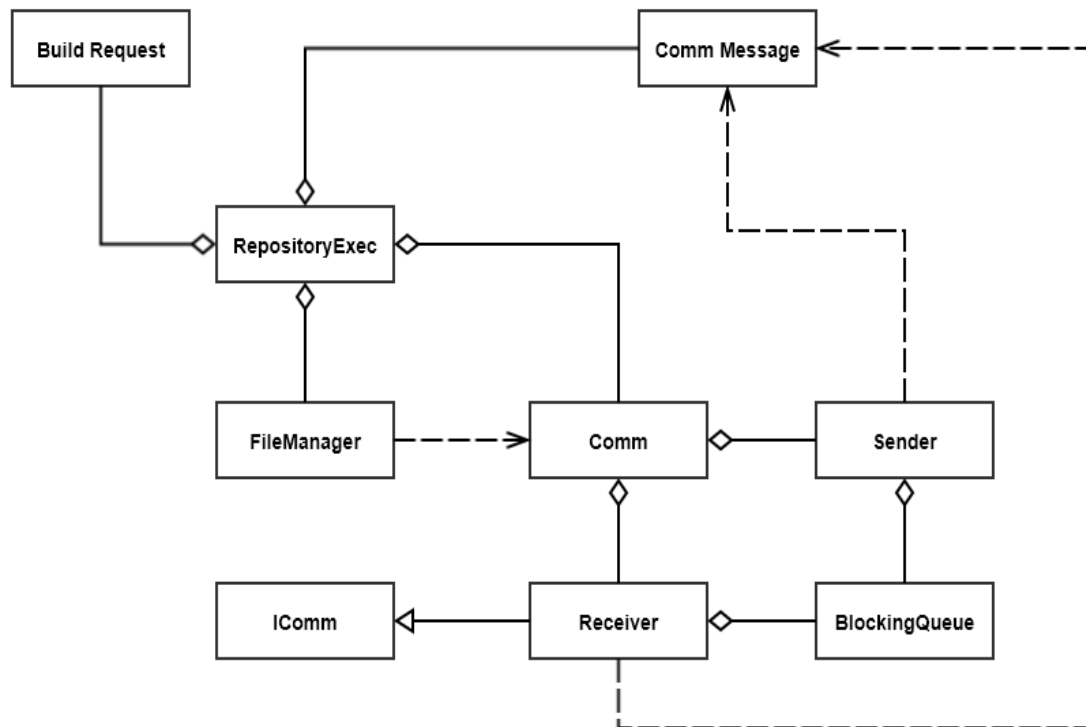


Figure 16: class diagram for Mock repository

Build Request contains definitions of build request structure and has ability parse and get the source files.

Sender can connect to a channel, post message, post file.

Receiver can start and close the service host, post message, get message, write file block, open files for write.

Blocking queue is thread safe and contains enqueue and dequeue methods.

COMM and COMM Message support communication between Federation servers like Mock Repository

File Mgr is used to create temp directory, find the files for Comm.sendFile, and delete directory. Repository Exec which contains functionality for validating the build request and responding to the file request from the federation servers like child builders and storing of logs sent by child builders and mock test harness.

Process pool

Below class diagram shows design of process pool which is the most critical implementation of the Remote Build Server.

