REMOTE BUILD SERVER

OPERATIONAL CONCEPT DOCUMENT
CSE681-PROJECT 4

By Vishnu Prasad Vishwanathan
(SUID: 793782749)
Instructor: Jim Fawcett
Date: 8th December 2017

Table of Contents

Table of Figures

Remote Build Server

1. Executive Summary

The Remote Build Server is a Federation of servers helping us to efficiently run big software systems by partitioning the code into small parts and automating the tests on those small parts. In complex systems, we would have million lines of code in thousands of packages.

For a successful implementation of such big complex systems we need to partition the code into small parts and test them thoroughly before adding them into a software baseline.

The Remote build server helps us to ease this process. It helps us in automating the test process for the whole baseline so that human intervention is minimal during this process, as small as a button click for the whole system testing. This helps in the continuous integration of codes in a system because when a new code is created for a system it builds and tests the code, its calling and called codes. Based on the result of the test we proceed, if it passes then we check in the code, so it is successfully inserted to the current software baseline else notify the user about the failure (build or test).

We have to make sure that the primary goal is achieved which is minimizing the human work. We can consider this as a risk in the development because the user shouldn't experience any difficulties in using the Remote Build Server at any point. The user interface should be simple and effective. The proper communication between various federation servers is also a risk.

The Client, Repository, Build Server, Test Harness, Mother Builder are the different parts expected to in this Remote Build Server to carry out the above-mentioned services efficiently.

2. Introduction

This Operational Concept Document is to put forth our ideas for efficiently designing and implementing the Remote Build Server which provides services like automated build and test, continuous integration for big complex systems with thousands of packages and million lines of code.

The Remote Build Server as mentioned above provides automated build and test and if the test is passed then it is checked into the repository. It is structured based on the following critical parts: Repository, Build Server, Test Harness, Client. As mentioned above big systems have million lines of code. In order to implement those systems, we need to partition the code into small parts and test them efficiently and put it into the software baseline. The Remote Build Server has the Repository that holds the codes that are tested and checked in to the current baseline.

The final result (which is Project #4) has a message communication system between the different parts, each part with its own set of functionalities. The parts are efficiently partitioned, and they stand alone and perform their own functions by receiving the requests via communication channel. Basically, these Federation servers are dedicated to the primary services of the Remote Build Server.

When a new code comes, or any change is made to the existing code in the baseline then we automate the tests on the entire baseline. The Test Requests are issued to the Build server which receives the request and builds the necessary packages based on the request. When the build succeeds the test request along with the necessary files is passed to the test Harness which obtains the necessary packages as build server did and tests and passes the result to the user.

The user has to interact with the system and that is the client interface. The user should be given the minimum work at any point of time which is the ultimate goal of the Remote Build Server. The communication between the servers should be efficient when it comes to displaying the results after the build or test.

In Project 2 we concentrated on the Build Server's design and implementation. At the end of project 2 we made sure that the Build server does what it should do. We did not implement the message passing communication or Process pooling. we developed mock repository, mock test harness, mock client as packages in the solution so as to develop initial prototypes for these final components of remote build server

In Project 3 the prototypes for the servers were developed along with the prototype for the message passing communication. This acted as a skeleton for the final project where we developed a fully functional Remote Build Server (Project 4).

2.1 Application Obligations

The primary goal is to support continuous integration for large systems by carrying out consecutive builds and tests. It should enhance performance, reduce the time in testing process in parallel way supporting security and scalability.

We focus on these tasks in our main application:

1. Interaction to application via user friendly GUI
2. Flexible build request creation by user
3. Timely notification sent to user via GUI
4. Maintaining all code and build request files in repository system thus supporting successful file manager system
5. Developing mother builder functionality
6. Supporting multiple child builders
7. Supporting simultaneous and continuous builds using mother builder
8. Maintaining a process pool for these packages thus each package runs on a separate process
9. Building of given build requests into **DLLs**
10. Flexible file transfer between repository and other packages
11. Supporting various toolchain (c++ ...gcc, java. java , c# csc)
12. Generating build logs and test logs
13. Displaying the logs to GUI for the user to check
14. Running test harness constantly so that multiple child builders can test their built files
15. Support use of robust message passing communication channel using WCF

