

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.....	5
2.1 Objective.....	5
2.2 Application Obligations.....	5
2.3 Key Principles.....	5
3. Uses and Use Cases.....	6
3.1 Uses for People at Different Positions.....	6
<i>Code Developers as Clients</i>	6
<i>Quality Assurance Analysts</i>	6
<i>Project Managers</i>	6
<i>Instructors, Teaching Assistants and the Developer</i>	6
3.2 Uses at Organizational Level.....	7
<i>Frequent Check-Ins and Commits</i>	7
<i>Backed Up Code in the Repository</i>	7
<i>Logged Data</i>	7
4. Application Activities.....	8
4.1 Top Level Interactions.....	8
4.2 Activities of the Build Server.....	9
5. Interactions Between the Client and Servers.....	11
5.1 Windows Presentation Foundation (WPF).....	12
5.2 Windows Communication Foundation (WCF).....	13
5.3 Message Passing Communication with Queues.....	14
6. Partitions.....	17
6.1 <i>Client Module</i>	17
6.2 <i>Mock Repository</i>	18
6.3 <i>Build Server</i>	19
6.4 <i>Mock Test Harness Server</i>	19
6.5 <i>File Manager Module</i>	20
6.6 <i>Packages used for Communication</i>	20
6.7 <i>Other Sub-Modules</i>	21
7. Critical Issues.....	21
8. Conclusion.....	23
9. References.....	24
<i>Appendix</i>	25
• <i>Sample of the Logs retrieved from the Command Line</i>	25
• <i>Sample Test Request File</i>	25

1. EXECUTIVE SUMMARY

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

In the project development for the industry, the code varies from a few hundred lines and can go to millions of lines in a single project. This makes it necessary for the developer to keep testing his code after every 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 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.

This development will create a Remote Build Server, capable of building C# libraries, 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 Remote Build Server, which communicates with the mock Repository, mock Client, and the 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.

Critical issues include: building source code using more than one language, scaling the build process for high volume of build requests, and using a single message structure for all message conversations between clients and servers. These issues have viable solutions.

The Build Server will function as one of the principle components of a Software Development Environment Federation, the others being Repository, Test Harness, and Federation Client. Building these other Federation parts is beyond the scope of this development.

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 test requests sent by the client and store them in the Mock Repository.
- To parse the test requests when the clients selects to build.
- To get the code files required for compilation from the Mock Repository.
- To identify the files types of the received files and select an appropriate compiler tool accordingly.
- To attempt to build files and store the logs in the Repository and send the status to the Client.
- If build is successful, to send the test request to the Test Harness for execution.
- To send the generated test libraries to the Test Harness when asked for. which notifies the clients about the test results and stores the test logs in the repository.

2.3 KEY PRINCIPLES

The key principle is to automate the building and execution of the test files. The build server shall have a proper channel to communicate with the mock servers and the repository. It should also 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 codes of the following languages are expected to be built and executed by the build server: C++, C# and Java.

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:

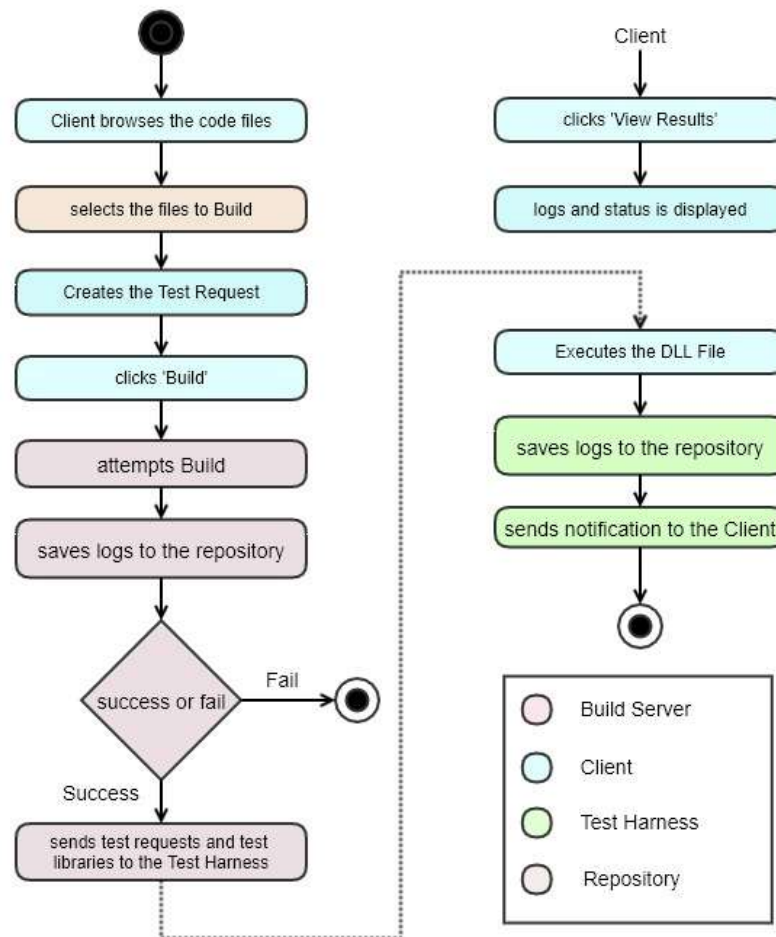


Fig 4.1 Top Level Activity Diagram

- The Repository stores the code files from which the Client creates the Test Request.
- The Build Server gets the code files from the Repository.
- The Client should 'Browse' from the list of files from the repository, select the ones it wants to compile and click on 'Create Build Request'
- The Client creates a Build Request file ie. an XML Document. This document has the details such as: Client information, DateTime and Test details like: Test Driver information, file dependency information, etc.
- The Client saves the Build Request and stores a copy of it in the Repository
- When the Client clicks 'Build' on the GUI, the Client sends a message to the Repository to forward the Build Request to the Build Server

- As soon as the Build Server receives the Build Request from the Repository, it parses the files required for compilation and asks the repository for the code files
- When it receives the required code files it continues with the compilation process, explained in detail in the next section.
- When the Build Server has completed its task, it creates the build logs and sends them to the Repository. Also, it notifies the Client about the status of the process through a notification.
- If the Build was successful, the Build Server also sends a Test Request message to the Test Harness.
- When the Test Harness is ready to serve the next execute command, it acknowledges the next test request and asks the Build Server to send the specific DLL file to execute.
- When the DLL Library is received from the Build Server, the Test Harness Loads it to its App Domain for execution.
- When the execution process is completed, the Test Harness in a similar manner, stores the test in the Repository and notifies the Client about the status.

4.2 ACTIVITIES OF THE BUILD SERVER:

The Build Server is made up of two parts:

1. Mother (Main) Builder
2. Process Pool of Child Builders

The Build Server will follow the following steps in sequence:

- When the Main Build Server is started, the communication is setup and initialized. The mother builder spawns up a no. of processes.
- While initialization the mother builder sets up two different queues too. One for serving the large no. of incoming test requests and the other for storing the ready messages to be received from the Child Builders.
- When the Child Processes are spawned, they have their own individual storages as well as communication channels. Once they are ready, the child builder sends a ready message to the mother builder, indicating that they are ready to receive a test request.
- It is the responsibility of the child builders to ask the repository for the code files, get the files and compile them. After the build process is done, the child builder creates and saves the build logs as well as sends them to the repository. It also sends the result status of the build to the Client.