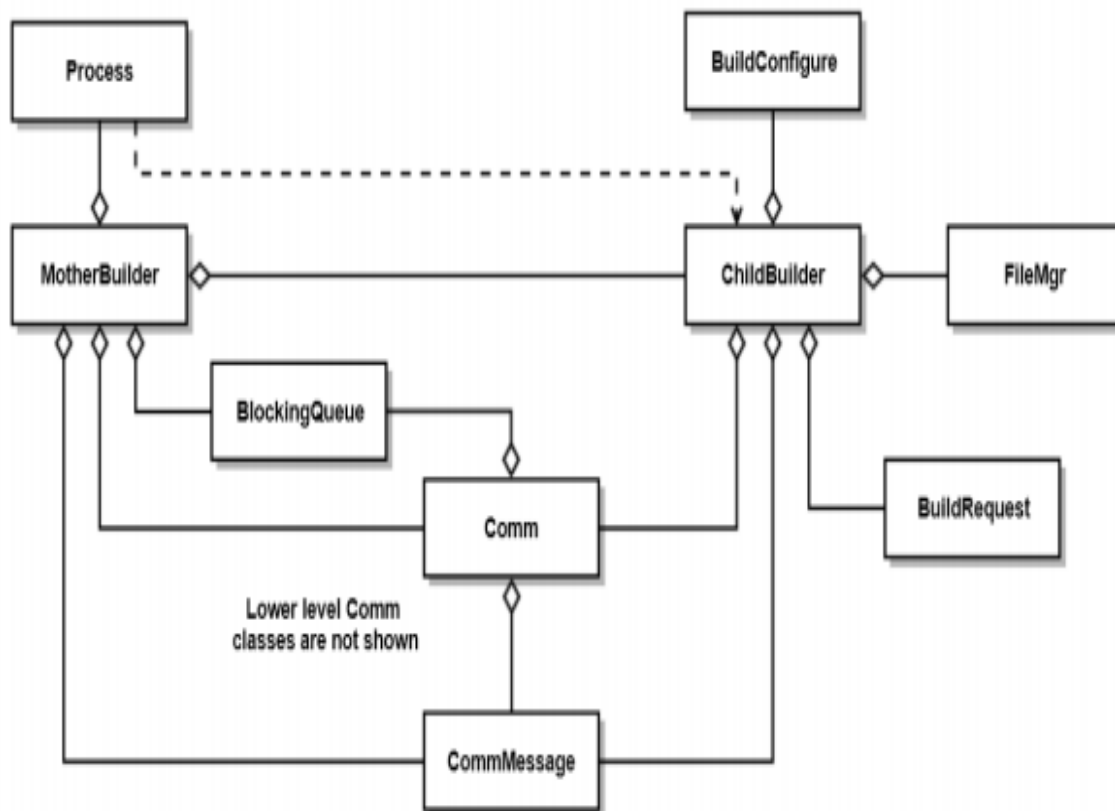


Figure 17: class diagram for Build Server(which illustrates process pool design)

Mother Builder manages Child Builders, passing them Build Requests when ready

Child Builder loads files, matching Build Request, from Repository and builds them into libraries and sends them to the Test Harness

Build Configure sets the environment and paths to toolchain to prepare for build

File Mgr is used to create temp directory, find the libraries for Comm.sendFile, and delete directory. System.Diagnostics.Process is used to support starting child Builders and tracking their exit events.

- COMM and COMM Message support communication between Mother Builder, Child Builders, Repo, and Test Harness

Build Request Parses Build Request message for source code file names

Mock Test Harness

Below class diagram shows the design of Mock test harness with all the details about methods