2.2 Organizing Principles

The organizing principles are to execute the key functionalities of remote build server system by the use of repository [file manager system], mother builder, child builders [core builder], test harness [testing hub], WCF [message passing channel] and process pooling [running all the packages on separate processes]

2.3 Key Architectural Ideas

The Key idea in Remote Build Server is to support continuous integration for the large systems, our application will be developed in c# language with .NET framework 4.6 and Visual Studio 2017. GUI sends build request for particular code files. Repository Holds all code files, request files, log files for the current baseline, the mother builder batches build requests to child builders, the child builders build test libraries for submission to the Test Harness. Test Harness runs tests, concurrently for multiple users, based on test requests and libraries sent from the child builders.

3. Actors

The following type of users can interact with the primary adaptation of Build server system.

3.1 Developer's Repeated Testing

The developer has to test the entire baseline every time when something is added or if the existing code is modified because of the code dependencies. So, it would be better to have the baseline divided into small parts and an efficient way of testing it reducing the human work.

Design Impact:

The Remote build Server shall support in running the tests over the packages of the baseline automatically. If the user provides the required test cases as a request package, then the system should be able to load the required packages and test cases for testing them. The design should automate this testing by automatically reading the request, getting the code, building and testing it and finally sending the results and logs to the user and repository through message passing channels. Depending on the requirements we should be able to add features like multiple testing servers. The system that we develop should be robust. The developer should be able to run the tests overnight and the results of all the tests should be shown to the developer in a sophisticated way. One way of making the development environment look better is to have a separate Repository for each developer.

3.2 The Manager and Customer level satisfaction

During the development phase the client has to be updated on a regular basis. One of the main reasons for making the Operational Conceptual Document is that the development phase has to be planned in such a way that we show proper form of progress so that the customer is satisfied at any point. The Program Manager who is responsible for delivering the product should be satisfied as well. He is the starting point of presenting the project or its parts.

We are going to talk about process pool which is a limited set of processes spawn the startup.

Design Impact:

After preparing the OCD, we should provide a considerable prototype that showcases the basic system which is build server. The second step is to make the Build Server working which is the important part, so it needs to be tested thoroughly. At the end of this step the Build Server should be robust.
The third step is to create the prototypes for the Client interface (Graphical User Interface), Process Pools and message passing communication along with other additional packages needed for the Final Remote Build Server.

The Process pools have the same functionality of the Build Server that is developed and tested in the second step. If the steps are carried out as planned above, we believe that we would have considerable results to satisfy the customer.

The fully developed Build Server would give the ease of access for the higher authorities as it takes minimum time to start the process or to view the results from build or test. Generally, the logs are the important parts for the managers to view so making the repository understandable and easy to navigate might be of a greater use. Even though the build server and test harness are developed properly, the communication channel plays a major role here. The user will know the functionalities better only if the logs and results are communicated properly to the client and repository.

The client server is the way through which the user communicates with the Build Server system and as mentioned before the user should get the information just by clicking a button. The User interface should be easy enough to understand and use as the beauty of the design is that all the complexities must be hidden from the user point of view and they should be able to provide and receive the information without any complexities. The complex systems are built to reduce the human work. The scheduling of the tests and the efficient displaying of the various test results are to be taken care of.

3.3 Quality Assurance

QA maintains the quality of the product by performing a lot of tests on individual components of the system and on the whole system too. If they have a medium for automating those tests and if they can get the results in a clear, understandable way then a lot of time can be saved. Generally, the QA would not have the complete knowledge about how the project is developed so they need to be provided with the correct test cases and packages.

"We are going to talk about process pool which is a limited set of processes spawned at the startup."

Design Impact:

A Repository is maintained for a System baseline. The repository should be designed in such a way that the user can efficiently access the components to be tested in such a way that the efficiency is maintained even in relatively big systems while accessing the repository or communicating the results and logs from build server and test harness. We can have a repository for the developer which can be cloned for the use of QA so that we create a backup at each point. In that case the updates should be taken care of and the version number too. It is easy to copy but the challenge is synchronizing them (which should be done in regular intervals).

"A software baseline consists of all the code that we currently consider being part of the developing project, e.g., code that will eventually be delivered as part of the project results. It does not include prototypes and code from other projects that we are examining for possible later inclusion in the current project."