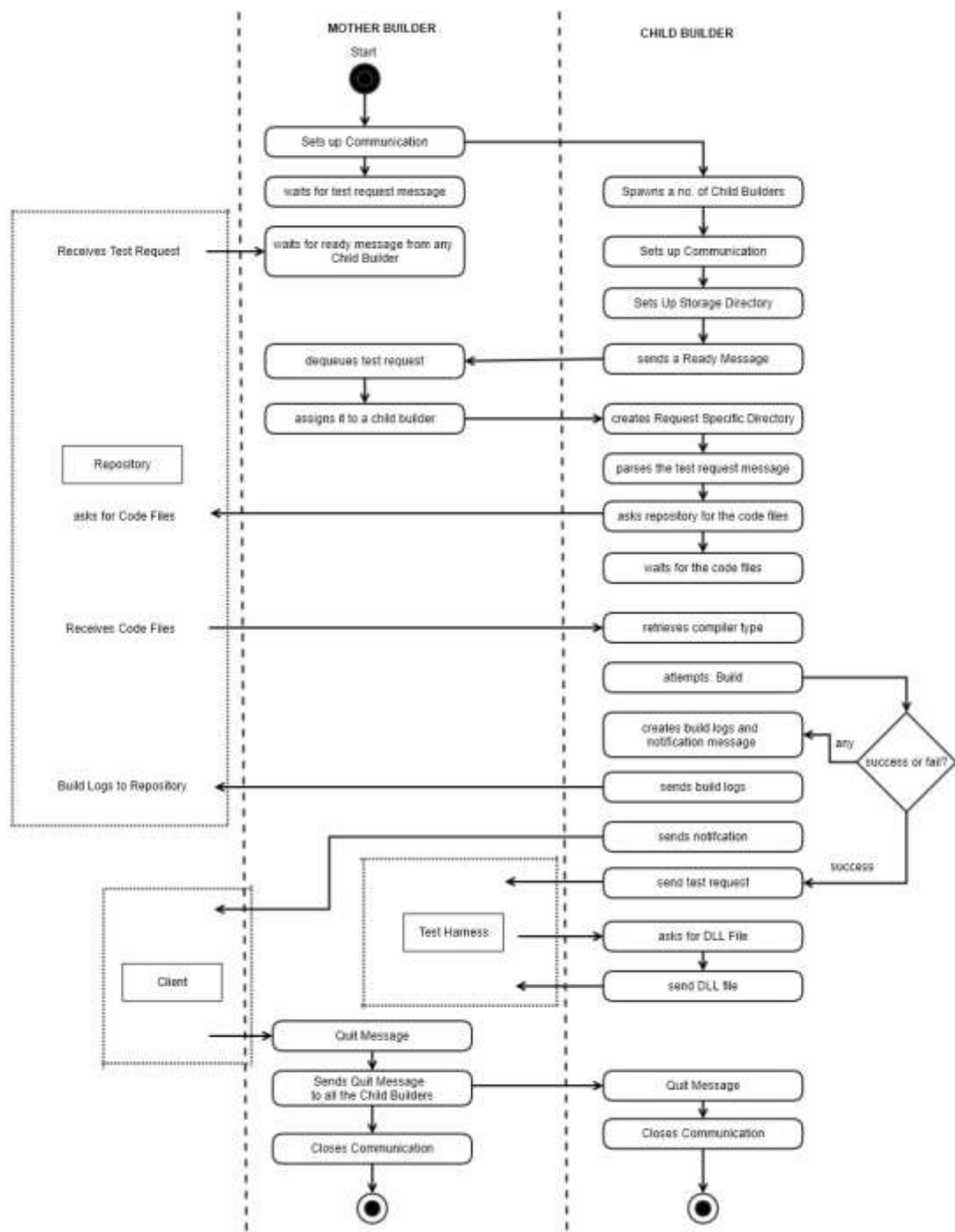


Fig 4.2 Activity Diagram of a Build Server

- If the Build Result is Successful, then the child builder sends the Test Request to the Test Harness for further execution.
- Also, the child builder expects to get a 'dllRequest' message from the Test Harness. It should then reply the test harness with the specific dll file for which it asked.

- When the child builder is done with fulfilling all it needs to take care of, it creates a ready message and sends it to the mother builder, indicating that it is ready to receive another test request.
- The mother builder doesn't necessarily have any storage of its own.
- It is responsible for maintaining the ready and the requests queues and sending the test requests to the child builders for processing.

5. INTERACTIONS BETWEEN THE CLIENT AND SERVERS:

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

- Client to the Repository Server, to the Build Server and Test Harness.
- From Build Server and/ or Test Harness to the Repository and to the Client.

The servers and the client are not necessarily in the same premise. They can be operating from different machines or locations. Here is where the communication comes in. Let us discuss in detail how the Builder Server is expected to communicate things to and from the client as well as in between the servers.

5.1 WINDOWS PRESENTATION FOUNDATION (WPF):

Windows Presentation Foundation is used for developing the User Interface between the client and the Server. The following are some of the reasons of using WPF in this application.

WPF is a Graphical User Interface technology that replaces WinForms. WPF separates the appearance of a user interface from its behavior. The appearance is generally specified in the Extensible Application Markup Language (XAML), the behavior is implemented in a managed programming language like C# or Visual Basic. The two parts are tied together by databinding, events and commands.

WPF is included in the Microsoft .NET Framework, so it can build applications that incorporate other elements of the .NET Framework class library. The controls in WPF most often detect and respond to user input. The WPF input system uses both direct and routed events to support text input, focus management, and mouse positioning.

WPF provides a data binding engine which simplifies application development. The core unit of the data binding engine is the Binding class, whose job is to bind a control (the binding target) to a data object (the binding source).

The Graphical User Interface of the Build Server will look somewhat like this:

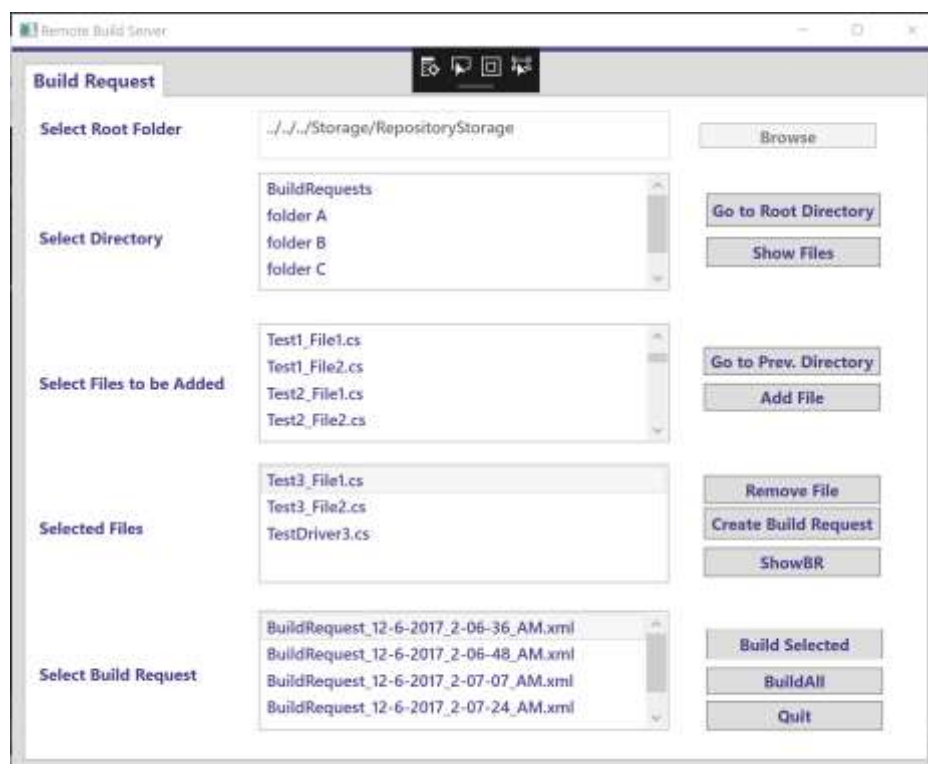


Fig. 5.1 Graphical User interface of the Build Server

5.2 WINDOWS COMMUNICATION FOUNDATION (WCF):

If we use a communication framework like Windows Communication Foundation (WCF), then each message will consist of a Simple Object Access Protocol (SOAP) wrapper around a serialized instance of a data class that defines the request, the 'to' and 'from' addresses, and any parameters needed to execute the request.