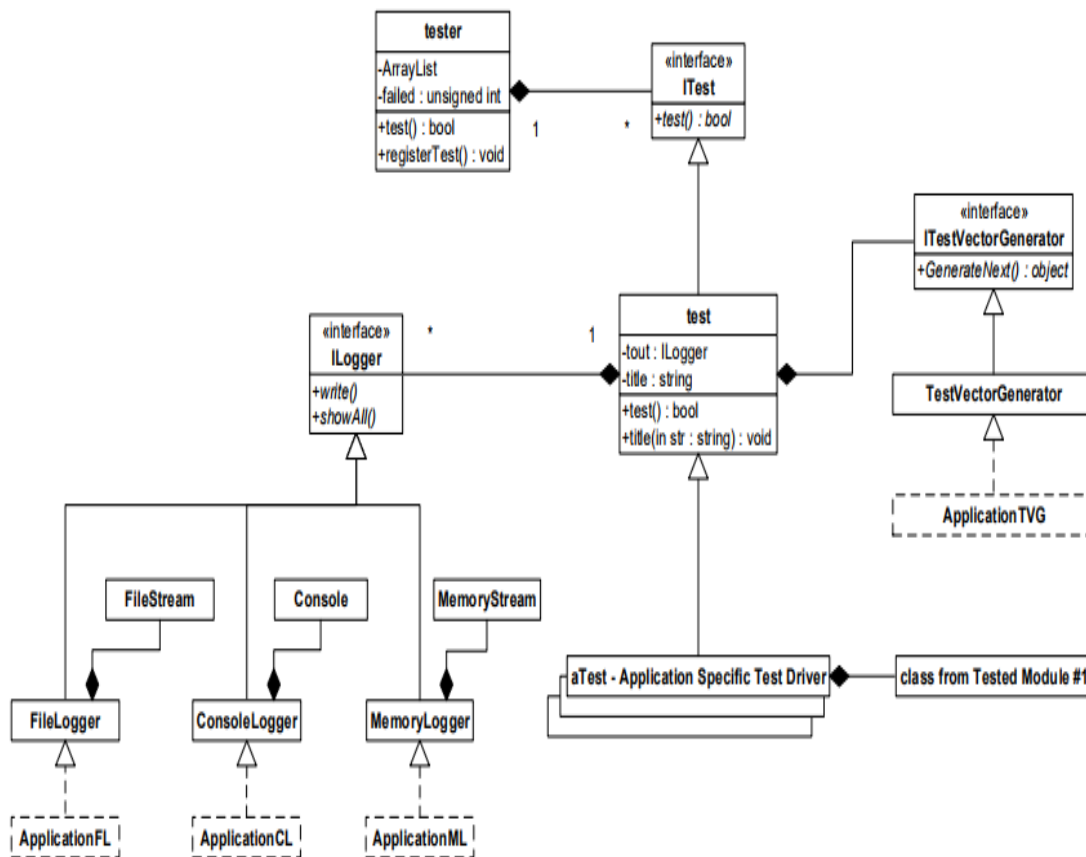


Figure 18 : class diagram for test harness

Loader loads test libraries into Test Application Domain

ITest declares ITest interface

Logger Provides logging facility for tests
Tester acts as Harness Test Aggregator
Test Vector Generator starter implementation

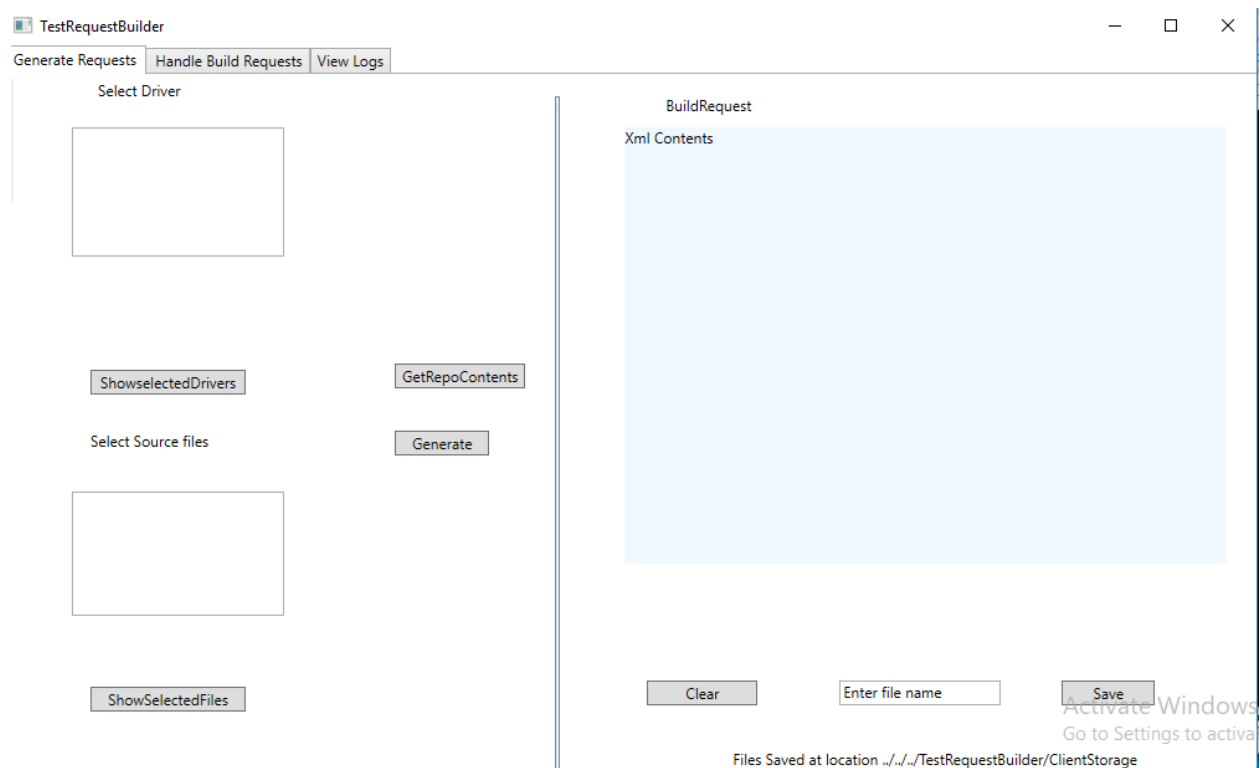
7. Views

Remote build server provides graphical user interface for the users to interact with the system. The client side of the system gives users ability to generate build requests and send the generated build requests for processing to mock repository which in turn sends to build server if it is valid build request and further process continues i.e. generating libraries and loading them by using mock test harness and provides functionalities as listed below.