4. Advantages and Uses

One of the primary features of Remote Build Server is Continuous Integration. Continuous Integration is a way to increase code quality without putting an extra burden on the developers. Tests and checks of your code are handled on a server and automatically reported back to you. We will discuss some of its uses:

4.1 Run your tests anywhere anytime

The Continuous Integration Servers can help running the tests wherever we want. If the test multiple machines, then if it runs without any errors in all of them we are clear. If it with fails anywhere then we have to look into it. The multiple tests are done efficiently using CI servers.

4.2 Production Deployment

You can have the CI server automatically deploy your code to production if all the test within a given branch passes without any failures. This is what is formally known as Continuous Deployment. The production deployment delays or failures are drastically reduced here.

4.3 Code coverage

Every time the test is done on a piece of code we have to make sure that all the test cases are covered. If the manual testing is done, we have very high chances of missing at least one of them. The CI helps in this case, all the tests are automated so that every time the same test cases are run. The complete code coverage is guaranteed to the users. The code coverage is one of the important aspects when it comes to test efficiency.

4.4 Build stuff now

CI servers can also trigger build and compilation processes that will take care of your needs in no time. No more having to sit in front of your terminal waiting for the build to finish, only to have it fail at the last second. The CI server will run this for you within its scripts and notify you as soon as something goes wrong.

5. Application Structure

The structure of the final Remote Build Server is discussed here along with the prototypes and Core Build Server developed in the development phase. The block structure shows the basic structure of our final application. All the packages the GUI, the Test Harness, the Repository, the Child Builders interact via **comm** channel which has a message dispatcher.