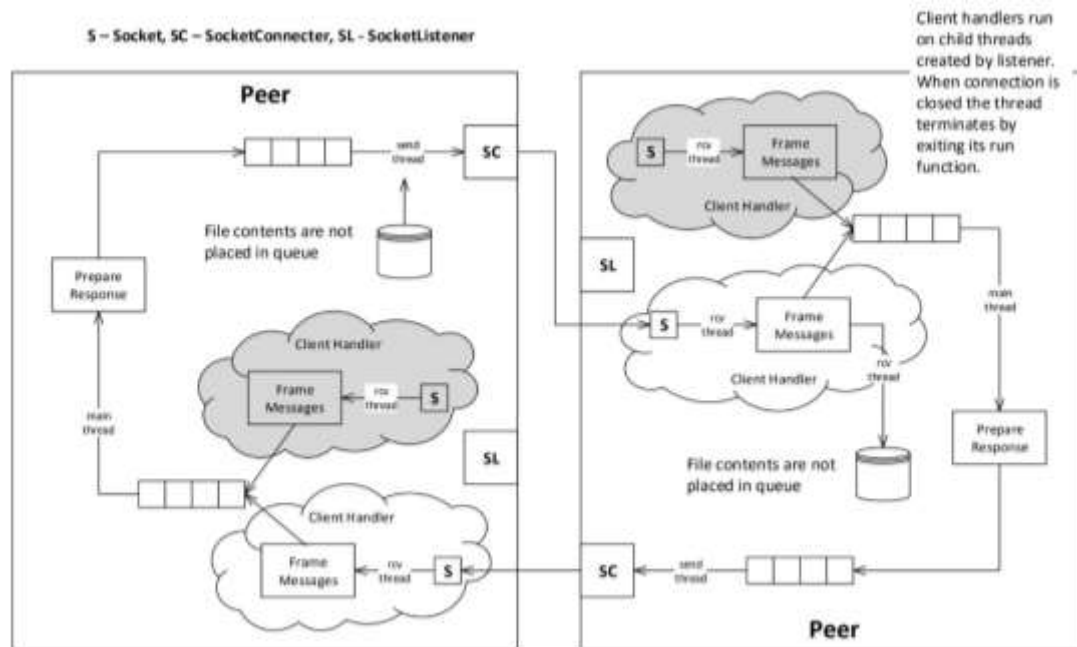


Fig 5.2.1: Communication Channel Structure

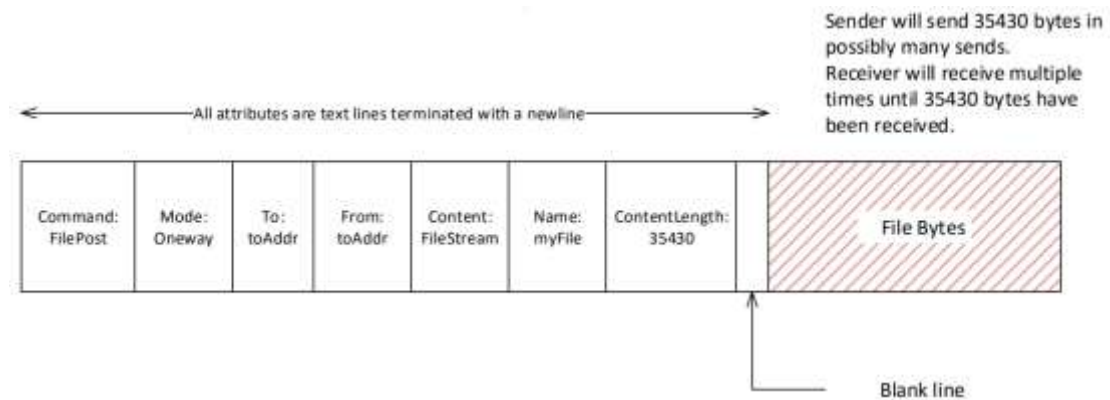
Each Receiver instantiates a WCF service. The service objects simply enqueue incoming messages, for processing by the dequeueing thread. If a reply is appropriate the message processor builds a reply message, using the incoming message return address and passes it to its sender, usually with a `PostMessage(msg)` invocation.

MESSAGE QUEUES:

Each receiver has a thread-safe blocking queue that is shared by all senders to that endpoint. If we are using WCF, we make the queue a static member of the service class so that every service instance shares the same queue, e.g., each sender to that endpoint gets a service instance and the service simply enqueues messages for the endpoint's processing thread to dequeue and process.

FILE TRANSFER:

For sending a file between endpoints we could send blocks of bytes from the file in a sequence of messages, but it would be somewhat more efficient to send a beginning message that identifies the file name, length, and block size, and then a sequence of blocks of bytes, perhaps followed by a terminating message.

**Fig. 5.2.2: Message Structure**

Note that, while performing a file transfer, the service objects will simply write incoming blocks into a file, but not post them to the receive queue. When transfer is complete it posts a message to the queue, so the message processor knows a file has arrived.

5.3 MESSAGE PASSING COMMUNICATION WITH QUEUES:

Message-Passing Communication (MPC) establishes a channel between processes to communicate by sending messages.

CHANNELS:

There are one-way message channels between a sender on one peer and a receiver on another, as shown in the diagram below. Each of these "peers" might be located on separate machines or in separate processes in the same machine. Each peer contains a sender and receiver package. The sender communicates with one receiver at a time. The receivers each handle concurrent senders by accepting messages in a receive queue. Often those messages are processed sequentially by a single thread. This means that the receiver and consequential processing do not have to be thread safe, if the receiving queue is thread-safe.

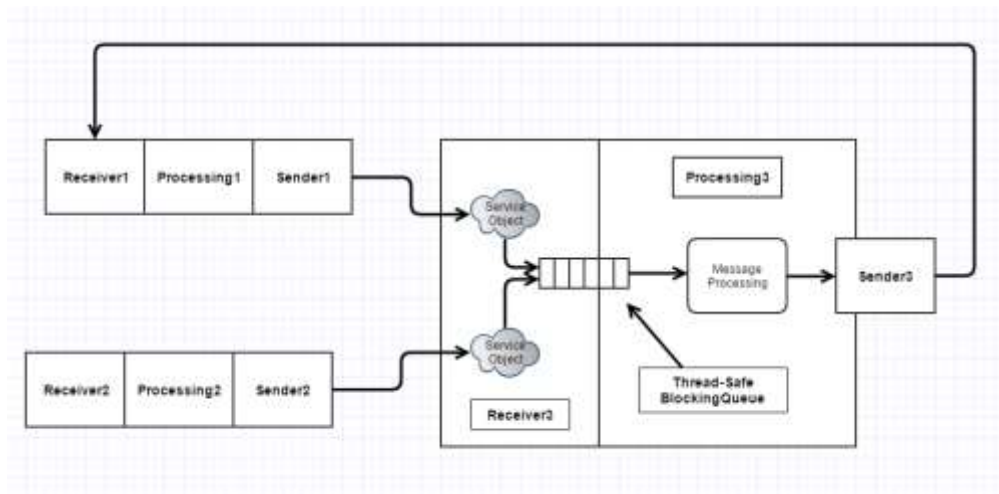


Fig 5.3.1 WCF Message-Passing Communication

The Receivers are wrappers around a WCF Service. Senders are wrappers around a proxy for the WCF Service.

ENDPOINTS:

Each receiver has a listener assigned to a specific port, and we describe the ip-address and port number as an endpoint which we represent as an attribute pair ip : port.

For example: <http://localhost:8070/IPluggableComm>

The endpoint is an address for a service provided by a peer. Note that the diagram emphasizes that the communication endpoints are identical, even though the processing that is provided at an endpoint may be quite different from that at another endpoint. The communication endpoints are peers, but the machines are not.