Generate Build Requests
Handle Build Requests
View Logs

Below is the description of all the different views with their screenshots and description. As it can be clearly seen the GUI has different tabs for the user to perform different functionalities.

Generate Build Requests:



In this window user can retrieve the mock repository contents using GetRepoContents button and then drivers and source code files would be populated in the list box.

Users can select the driver and click on show selected drivers to get rid of unwanted drivers similarly select source files and click on show selected files to get rid of unwanted source files.

Users can generate a build request item after clicking on generate button which will be added to the build request.

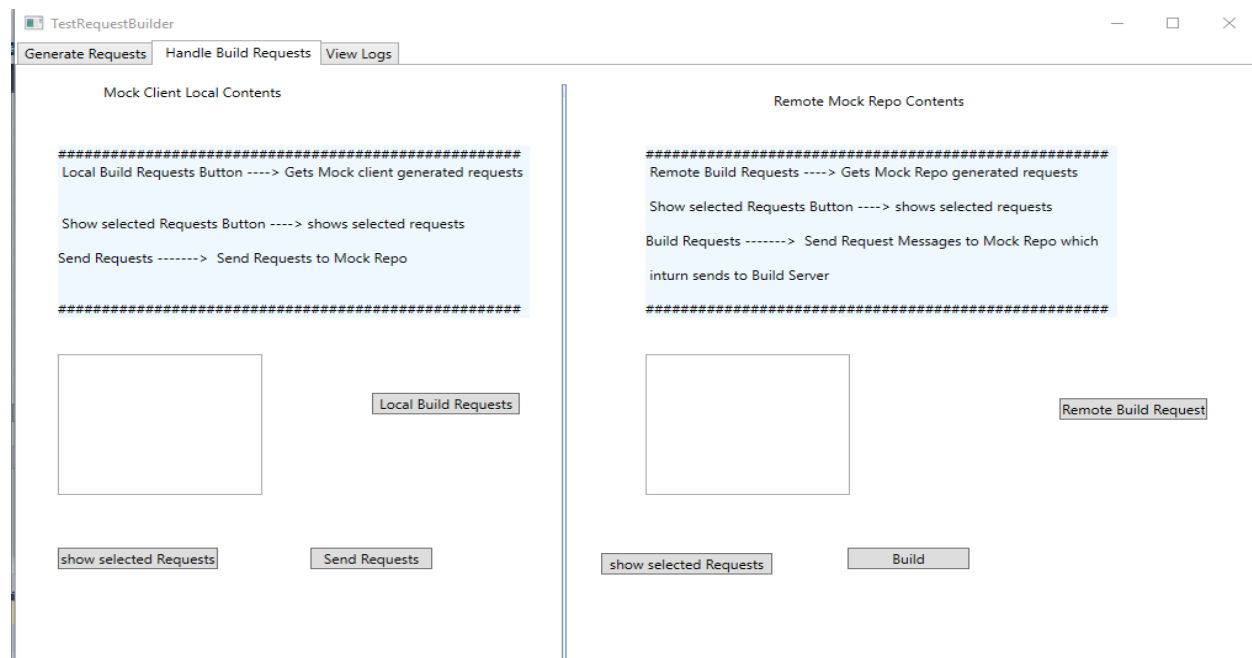
Users can again click on GetRepoContents to get the drivers and source files again next time when they click on generate new build request item would be created and added to the existing Build Request.

Users can view the build request generated in the WPF textblock.

Users can save the requests by clicking on save button and it would be saved to local mock client Repository.

If user want to start everything again they can click on clear button where all the values will be cleared from WPF elements.

Handle Build Requests



In this window users can view generated build requests on the list box beside local build requests button by clicking on that button.

Users can select the requests and click on show selected requests to get rid of unwanted build requests.

