

# Remote Build Server

Operational Concept Document

Project #1

CSE681 Software Modeling and Analysis

Instructor: Dr. Jim Fawcett

Amritbani Sondhi, Fall 2017

SUID: 903888517

## CONTENTS

1. Executive Summary.....	3
2. Introduction.....	4
2.1 Objective.....	4
2.2 Application Obligations.....	4
2.3 Key Principles.....	4
2.4 Structure.....	5
<i>Client Module.....</i>	<i>5</i>
<i>i. GUI.....</i>	<i>5</i>
<i>ii. Display.....</i>	<i>6</i>
<i>iii. Mock Client.....</i>	<i>6</i>
<i>Mock Repository Server.....</i>	<i>6</i>
<i>Build Server.....</i>	<i>6</i>
<i>Mock Test Harness Server.....</i>	<i>6</i>
<i>File Manager Module.....</i>	<i>7</i>
<i>Request and Notification Module.....</i>	<i>7</i>
<i>i. Request Handler.....</i>	<i>7</i>
<i>ii. Notification Handler.....</i>	<i>7</i>
<i>Parser Module.....</i>	<i>7</i>
<i>Result Logging Module.....</i>	<i>8</i>
<i>i. Logger.....</i>	<i>8</i>
<i>ii. Result Handler.....</i>	<i>8</i>
3. Uses and Use Cases.....	8
3.1 Uses for People at Different Positions.....	8
<i>Code Developers as Clients.....</i>	<i>8</i>
<i>Quality Assurance Analysts.....</i>	<i>9</i>
<i>Project Managers.....</i>	<i>9</i>
<i>Instructors, Teaching Assistants and the Developer.....</i>	<i>9</i>
3.2 Uses at Organizational Level.....	9
<i>Frequent Check-Ins and Commits.....</i>	<i>9</i>
<i>Backed Up Code in the Repository.....</i>	<i>9</i>
<i>Logged Data.....</i>	<i>9</i>
4. Application Activities.....	10
4.1 Top Level Interactions.....	10
4.2 Activity Diagram for Build Server.....	12
5. Critical Issues.....	14
6. Conclusion.....	16
<i>Appendix.....</i>	<i>17</i>
• <i>Loading a Visual Studio Project from a path and compiling it.....</i>	<i>17</i>
• <i>Sample of the Logs retrieved from the Command Line.....</i>	<i>18</i>
• <i>Sample Test Request File.....</i>	<i>18</i>

## 1. EXECUTIVE SUMMARY

This document covers all the Operational Concepts for constructing a Remote Build Server.

Project development in the industry is at a completely new level. It varies from a few hundred lines of code and can go to millions of lines of code in a single project. This makes it necessary for the developer to keep on testing his code after ever short interval. The developer may choose to partition his code which will also help him create pluggable codes, and perform unit tests on each of them as they are built. This calls for a continuous compiling and execution of the code from time to time. This is achieved by the concept of Continuous Integration.

In Continuous Integration, small independent parts of the code are created step by step, they are compiled, and if successful, executed and added to the main line code. It can again be repeated for the newly formed main code too. This process is can be automated by having a 'Build Server'.

A remote Build Server is useful in the case of distributed development, where several developers are working on the same part of the code. It becomes important for them to keep checking in the code in a common repository so it could be available to the other developers with the latest changes. It is also expected that the new code is clear of any errors or bugs. So, running the code in a separate machine using a build server provides a clean environment. It lets the developer flag the code and get notified about the runtime and file dependency errors if any. It also contributes in version and change control. Another advantage is that the developer doesn't have to wait for the whole testing process to get completed. He can continue to work on the other parts of the project on his local machine till the Build Server completes testing and notifies him about the results.

## 2. INTRODUCTION

In big development systems, where there is large no. of code developed by many developers, managing and testing the code can be a tedious job. A remote build server can simplify and automate the testing of the code. Moreover, when distributed development is done, it is a best practice to build and check in small blocks of code and be continuously integrated with the main line of code, so that, finding and resolving the errors would become easy and the probability of someone else using an older version of the code will be less.

### 2.1 OBJECTIVE

To provide a mechanism where developers are encouraged to merge and commit their code to the main line with a hassle-free test structure to avoid heavy build fails.

It helps to keep the code reproducible as packages are distributed. The automated testing and logging keeps it traceable.