TALK PROTOCOL:

The resulting talk protocol is very simple because messages only flow one way in each channel. A sender can send to an endpoint at any time, and a receiver can handle enqueued messages at any time. This makes a very flexible and fluid style of communication. The sender does not wait for a response. After sending a message it may send messages to other peers or do other processing. Eventually the receiver of the message may elect to send back a reply message to the sender, but is not required to do so.

MESSAGES:

Messages contain a destination address so that the sender can connect to that endpoint. If the receiver will eventually reply, then, since there will be multiple senders, the message needs a return address. The message also needs to define the requested operation and provide any parameters needed to carry out the requested action.

Name	Source	Destination	Purpose	Contents
FileRequest	Client	Repository	Get file list	Command
FileList	Repository	Client	Return file list	File list
BuildRequest	Client	Repository	Send XML string	BuildRequest
SendBuildRequest	Client	Repository	Command	Command
BuildRequest	Repository	BuildServer	Send XML string	BuildRequest
BuildRequest	BuildServer	ChildBuilder	Send XML string	BuildRequest
Ready	ChildBuilder	BuildServer	Notification	Ready status
FileRequest	ChildBuilder	Repository	Get file	File name
File	Repository	ChildBuilder	Send file	File contents
BuildLog	ChildBuilder	Repository	Send build log	Log string
TestRequest	ChildBuilder	TestHarness	Send XML string	TestRequest
FileRequest	TestHarness	ChildBuilder	Get file	File name
File	ChildBuilder	TestHarness	Send file	File contents
TestResult	TestHarness	Client	Notification	Test status
TestLog	TestHarness	Repository	Send test log	Log string
Complete	TestHarness	ChildBuilder	Notification	Complete status

Table 5.3 Communicated Messages

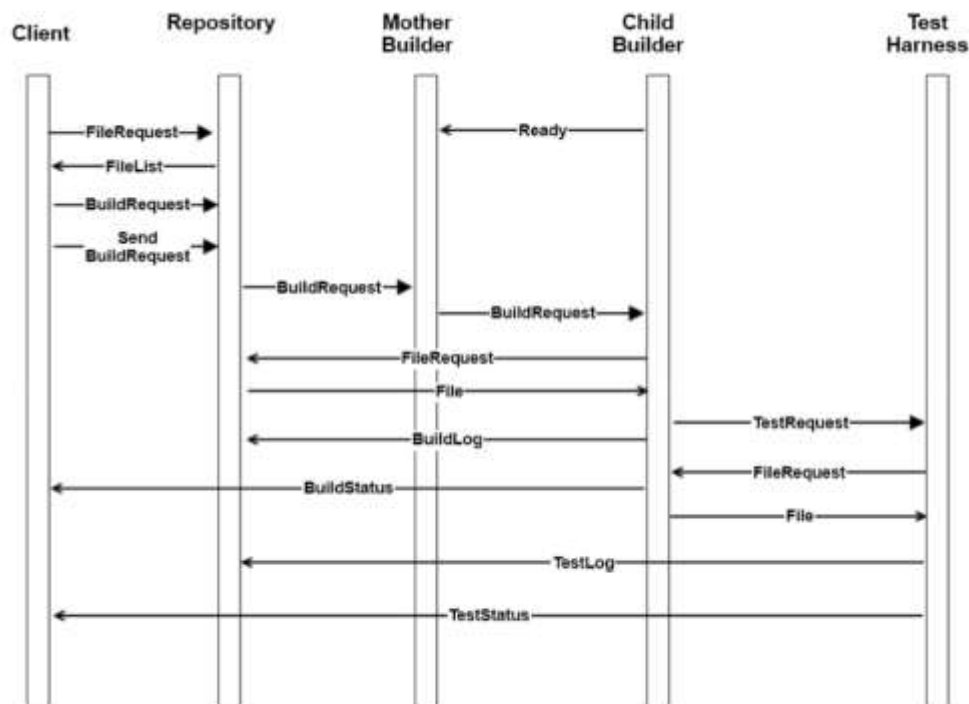
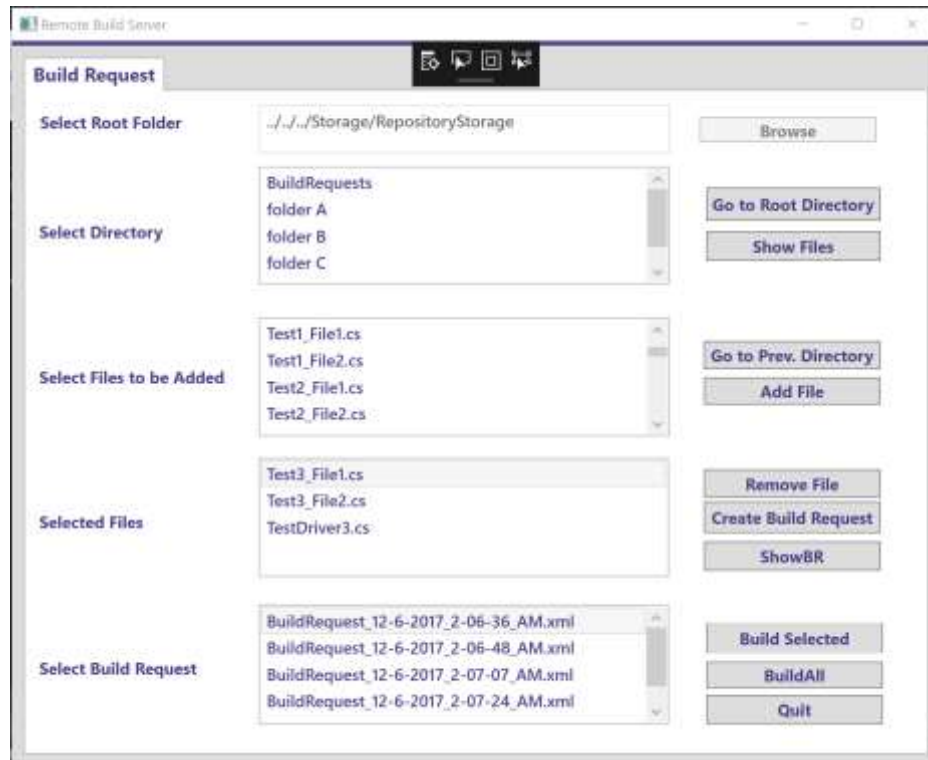