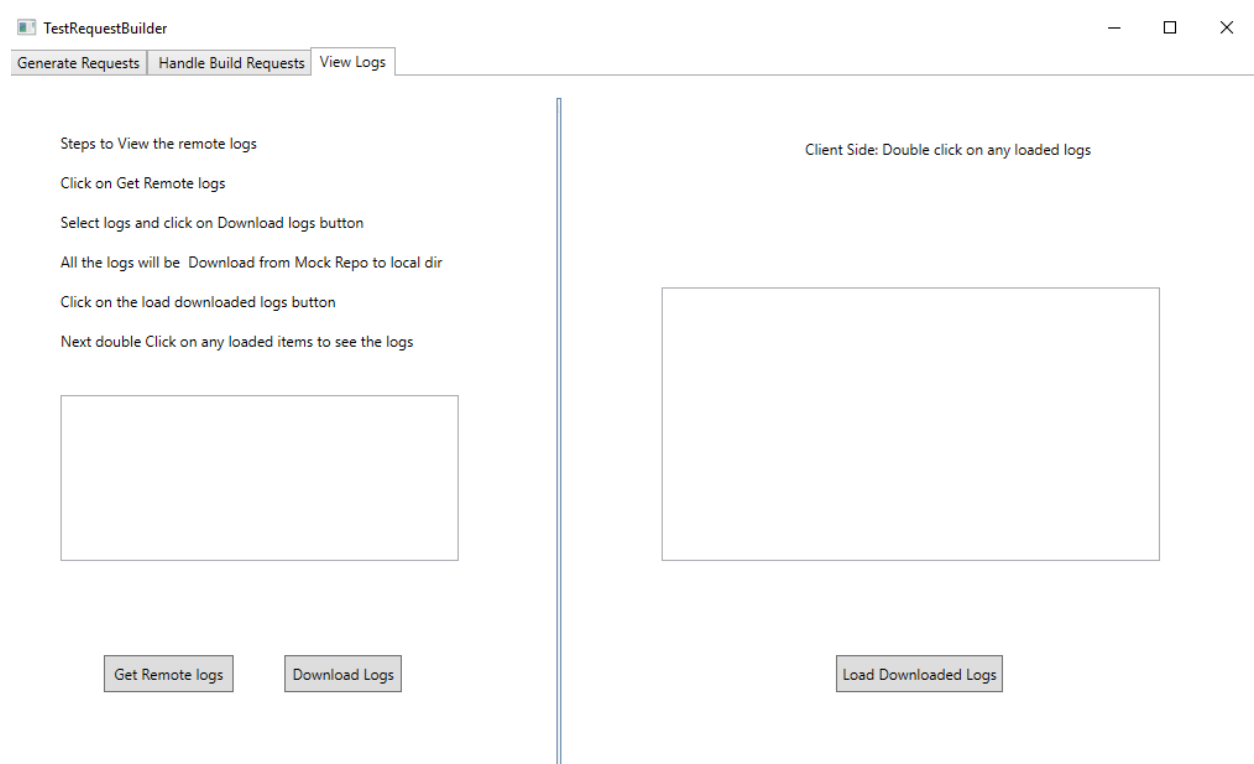
To send the requests for validation to mock repository they can click on send requests button.

Users can click on remote build requests button to get the mock repo successfully validated build requests.

Now users can select the requests and click on show selected requests to get rid of unnecessary requests.