### 2.2 APPLICATION OBLIGATIONS

The Remote Build Server shall be able to provide means to automate the compilation and execution processes which takes place while developing an application such that it provides unit testing of the block as well as continuous integration of it to the main base code.

The following are some of the obligations a build server is expected to fulfill:

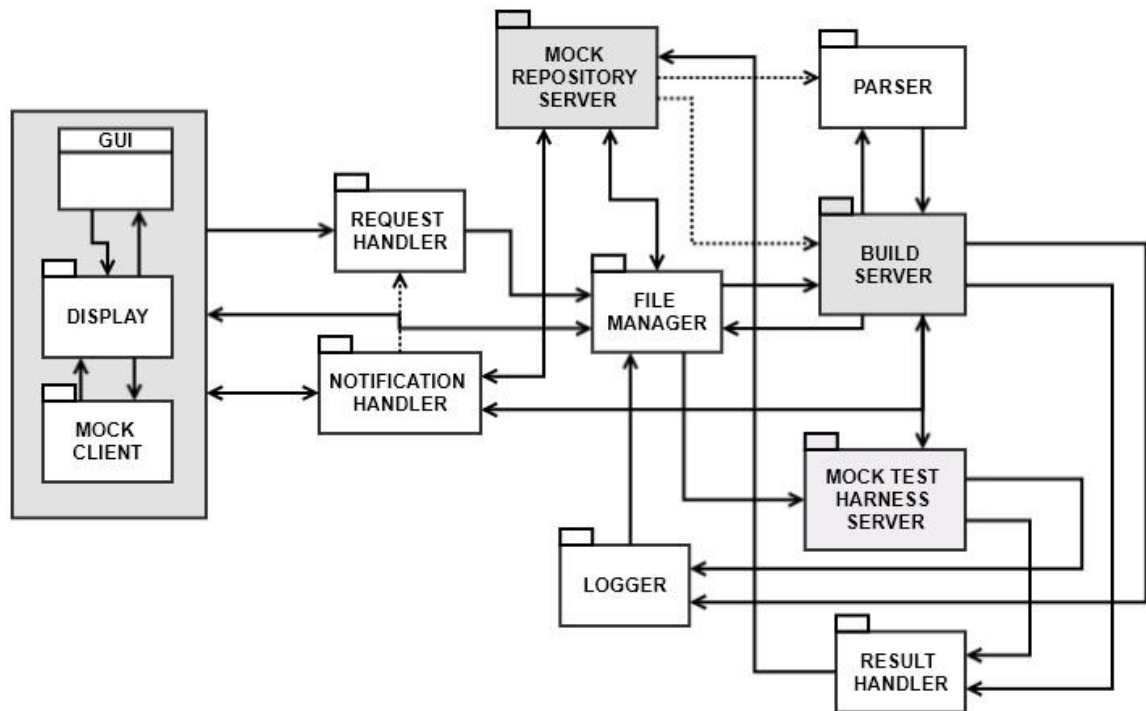
- To accept, code files and test requests sent by the client and store them in the Mock Repository.
- To identify the files types and to select a correct compiler tool accordingly.
- To attempt to build files and store the logs and test results in the Repository.
- To send the generated libraries and the test requests to the Test Harness.
- To trigger the Test Harness to execute the libraries and notify the clients about the test results.

### 2.3 KEY PRINCIPLES

The key principle is to automate the building and execution of the test files. The build server shall have tools to parse the test request and know which compiler is needed to build them properly. Also, the Build Server should be able to provide a clean environment and ensure that any compiler can be used for the process. The following languages are expected to be built and executed: C++, C# and Java.

## 2.4 STRUCTURE:

To fulfill all the obligations mentioned above, the top-level package diagram for this project is as shown below:



**Fig 2.4 Remote Build Server Package Structure**

There are two directions in which the process takes place in a Remote Build Server.

1. Client to the Repository Server, to the Build Server and Test Harness.
2. From Build Server and/ or Test Harness to the Repository and then to the Client.

### 2.4.1 CLIENT MODULE:

The Client Module comprises of a Graphical User Interface which we would refer as GUI, the Display and the Mock Client.

- **GUI:**

The GUI allows the Client to communicate with this application without the need to use the command line for sending the requests and code files and receiving the status and log files of the test. An interactive GUI will be created, so that the Client can create and send requests, browse and select files he wants to upload to the repository, giving commands to Build and Execute to the Build Server and view results and logs of the complete test.