Fig 5.3.2: Message Flow Diagram

MAIN WINDOW:

The 'MainWindow' Package provides the functionality of creating an interactive UI and provides the event handling and routing on the client side. It looks something like this:



REQUEST BUILDER:

The test requests are supposed to be created on the client side according to how the client wants them to be. Hence, the Request Builder creates all the test requests using a same template. It is better to have a single template so parsing it in the other servers is easier.

6.2 MOCK REPOSITORY:

This acts as the main Repository which stores all the code files, test requests, and build and test logs. The Mock Repository Server receives test requests from the clients. When the client notifies to build, the repository parses the test requests sent by the client through the message. It modifies the same test request and sends it to the build server for compilation. The repository stores the build logs and the test logs received from the Build Server and the Test Harness after they have completed the compilation and/or execution processes. It is also responsible to send the list of code files present in the repository to the GUI client, so it can select the code files to create a test request.

STORAGE MANAGER:

Storage Manager is used by the repository to provide different storage spaces for Code Files, Test Requests and Log Files.

6.3 BUILD SERVER:

The Build Server, has the main responsibility of compiling according to the received test request. It however creates a no. of child process to fulfill this responsibility. The main build server, receives build requests from the repository which it stores in a request queue. This is done so that it can handle large no. requests without having to miss on any of them.

CHILD BUILDER:

The responsibility of the Child Builder is to receive the test request from the mother builder and ask for the code files from the Repository. Before receiving any files from the Repository, it creates test request specific Directory to store the code files, test request and the library in one place. When it receives the code files, it then parses the test request for selecting 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 logged results to the repository. If the build is successful, then to send the test request to the Mock Test Harness for executing the generated library file. It also sends a status of the build process to the Client. In case the build fails, the build server must only notify the client about the failure and send the log file to the Repository.

BUILD CONFIG:

The Build Server uses 'BuildConfig' package to parse the test request and select which compiler is to be used for the process. The Build Config 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.

6.4 MOCK TEST HARNESS SERVER:

The Test Harness, receives the test requests from the Build Server after the build process is successful. It then asks the builder for the Dynamic Link Library (DLL) generated by it. As soon as the DLL is received, it loads the libraries and attempts to run the dlls and the test drivers. After the execution gets completed, it sends the log files to the repository. It also notifies the Client about the status of the execution.

APP DOMAIN MANAGER:

This package is used to load the dll library in a separate app domain for testing. According to how the test request was built, the app domain will use the C# or the C++ tester for executing the libraries.

6.5 FILE MANAGER MODULE:

This module is used for all the files and directory related processes. It will have methods which will allow us to pick up the files from the Server's physical storage and send the files or their details which will be used by the Repository, Build Server and the Client. The File Manager will also keep a track of the sent files details in its stack so we can retrieve any information of the previously processed request or storage.

6.6 PACKAGES USED FOR COMMUNICATION:

For implementing message passing communication described in the previous chapters, three of the below packages are used:

IMPComMSERVICE PACKAGE:

The IMPComMService Package is an interface which provides WCF capabilities to any packages using it for establishing communication. It provides the Service and Operation Contracts which can be used while communicating between servers.

It also provides the DataContract functionalities which is used for exchanging messages between the servers.