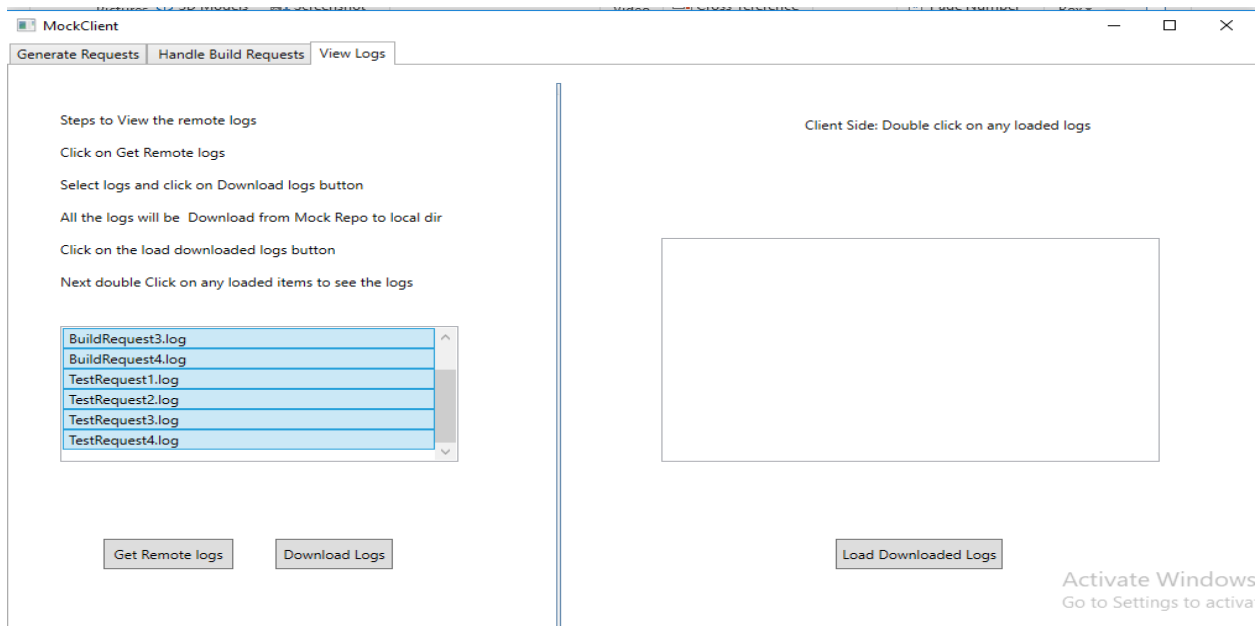
Now users can command the mock repo to send the selected requests in its storage to Mother builder for further processing by clicking on Build button. Then the background process starts like building source files and generating libraries and loading the libraries and sending logs to mock repo both build logs and test logs.

[View Logs](#)



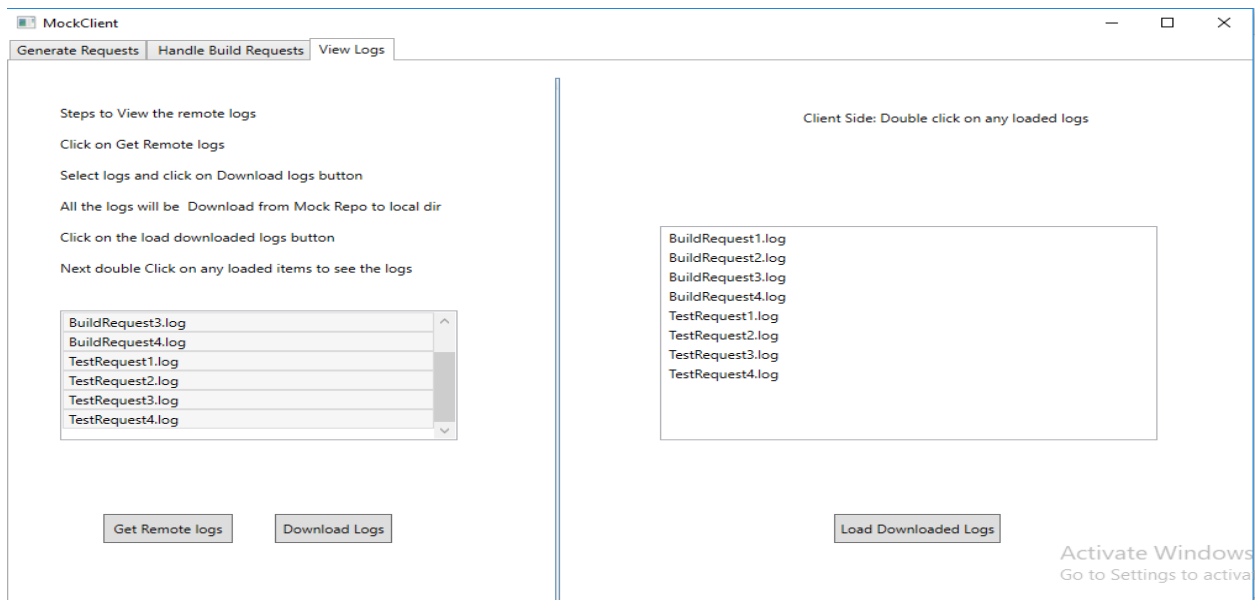
In this window user can view the remote logs.