- **DISPLAY:**

The Display package is a small part of the Client Module. It holds all the methods and properties, to get and send the data required by the GUI and the Mock Client.

- **MOCK CLIENT:**

The Mock Client is the actual link which communicates the requests and commands to and from the Repository and the Client. It then sends the appropriate data which is used by the Display package to be viewed at the GUI.

## **2.4.2 MOCK REPOSITORY SERVER:**

This acts as a Repository for storing all the files, requests, logs and results. It commands the Parser to parse the test requests sent by the client. The Mock Repository Server receives code files and test requests from the clients. It stores the build logs and build results received from the Build Server after it has completed the compilation process. It stores the test logs and test results received from the Test Harness after it has completed the execution process. It is also responsible to send the code files present in the repository when the client browses them for the test process.

## **2.4.3 BUILD SERVER:**

The main responsibility of the Build Server is to get the code files and test requests from the Repository, check which compiler to use for the building process and attempt to build the files. Before receiving any files from the Repository, it creates Temporary Directories to store the files for its use. It then parses the test requests for any file dependency information and selects a compiler suitable for the build process. It then attempts to compile the code files. When the compilation is done it is supposed to send all the logging information and the build results to store in the repository. In case the build fails, the build server must notify the client about the failure. Also, it clears the Temporary Directory and deletes it. This is done so that there are no junk files present for handling the next build request.

The Build Server, in the case of successful compilation, sends the test requests and the library created to the Test Harness.

## **2.4.4 MOCK TEST HARNESS SERVER:**

The Test Harness, too, creates a Temporary Directory before receiving any files. After the build process is completed, it receives the test requests and the Dynamic Link Library from the Build Server. When notified by the Client, it loads the libraries and attempts to run the DLLs and the test drivers. After the execution gets completed, it sends the log details and the test results to store in the repository and deletes its Temporary Directory. It also notifies the Client about the status of the execution.

### 2.4.5 FILE MANAGER MODULE:

This module is used for all the files and directory related processes. It will have methods which will allow us to get files which will be used by the Repository (for code files and requests), Build Server (for code files and requests) and the Client (for build and test logs and results); and to send files which will be used by the Client and the Repository for then code files and test requests. It will also provide the functionality to create Temporary Directories, used by Build Server and the Test Harness.

### 2.4.6 REQUEST AND NOTIFICATION MODULE:

The responsibility of this module is to handle all the requests, commands and notifications which being created, sent and received. It keeps the creation of the requests and notification logic separate from each other.

- **REQUEST HANDLER:**

The Request Handler creates the Test Request files. The Client sends all the details and configurations that are to be combined in a test request file and sends it to the Request Handler. The Request Handler creates and XML Document with all the details related to the build and test configurations such as: client information, datetime, test cases, test driver, test packages, code file dependency information if any.

- **NOTIFICATION HANDLER:**

The Notification Handler creates all the notification messages which are being passed in Remote Build Server system. It receives notification creating requests from the Mock Clients, Build Server and the Test Harness Server to: start build process, start run process, build pass and build failed, run pass and run failed notifications, which should be sent to their respective Servers.

### 2.4.7 PARSER MODULE:

The Parser parses the Test Requests when the Build Server needs the required configurations contained in the file. It gives information about the configuration of the file and intimates the Build Server about the tool which shall be used for the compilation process. It will also provide the functionalities to set up the development environment in the command prompt. Moreover, it parses the Test Request to provide the Test Drivers to the Test Harness which could be required for the execution process.

### 2.4.8 RESULT LOGGING MODULE:

The Result Logging Module is used for creating Log files and build and test results

- **LOGGER:**

The Logger provides the functionality to get the Console Logs and write it to a text file. This is used for logging the build and test log details which we get from the Build Server and the Test Server after the compilation and execution processes. The Logger will log details such as DateTime, Build/ Run Errors, Build warnings, Build/ Run Success and Build/ Run Failure.

- **RESULT HANDLER:**

The Result Handler will be used to provide the result of the complete process. This information can be used to notify the Client about the status of the Build and Run process.

## 3. USE CASES

The Build Server can be important to people at different levels as well as it can have several uses as an application itself. Let us divide the Use Cases with respect to the uses from an organization's point of view and with respect to people at different designations.