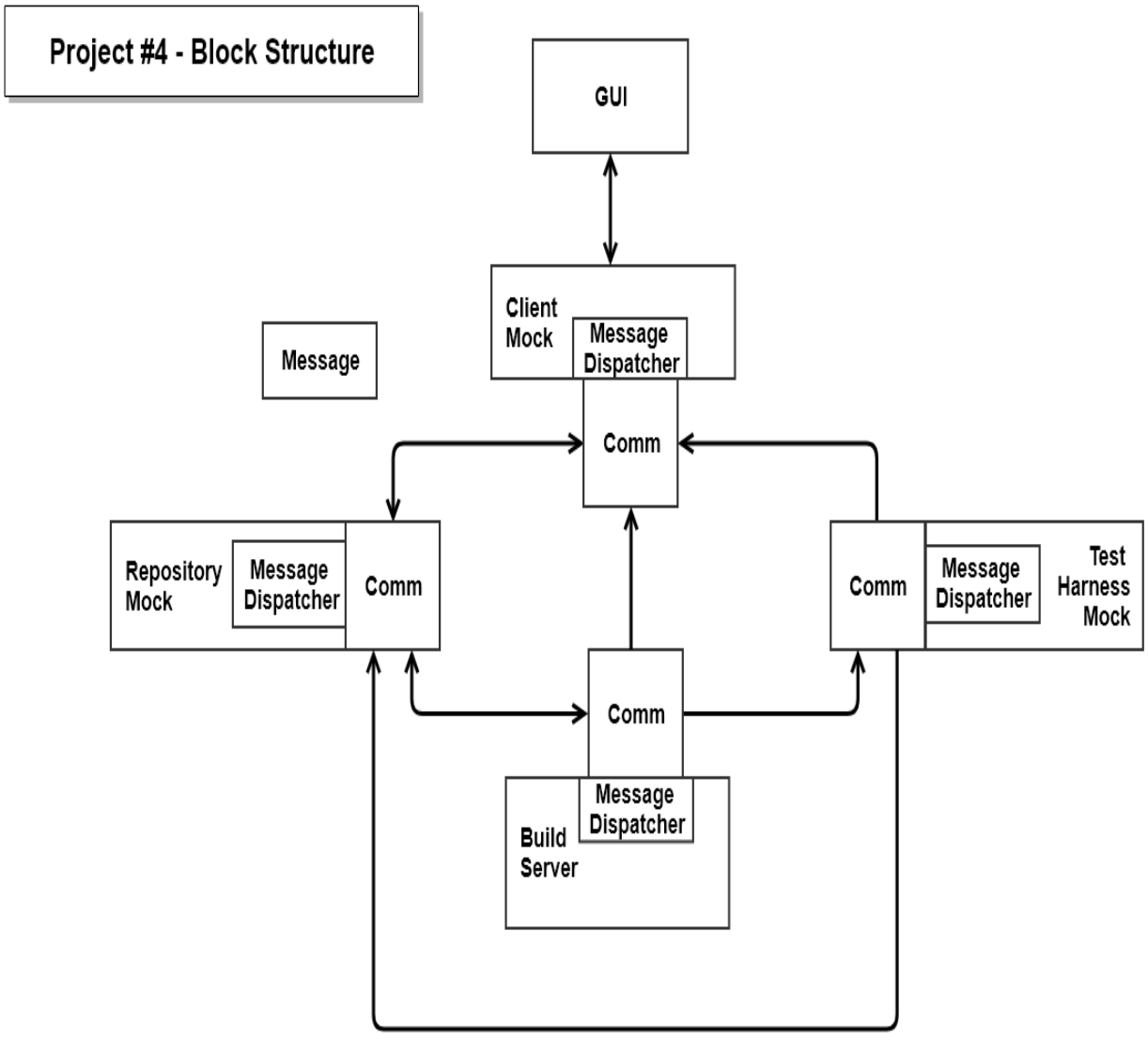


Figure 1 Block Structure for Remote Build Server

.

5.1 Repository

The packages of the repository run on a different process in our system. The repository is used by mother builder, GUI, Child builders and the test Harness. Repository is communicated by message passing channel by all the modules. It's the primary file manager of our system.

5.2 Client/ GUI

The GUI is the first and foremost point of communication of our application as it is the user interface. The GUI helps in selecting the files for build process, sending the build command to repository, Showing the logs of process and the status of build/ test process.

5.3 Test Harness

The test harness is the package which Runs tests, concurrently for multiple users, based on test requests and libraries sent from the child builders. The Test Harness executes tests, logs results to repository, and notifies the GUI of the tests of the results.

5.4 Child Builder

The child builder is our core builder responsible for building the files selected by user into dlls. It takes the files from repository and builds them. It further sends the build logs to repository and notifies the GUI about the build results.

5.5 Mother Builder

The primary build server which maintains a Pool Process of child builders thus establishing coordination of repository to these builders by a well-versed communication channel. It batches the build requests to child builders.

6. Module Structures

The following are the primary tasks that should be provided by the system Remote Build Server.

6.1 Repository

The repository stands as a standalone server in the Remote Build Server. But while developing the Core Build Server we use the mock repository which stands as a package in the same solution.

Store the software baseline in a Repository folder as packages which support:

1. Addition of new code files
2. Storing the build requests
3. Creating build request for the files selected by user
4. Parsing the User's build Request on command by sending it to Mother builder.
5. Store the build logs received from the child servers.
6. Store the test logs received from Test Harness.
7. Send the files to the child builder on command.
8. Send the logs stored to the UI on command.

6.2 GUI

The application has the standalone server supported by Graphical User Interface using the Windows Presentation Foundation (WPF) which:

1. Allows the user to browse the repository and select the required files for the build process
2. Allows user to command repository to create build request for the selected code files
3. Allows user to browse from already existing build requests
4. The Interface would display the build and test logs sent from the repository on command
5. The Interface would display build and test status results sent from the repository on command

6.3 Test Harness

The Test Harness is the final part of the Remote Build Server. As the name suggests it is the testing hub for the **DLL** files built by child builder, thus performing:

1. It accepts the test request from child builders
2. It parses the test request
3. It loads the libraries from child builder
4. It executes the tests on command.
5. it sends the test logs to the repository.
6. It sends the results to the client UI.

6.4 Child Build Server

The application has multiple child builders which run on different processes to separate simultaneous building and testing:

1. Accept the build requests from the mother builder and parse the request to know the files to be extracted from the repository and understand the test configuration.
2. Create a temporary folder before getting the files from the repository.
3. Send the file request to get the required files from the repository.
4. Accept the files from the repository and store them in the temporary folder. We have to look at the parsed test request to know which tool chain we have to use (Compilers of C++, C# etc.).
5. Attempt to build these code files into **DLLs**.
6. Create build logs and send it to the repository along with the results.
7. If the build fails, then notify the client.
8. If the build succeeds send the test request and library that contains the required files to Test Harness.
9. Remove the temporary library and command the Test harness to proceed with the test.