Users first need to get all the build and test logs by clicking on get remote logs button.



Users can select all required logs and click on download logs where all the logs would be downloaded to local mock client repository

Users can load the downloaded logs by clicking on load downloaded logs



A new pop window will be created when you double click on loaded logs which are displayed on list box.



The screenshot shows a window titled "CodePopUp" with standard Windows window controls (minimize, maximize, close). The window contains a text area with the following log output:

```
##### start of the Build request #####

##### log start for the Test driver #####

the command and output that is executed is shown below
the time stamp is 131569245865830912

csc /target:library SixTestDriver.cs Interfaces.cs TestedLib.cs TestedLibDependency.cs

Generated----->SixTestDriver.dll in ../../CoreBuilder/Builderstorage/Builder8091

##### log end for the Test driver #####

##### end of the Build request #####
```

At the bottom right of the window, there is a watermark that reads "Activate Windows" and "Go to Settings to activate Windows."

8. Critical Issues

Ease of Use

Ease of use will be the one of the major concern for the developers. A complicated user-interface discourages the use of system, thus making a graphical user interface which is easier to understand and simpler to use is crucial.

Solution

Well Structured code design and different use cases should be taken into consideration. Graphical user interface created using windows presentation foundation should be able to upload files and test requests to the repository. But in project2 we are not building graphical user interface users will be most probably given the location of repository as the command line arguments to the automated build process

Building Different Sources

Build Server would be desirable to build sources from several used languages e.g. C, C++, JAVA. If we use our basic build process on MS Build Library, then it will not be working for Java and other Linux platforms

Solution

The solution is to create our own build infrastructure that uses compilers and tool chains for the target platform. You need to set up configuration file for each platform, tool chain that identifies the path of the tools and translating your builder commands into those needed by the specific tool chains

Throughput

As the build server is one of the busiest sever it will be having heavy workloads so through put should be as high as possible otherwise build process may take very long time

Solution

The solution is use concept of process pooling. A limited set of processes will be spawned at startup. The build server provides a queue of build requests each pooled process retrieves a request and process it, sends the build log and if build successful libraries to the mock test harness.

Malformed Code

There may be circular set of C++ include statements in the source code which overflow the process stack which may cause processes to crash

Solution

The idea is to create a new process replacement and report the build error to the repository to achieve this process pools should be always able to communicate with the builder process and each pooled process should have the functionality of build process

Exceptions

Build process may fail if there are any exceptions at run time

Solution

.Net provides to handle exceptions at runtime and developer must try to catch the runtime exceptions

Performance

Build server is one of the busiest server during releases and demos. Therefore, performance of build server is one of the critical issue.

Solution

Multithreading and sharing of resources would decrease a time required to process a build request. Without the use of multithreading each process need to wait in the queue for long time.

Versioning Control

There may be chances of build failures sometimes due to the new code modifications .so entire application servers may be down.

Solution

If proper versioning control is implemented in the repository then it may serve the previous version of working code which can be delivered to make things work instantly.

Inconsistency in Input Request

If the input request has some inconsistencies that is it misses some certain fields or values, then application might not be able to understand what to do at that situation

Solution

We should provide a validation to the input request to check all the mandatory fields are there in the input request otherwise ask the user to correct the request

Requirements Demonstration

Demonstrating that all the requirements are met to the graders and teaching assistants is one of the critical issue

Solution

Printing the outputs to the console for every requirement clearly can demonstrate the requirements are successfully met. Implementing of logger with the date and time stamp across the entire application architecture containing logs of every requirement pass fail status can easily help to demonstrate the requirements are successfully met.

Managing End Point information

Managing End Point information for Repository, Build Server, Child builders and Test Harness.

Solution

Store End point information in XML file resident with all clients and servers and load at startup.

Client crashes after sending a request

There can be a scenario where a client can crash after it has sent the request but not received response.

Solution:

Even though client might have crashed but server would have started the database operation, or worst finished it but failed to send the response. This will lead to wastage of server bandwidth. Server can implement a failed queue and store the result until client comes up. Once the client is ready it can send out the results to it. As the solution suggested, the design impact would accost the implementation of a queue that would store the result until the Client wakes.