### 3.1 USES FOR PEOPLE AT DIFFERENT POSITIONS

- **CODE DEVELOPERS AS CLIENTS**

The users or the Clients of the Remote Build Servers will be the Code Developers working on different languages. The major advantage of using a Build Server by Developers is that it will provide a separate environment to test their code on. Usually an issue faced by developers is, while coding they include libraries and references on their machine, but when their code goes out, it fails to build because of some missing references. Using a Remote Build Server allows them to work on another server, and when the compilation is successful, the risk of failing it in another machine is reduced considerably.

Also, sometimes the building and testing of the code takes a long time, if the developer uploads the code to build and run in the Build Server, it saves a lot of time for him to continue working on some other functionality on his local machine. Thus increasing the efficiency of work.



- **QUALITY ASSURANCE ANALYSTS**

The performance of the code which is being continuously checked in the repository can be checked by QA Analysts.

- **PROJECT MANAGERS**

The Build Server functionality if used in a team, helps even the managers to keep track of the progress of the project. The managers and team leads can view the logs stored in the repository and check who completed which part of the product and when it was last successfully built.

- **INSTRUCTORS, TEACHING ASSISTANTS AND THE DEVELOPER**

It will be used by the Developer (ie. me) to demonstrate all the requirements of Project 4 to the Instructor and the TAs. The Instructors and the TAs will use it to check if it fulfills the requirements. It is expected to have the core Build Server functionalities. It shall use the Socket Based Message Passing Communication to send and receive requests and responses.

### **3.2 USES AT ORGANIZATIONAL LEVEL**

Apart from all the uses it has for the people, having Remote Build Server is advantages for a team and organization as well.

- **FREQUENT CHECK INS AND COMMITS**

The developers should keep checking in the code to the repository after every successful build. This helps when there are a lot of developers working on the same project and so they can have the same set of working mainline code. This ensures that there will be minimum risk of any developer using and working on the older version of the code as it will be always available to them with any commit changes.

- **BACKED UP CODE IN THE REPOSITORY**

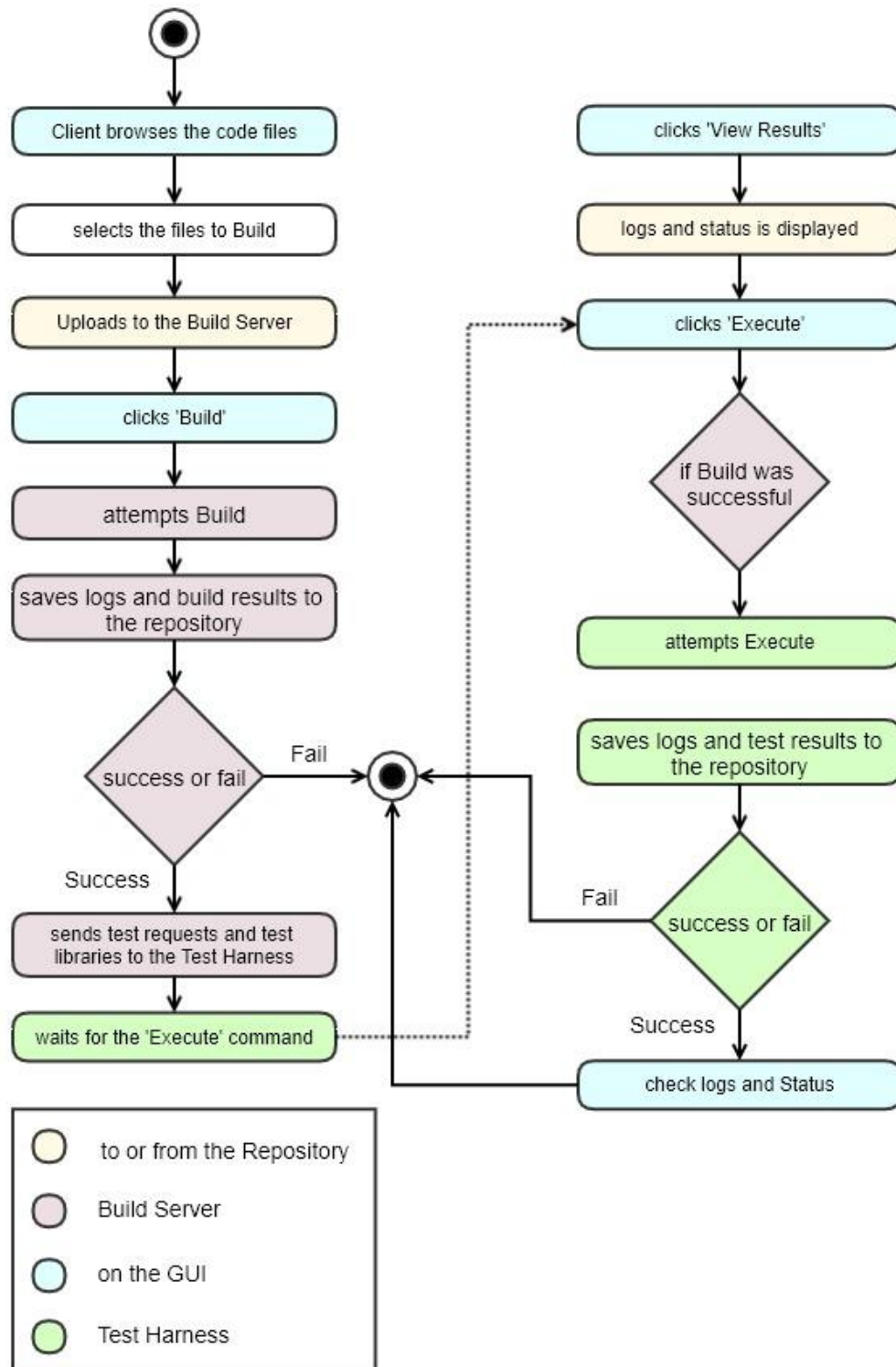
Apart from the backups a team would normally have, it will serve as another backup repository with all the latest changes. Just in case something goes wrong, then the developers can check out the code files from the main repository.