MPCOMMSERVICE:

The MPComMService package derives from IMessagePassingComm package. It defines the Comm class, which provides all the required objects for the communication to the Server. The Comm has classes which provide the Sender and the Receiver functionalities explained in the Message Passing Communication section above.

BLOCKING QUEUE:

The Blocking Queue provides a queue structure for the Sender and Receiver to use. The Receiver uses it when multiple comm objects are trying to send it to the single receiver. The receiver just queues the messages on first come first served bases for giving it to the server. The sender before sending any messages, sends them through the queue.

6.7 OTHER SUB-MODULES USED IN THE PACKAGES:

- **MESSAGE REQUEST HANDLER:**

This sub-module is used in all the packages where communication is set up. The message request handler is supposed to provide the functionality of how different messages will be handled in the Server. It will redirect the message received to its respective methods.

- **LOG AND NOTIFICATION HANDLER:**

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 Harness 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.

The Notification Handler creates all the notification messages which are being passed from the Build Server and the Test Harness to the Client. It gets the result of the execution of the Build or Test and notifies the Client via a message.

7 CRITICAL ISSUES

Critical issues for developing the Remote Build Server are:

- *Issue #1:*
How to define a single message structure that works for all messages used in the Federation.
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.
- *Issue #2:*
Building and testing both C# and C++ code.
Solution: For building, set environment variables as demonstrated in Help Session demo, and use tool chain commands for each type of source. For testing, trap exceptions on loading native code libraries in the C# Test Harness and direct to C++ Test Harness, as demonstrated in class.
- *Issue #3:*
Managing EndPoint information for Repository, BuildServer, and TestHarness.
Solution: Store Endpoint information in XML file resident with all clients and servers and load at startup.

- *Issue #6:*

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 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.

- *Issue #8:*

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.

- *Issue #10:*

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.

- *Issue #11:*

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.

8 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.

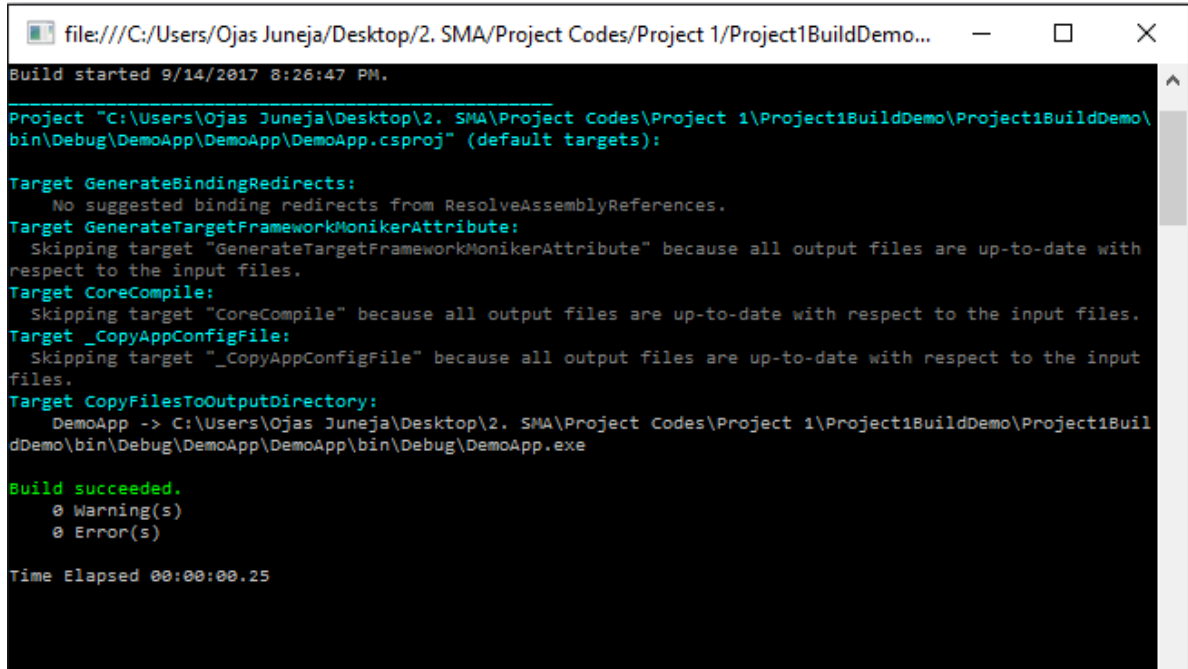
9. REFERENCES:

- Dr. Jim Fawcett's website and blog:
<http://www.ecs.syr.edu/faculty/fawcett/handouts/webpages/CSE681.htm>
- <http://www.wpftutorial.net/HelloWPF.html>
- [https://msdn.microsoft.com/en-us/library/aa970268\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/aa970268(v=vs.100).aspx)

APPENDIX

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>
  
```