Embed application logic in Client

The UI of the application contains a considerable amount of control, and would be the eventual executive of the application; this could lead to the temptation of dumping all the code behind each dialog window control to the respective button handler. This is a malpractice and even though it might sound easier to do, it usually causes the system to slow down considerably.

Solution:

The Client GUI in the application would act as nothing but a façade to the further complex sub systems that it hides. This leads to a light UI package structure with less amount of code, thus making it faster and adhering to code standards. Design Impact would be the client GUI package would act as a delegate, calling other respective functions based on which control button is accessed by the user. Thus the Client package structure would not contain the complex sub system design logic that actually carries out the bulk of the operations in the application

Message passing using Blocking queue:

Although implementation of blocking queue solves the issue of concurrent access, there may be situations when the there are an overwhelming number of messages in the blocking queue,

more than what the Server has been designed to handle. This leads to an issue with message passing using the Blocking queue.

Solution

A dynamic or open-ended blocking queue might be helpful in a situation of heavy load of messages, but eventually it might be a good idea just to let the system run its course and re-run the queries when the load is less. Design Impact would be dynamic blocking queue may be designed using C# and might look like this

```
public class BlockingQueue<T>
{
    private Queue blockingQ;
    object locker_ = new object();
    //----< enqueue a string >-----
    public void enQ(T msg)
    {
        lock (locker_) // uses Monitor
        {
            blockingQ.Enqueue(msg);
            Monitor.Pulse(locker_);
        }
    }
}
```

Defining a Single Message Structure

In the federation we need to define a single message structure that works for all the messages

Solution:

A message that contains To and From addresses, Command string or enumeration, List of strings to hold file names, and a string body to hold logs will suffice for all needed operations.

Testing both C# and C++ code

Testing different source code is one of main issue.

Solution:

For testing, trap exceptions on loading native code libraries in the C# Mock Test Harness and direct to C++ Mock Test Harness

Client sends same Request Multiple times

When users tend to send the same request, multiple times is one of the issue that needs to be handled.

Solution:

In such cases in the test harness there may be exceptions because we try to delete the previous dll's send by the child builders. They are still used by test harness so cannot be deleted unless

they are unloaded but if they are loaded into the main app domain we cannot unload them if we unload main app domain then program stops so here we need to create a child app domain and then load libraries in to that child app domain and run tests and unload the child domain. By doing this way we can safely delete the previous dll's files. Now client can send same requests any number of times no problems will arise.

9. Deficiencies

Authorization and Authentication:

Authorization and authentication has been not implemented for Remote build Server. Any one can access the application and perform the builds. Due to which critical information will be exposed.

Building Only C# source files:

Remote build server has been implemented for only building C# source files but not for C++ or Java.

Implemented message dispatcher only for Mock client

Build server, Mock repository, Test harness not implemented message builder if would had been then code will be in a more readable, simpler and understandable way.

Not Added Build Request declarations in Comm Message

If it would have been then complexity would be very less rather than sending xml files we can send comm messages which contain build request. Computation would be very fast.

Bad Version management in mock repository

Mock repository does not have version management it only stores the latest file version. If want previous version, then it would not serve the purpose

Local File storage in Mock repository

Storing all the files in the local mock repository system storage rather than that we store everything in the cloud and let mock repository know the file storage location in the cloud. This would be better design because auto scaling is also possible in cloud.

10. Conclusion

In conclusion to successfully implement big software systems we need to support continuous integration which can be achieved by the federation of servers that is Repository Server, Build Server, Test Harness. Thus, application that we developed provides scalability, flexibility and performance for building big software systems. This OCD explains which actors and how they use the system. We discussed about several use cases. The system is divided into cohesive packages which can interact with each other to perform all the tasks defined in the requirements. It also provides code reusability.

We have covered all the activities being carried out by the system. It describes how events takes place in the system and makes it easy to understand the system. We also discussed few critical issues which can result in serious design flaw if not provided the proper solutions. It can be ensured that from this OCD that system can be developed in considerable period with the available resources.

11. References:

<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project1-F2017.htm>
<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project2-F2017.htm>
<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project3-F2017.htm>
<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project4-F2017.htm>
<https://stackoverflow.com/questions/6511380/how-do-i-build-a-solution-programmatically-in-c#>
<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/StudyGuideOCD.htm>
<https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/AppDomainDemo/>
<https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project4HelpF2017/>
<https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project3HelpF2017/>