- **LOGGED DATA**

All the logs are preserved for all the build and test requests. So, in case, later if the main line code when compiled together, fails or gives any major errors; the information about who changed the code last are available at the repository.

## 4. APPLICATION ACTIVITIES:

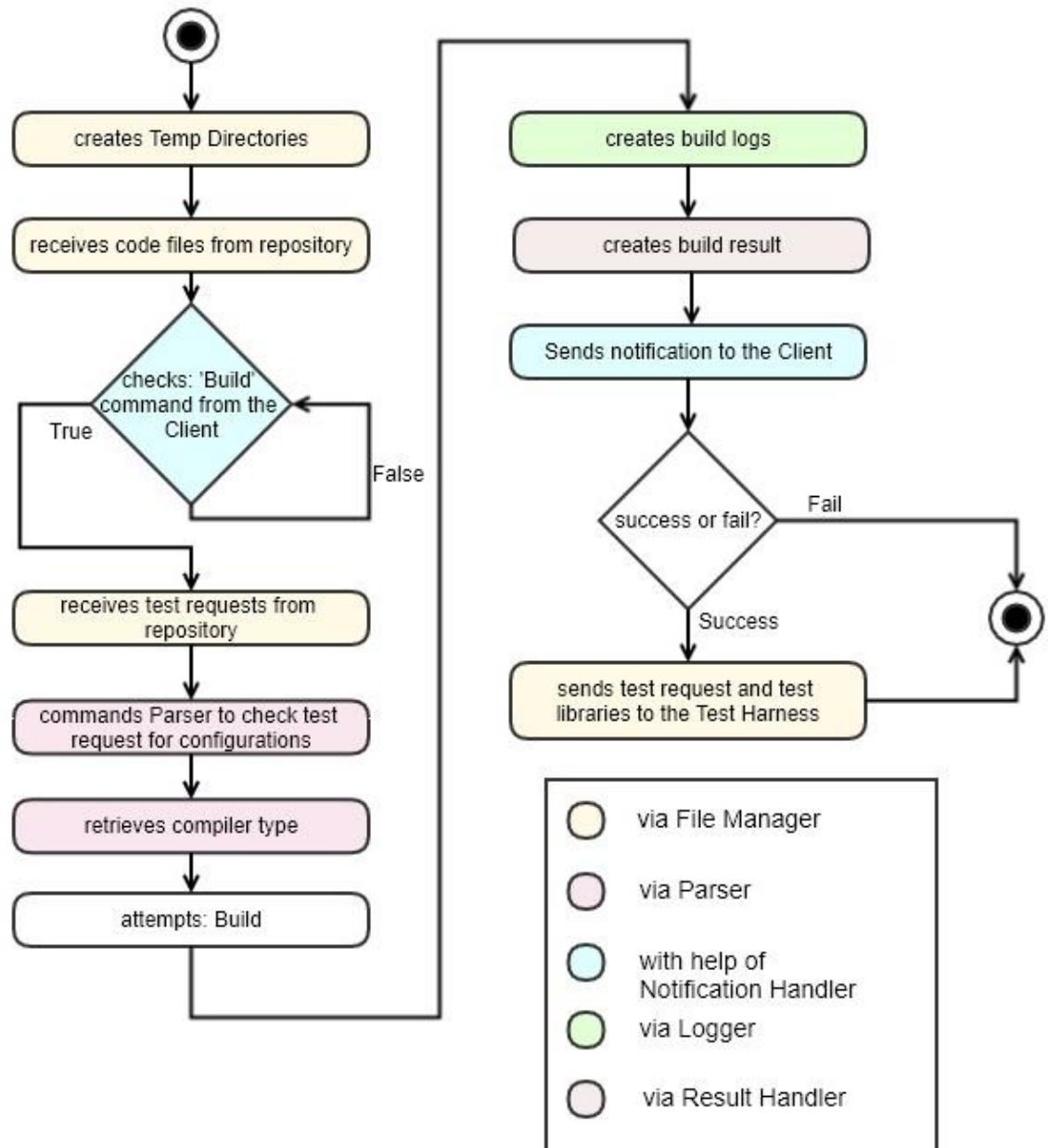
### 4.1 TOP LEVEL INTERACTIONS:



**Fig 4.1 Top Level Activity Diagram**

- The Repository Server holds the code files provided by the Client.
- The Build Server creates a Temporary Directory Structure and gets the code files from the Repository Server.
- The Client should 'Browse' the files, select which ones it wants to compile and click on 'Build' on the GUI.
- This request sends the file details to the Request Handler to create a proper Test Request file. We would be creating an XML Document for this purpose. This document will have details such as: Client information, DateTime and Test details like: Test Driver information, file dependency information, etc.
- The Test Request file is then send to the Mock Repository Server through the File Manager and is available for the Build Server to serve the request.
- As soon as the Build Server receives the 'Build' notification from the Notification Handler, it checks the Repository for the Test Request file.
- It retrieves this Test Request file and continues with the compilation process, explained in detail in the next section.
- When the Build Server has completed its task, it creates build logs and build results and stores them to the Repository with the help of the File Manager. Also, it notifies the Client about the status of the process.
- The Test Harness, too, creates a Temporary Folder, and stores the library files and test requests from the Build Server.
- When the Client is ready to Execute the libraries created, it clicks Run on the GUI and notifies the Test Harness to start the execution.
- When the Run process is completed, the Test Harness in a similar manner, stores the test logs and test results in the Repository and notifies the Client about the status.

## 4.2 ACTIVITY DIAGRAM FOR THE BUILD SERVER:



**Fig 4.2 Activity Diagram of a Build Server**

The Build Server will follow the following steps in sequence:

- Before receiving any code or request files, the Build Server with the help of File Manager, will create Temporary Directories. These Directories will be named according to the DateTime at which the Build Request files were being sent from the Repository.
- It then receives the code files with the help of the File Manager, from the Repository and saves them to the Temporary folder.

- The Build Server is now ready for the compilation process and waits for the Client to Send the 'Build' command with the Test Request files.
- It commands the Parser to parse the test request files. The Parser provides information about the configuration details of the test, the file dependencies between the code files and the compiler which should be used for setting the development environment. We will be using the default command prompt of Windows.
- It then attempts to build the files according to the information provided by the Parser.
- When the process is completed the Build Server then calls to create the Build Logs and Build Result with the help of Result Logging Module.
- It then stores the log files in the Repository Server via the File Manager, so the logs are available for the Client to view through the GUI.
- It also sends the results to the Notification Handler, so it can notify the status of the process on the GUI.
- If the Build Result is Successful, then the Build Server sends the Library File(s) and the Test Request to the Test Harness for further execution.