6.5 Windows Communication Foundation (WCF)

WCF is the internal component of our application. It functions to support seamless message passing between various modules. the packages such as repository, child builders, mother builder and test harness all communicate with each other via message passing using WCF

The message passing communication channel supported by Windows Communication Foundation(WCF) would be implemented for passing files between the servers in project #4, thus carrying out:

1. Provide communication services to application
2. Helping in file exchange via messages
3. Helping in message exchange between packages

6.6 Mother Builder

The application has a mother builder which is the paramount builder dealing with build requests and child builders:

The main tasks of mother builder are-

1. Maintains a queue of build request XMLs sent from repository
2. Maintains a queue of ready messages sent by child builders
3. Successfully assigning the build request to most recent child builder, thus allocating the build requests to respective child builders
4. It maintains a process pool of these child builder processes and repository process
5. Successfully coordinating with WCF message passing system

6.7 Process Pool

Process pool is maintained by mother builder so as to efficiently carry multiple build process at once. Each incoming build request is assigned to a child builder from the process pool, so as to handle subsequent build requests in fast manner.

The process pool is the way of spawning a limited set of processes at startup. All the newly spawned processes would have the functionality of the Core Build Server in project #2. This is done because the build server might get heavy loads during the releases. At that time in order to increase the throughput of the whole system, the build server provides queue of requests and each of the newly spawned processes would takes the request from it and executes it just like the main build server and provides the results and

logs to the repository and sends the requests along with the files to the test harness when the build succeeds.

The main tasks of process pool are-

1. Helps in running various packages on separate process so that main thread is not hampered
2. Helps in running several child builders at once
3. Helps in parsing multiple build requests at one time

7. Design Issues

The Remote Build Server has 4 distinct standalone servers. The implementation can be done in numerous ways. So, we analyze the issues that can cause at any point in time and propose a solution for each of them.

7.1 Data Access Redundancy

In a complex system, the file transferring between the repository may cause some issues as they have thousands of packages. The same package can be sent multiple times for different tests and If the test requires a lot of packages because of high dependency then it takes more time to send them through the WCF channel.

Proposed Solution:

If the system has a lot of packages with high dependency then they will be used in many tests which mean that the packages will be transferred to the repository, build server and Test harness a lot of times. Instead of transferring it repeatedly we can manage a cache folder for each of the servers so that whenever a package is needed the local cache is checked for the file and if the file is available then it saves a lot of time. In this case, we have to maintain the versions of the files. The versioning comes into place because we have to make a note of the changes done in the system and if we want to recreate a scenario in the system before the recent changes then this would be of great help.

7.2 Test Failures

The Test harness runs a lot of tests based on the test requests. If the developer or QA or anyone is running a bunch of tests overnight and if anyone of them fails, the system halts the process notifying the user and waiting for the response then the primary use of automating the test process is lost.

Proposed Solution:

The System should be designed in such a way that the failure of one of the tests shouldn't affect other processes as we know that we can have multiple test process spawned based on the requests. So, the automated run should finish all the tests as requested and the logs and results are to be shown to the user. If a test fails, then the failed log and result should be available for the user so that it can be taken care of. But in the end, the overnight process runs instead of stalling the process.

In some cases, the failed run would be expected as a part of the test case. in that case, we can put that in a try catch block so that the efficient result of the test can be obtained. We generally try to catch all the known issues using try catch statements.

7.3 Packages – Check In, Check out

We should maintain the privacy of the codes in any Software Baseline. There should be a mechanism to make sure that there are no concurrent writes on the same package by different processes. If a package is used by a user, then only he/she should have the privilege to make some changes in it. When they test and deploy the changed package then the privilege should be revoked.

Proposed Solution:

If there are two users trying to modify the same package, then the second user should be notified that the package has been checked out to the first user. After making the changes, thorough testing of the entire baseline is done after which the user needs to add changed the code and check it into the Baseline.

7.4 Logs Display

The logs and results are displayed to the users via Client Interface (WPF). It is the important part of the final project because all the functionalities in the different parts of the projects run in the backhand except the UI which shows the results of all these backhand processes to the user. If the display is not proper the user might not be able to use the Remote Build Server.