## 5. CRITICAL ISSUES

- *What if the build process fails and the code files and test requests are not forwarded to the Test Harness, how will the Client know that the code is not being executed?*

Solution: When the Build Server completes its attempt to compile the files, no matter what the result maybe: either Success, Failure or Success with Warnings, a status will be displayed on the GUI about the result of the build process. If the build fails, then the Client can correct the errors and re-upload the files for the build process again.

- *How will the client know which logs to check for the build failure?*

Solution: The logs in the log file will have Client information which it received from the test request file with the DateTime. Also, the logs provided will have details about on which line number did the build process fail.

- *There will be multiple clients accessing the Remote Build Server. How will we keep the client code files separated from each other?*

Solution: The idea is to store the files and requests received according to the time stamp at which it was sent by the Client. Also, we can use Blocking Queues so that no files are lost and even the reception and creating of the Directories in the Repository happen according to the Time Stamp at which it was forwarded out of the queue in the repository. By doing so, even if there are multiple clients accessing the Remote Build Server, the processing of build and execution will be simplified.

- *How will we ensure that there are no old code files present in the Build Server, so if the same files are uploaded with changes, it should provide a clean build to the Client?*

Solution: When the Build process is completed, no matter what the result, it will ensure to clear all the code files from the Directory as well as Delete the complete Temporary Directory. So, if a new request comes in, the Build Server will create a new Temporary Directory with the DateTime.

- *The code files provided by the client can be written in any language. How will the Build Server identify and compile the files?*

Solution: Our Build Server will be able to support codes written in the following languages: C#, C++ and Java. The Parser will help the Build Server to parse the XML Test Request and know which files are sent to the Build Server. The Parser will also provide the functionality to setup the build environment accordingly.

- *The dev environments are different with a Visual Studio command prompt and a default command prompt. How will the Build Server manage to switch between the environments successfully?*

Solution: There are Process and Environment classes available in .Net Framework to help setup and switch between suitable environments. We will use these classes to setup our Default Command Line Environment.

- *How will the Build Server know about the file dependencies among the code files, if any?*

Solution: According to the information provided by the Client, suitable Test Request files will be generated. These files will be parsed before the start of the compilation process and the dependent files will be linked to each other.

- *Will the Test Harness receive the Test Requests from the Repository or the Build Server?*

Solution: We can make it receive the files either way, but it will be much better if the Test Harness receives the Test Request files and the libraries from the Build Server collectively.

- *Will there be in any way, any History saved about the build logs and the test logs?*

Solution: The build and the test logs will be permanently saved in the Repository Server. They can be retrieved at any time.

## 6. CONCLUSION

Remote Build Servers are already being used by many organizations and developers for its number of benefits. It helps to provide a stable environment for the build function. The organizations can even improve their security if the deployment and migration is done only with the help of a Build Server and only authorized employees have access to it.

Therefore, by planning and creating the Operational Concept Document, I can say, that it is possible to logically build a Remote Build Server, which can be henceforth used to provide an automated and continuous integration.



## APPENDIX

### LOADING A VISUAL STUDIO PROJECT FROM A PATH AND COMPILING IT:

The loggers List is created directly using the ILogger Interface. It directly connects with the ConsoleLogger class and provides the result logs on the Console.

```
List<ILogger> loggers = new List<ILogger>();  
ConsoleLogger clogs = new ConsoleLogger();  
loggers.Add(clogs);
```

We can customize the Logging mechanism by deriving a class from the Logger Class and overriding the Initialize method in it.