Proposed Solution:

We can have a separate UI screen where we can display the logs. The logs would extract the author name, date and time of the run, its duration thereby prioritizing the information so that the logs are more understandable. The user should be able to find out the test after looking at the logs.

7.5 Poor Quality Build Server Machines

The CI server machine is continuously running jobs that can involve heavy tasks such as downloading dependencies from remote servers, backing up databases, running Docker containers, etc. In order to perform those tasks effectively, the CI server must be fast, reliable, and connected to the network.

Potential Solutions

Get a quick and reliable server for Continuous Integration; avoid cheap servers. Never install unnecessary software on the CI server machine. Smartly share the CI server machine by assigning specific jobs.

8. Risks

The design and implementation of the Remote Build Server are relatively complex. We are going to discuss the various risk factors related to the system.

8.1 Data Loss

Transferring data between different parts of the system might lead to loss of data. The test request is parsed to find the files required. If the system misses any data while parsing it might affect the result.

Possible Solution:

We might have to follow the best way of data transfer through the communication channel which should read and send the data without removing anything from the source file. The parsing technique should be thoroughly tested with a good load of packages throughout the development phase. We can keep an indicator (flag) for making sure that a particular file transfer is properly completed or not.

8.2 Data Security

We have proposed some techniques above to avoid unauthorized use of data. After deploying the Remote Build Server, it is the customer's choice to transfer data on a secured line while outsourcing the data and tools. The Remote Build Server could secure the data transfer by supporting some authorization of users.

8.3 Mitigation

We can authorize the users to access the tool with a user id and password. The Data can be accessed based on the environment. The developer environment, QA environment, and production environment are some of the mainstream ways of providing data integrity.

9. Diagrams

9.1 Package Diagram

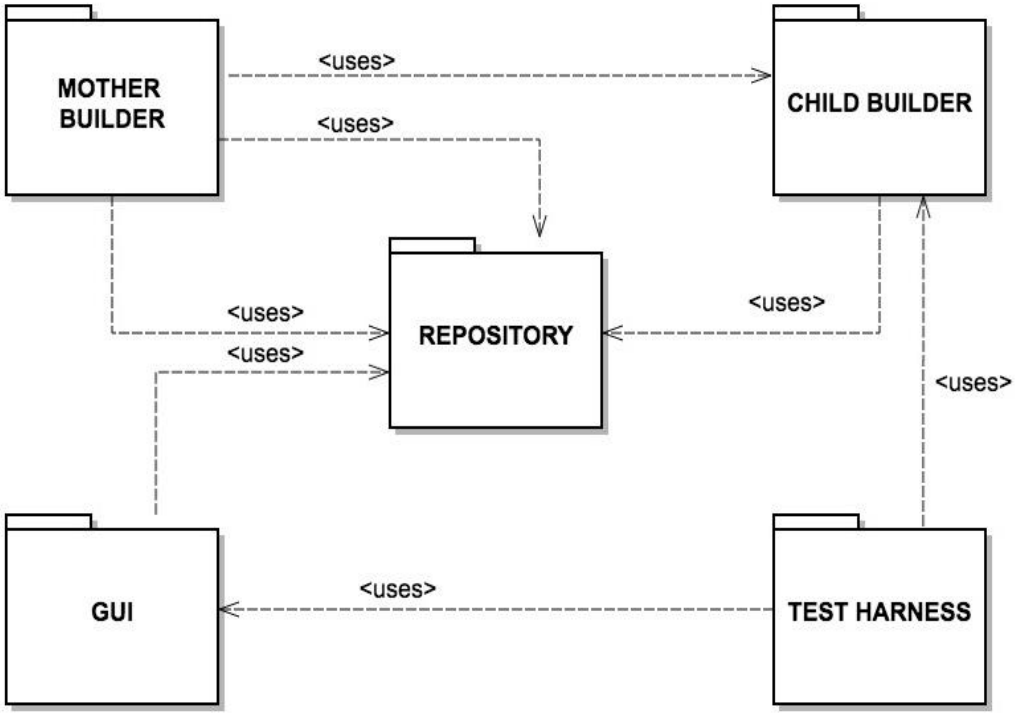


Figure 2 Package Diagram

1. **Client / GUI** - User Interface for Application
2. **Repository**- File Manager System of Application
3. **Test Harness-** The Testing Unit of Application
4. **Mother Builder**- The Build Server for the Application
5. **Child Builder**- Core Build Server of Application