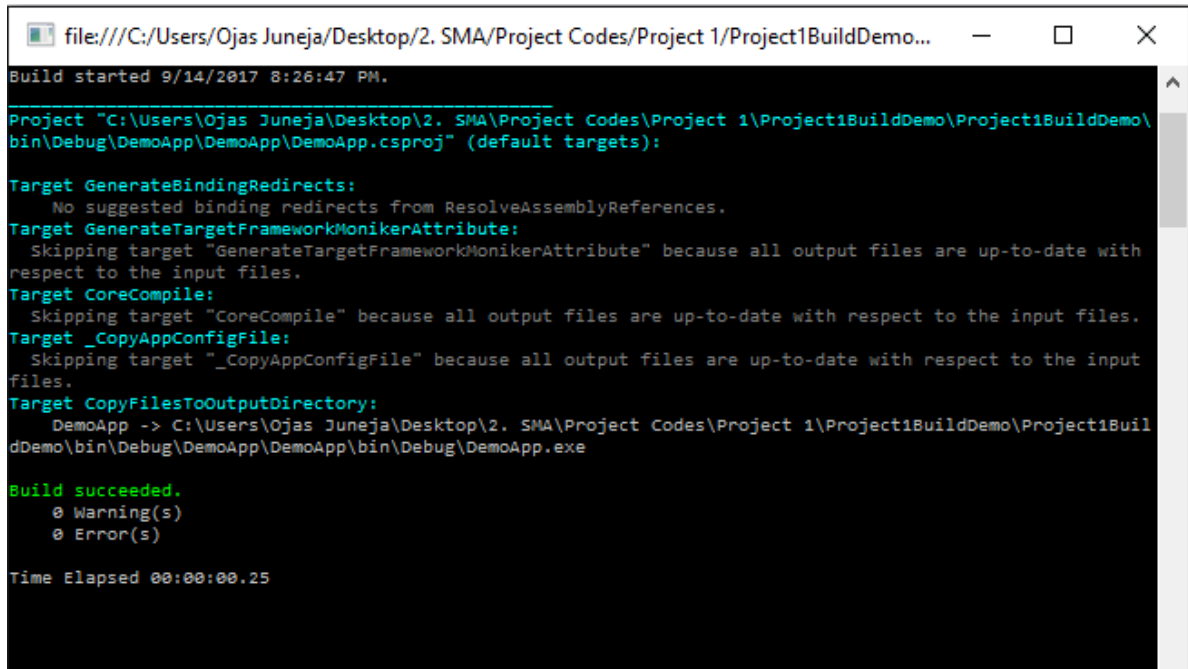
The 'ProjectCollection' Class is present in the 'Microsoft.Build.Evaluation' namespace. It encapsulates the project, its toolsets, a default set of global properties, and the loggers that should be used to build them. Apart from the 'Microsoft.Build.Evaluation', the 'Microsoft.Build.Logging' and the 'Microsoft.Build.Framework' is also needed for the compilation of the project.

We load the path of our Sample Project in the Project Collection object and call the Build() method.

```
var projectCol = new ProjectCollection();  
projectCol.RegisterLoggers(loggers);  
var project = projectCol.LoadProject(projectFilePath);  
project.Build();
```

### ***SAMPLE OF THE LOGS RETRIEVED FROM THE COMMAND LINE:***

The screenshot below, shows the logs retrieved when the Build was successful. In case the Build fails it shows the Build Failure and the errors which occurred.



```

file:///C:/Users/Ojas Juneja/Desktop/2. SMA/Project Codes/Project 1/Project1BuildDemo...
Build started 9/14/2017 8:26:47 PM.

Project "C:\Users\Ojas Juneja\Desktop\2. SMA\Project Codes\Project 1\Project1BuildDemo\Project1BuildDemo\
bin\Debug\DemoApp\DemoApp\DemoApp.csproj" (default targets):

Target GenerateBindingRedirects:
    No suggested binding redirects from ResolveAssemblyReferences.
Target GenerateTargetFrameworkMonikerAttribute:
    Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date with
respect to the input files.
Target CoreCompile:
    Skipping target "CoreCompile" because all output files are up-to-date with respect to the input files.
Target _CopyAppConfigFile:
    Skipping target "_CopyAppConfigFile" because all output files are up-to-date with respect to the input
files.
Target CopyFilesToOutputDirectory:
    DemoApp -> C:\Users\Ojas Juneja\Desktop\2. SMA\Project Codes\Project 1\Project1BuildDemo\Project1Buil
dDemo\bin\Debug\DemoApp\DemoApp\bin\Debug\DemoApp.exe

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.25
  
```

### ***SAMPLE TEST REQUEST FILE:***

The XML Document shown below is an example of how the Test Request File would be:

```

<TestRequest>
    <ClientName>Amritbani Sondhi</ClientName>
    <DateTime>13 Sept 2017 15:06:34</DateTime>
    <ClientNote>Not needed</ClientNote>
    <TestInformation>
        <TestDriver></TestDriver>
        <Tested>DemoApp.cs</Tested>
        <Tested>SampleFile.cs</Tested>
    </TestInformation>
</TestRequest>
  
```