9.2 Activity Diagram

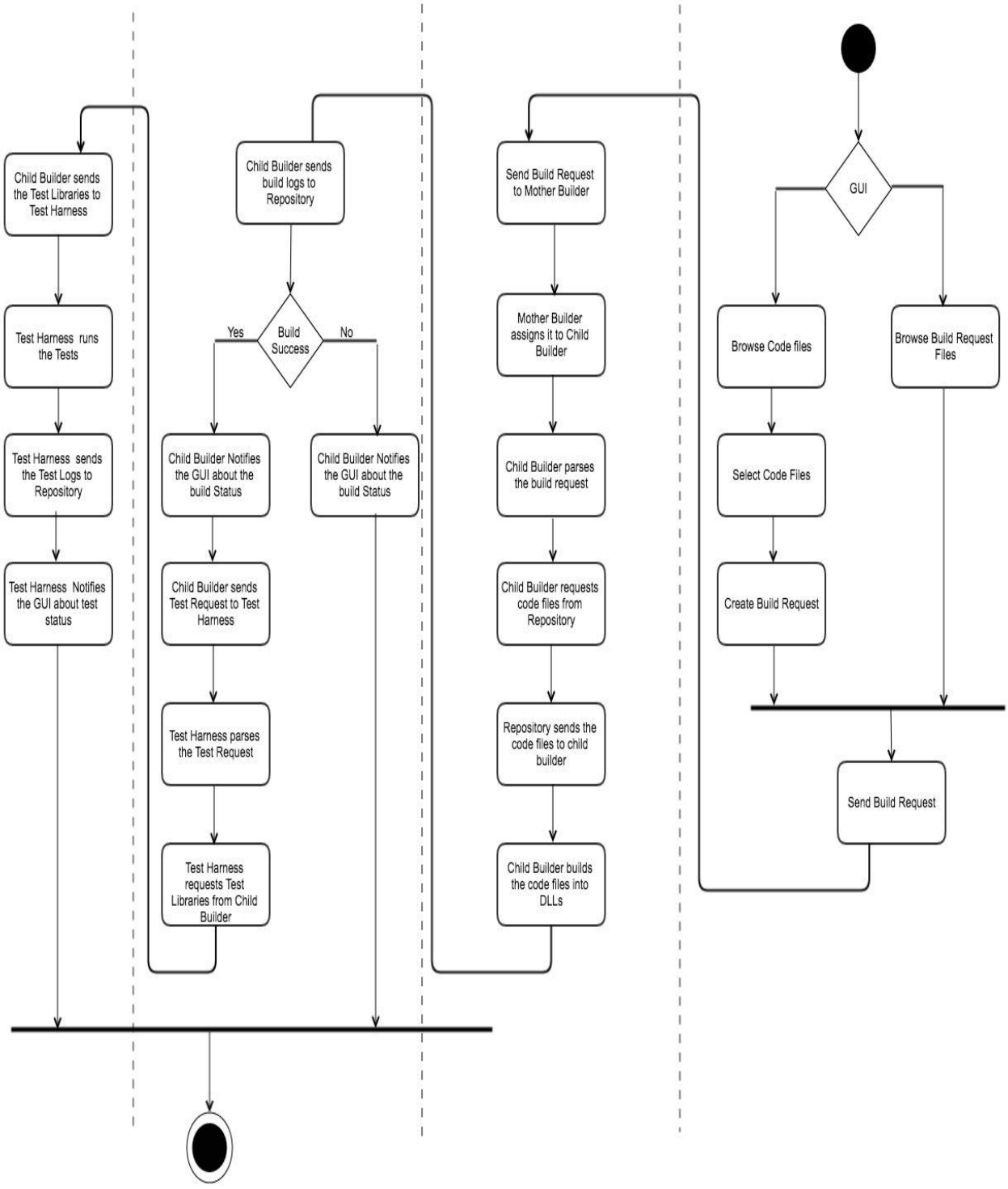
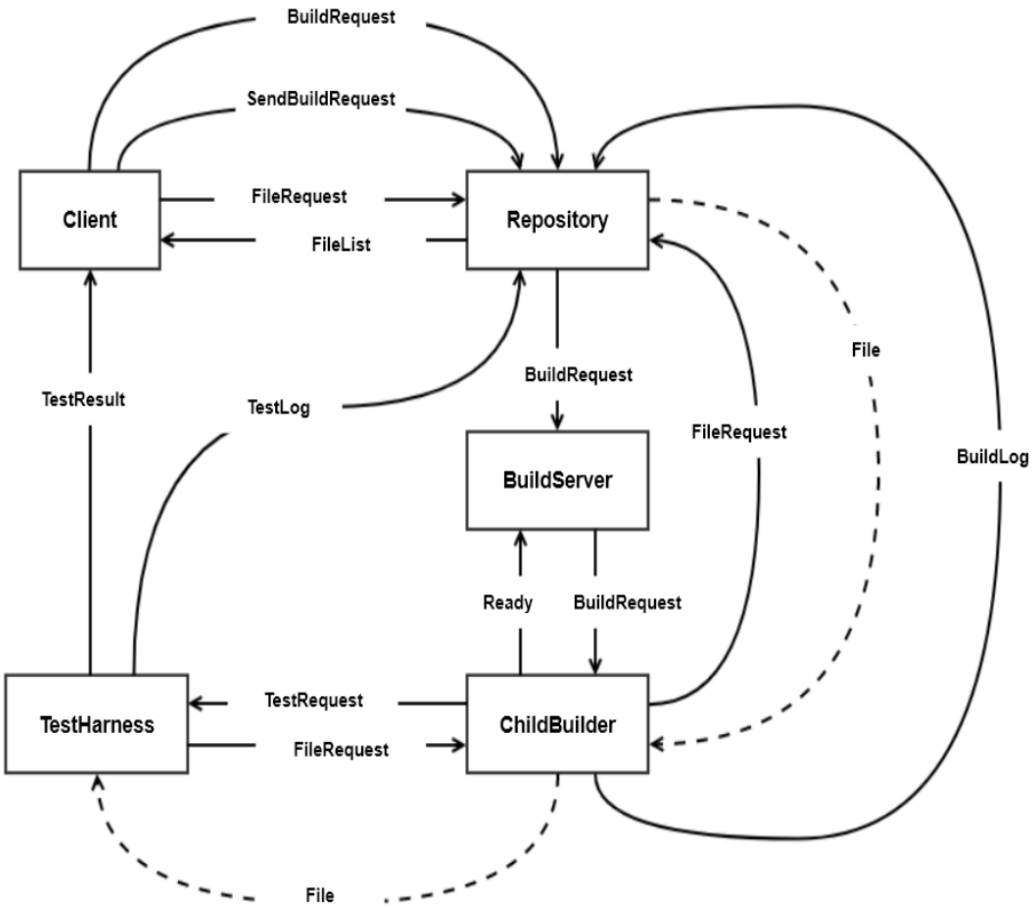


Figure 3 Activity Diagram

Activity Flow: The above activity diagrams shows all the functionality of this federation system.

1. User interacts with GUI to command the start of build process
    - it can either browse existing build requests
    - or it can create new build request by selecting code files
2. It sends build request to repository thus commanding it to start the build process
3. Repository on receiving the build request file and command to build gets triggered
4. Repository sends this build request along with build command to mother builder
5. Mother builder which already consists the ready states of child builders, puts the build request in the queue
6. Then it sends these build requests to child builders one by one by dequeuer
7. As a child builder receives build request, it parses it
8. Child builder then requests the files to build from repository
9. Repository sends the requested files to child builder
10. Then child builder builds the code files into **DLLs**
11. Child builder send the generated build logs to repository
12. Child builder notifies the client about the status of build
13. Further child builder sends the test request to test harness
14. Test harness parses the test request and sends request for test libraries to child builder
15. On receiving the test libraries test harness completes the testing process
16. Test harness then sends test logs to repository, also sends notification to client regarding test status

9.3 Message Communication Representation



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

Figure 4 Message Communication

9.4 Sequence Diagram for Message Passing

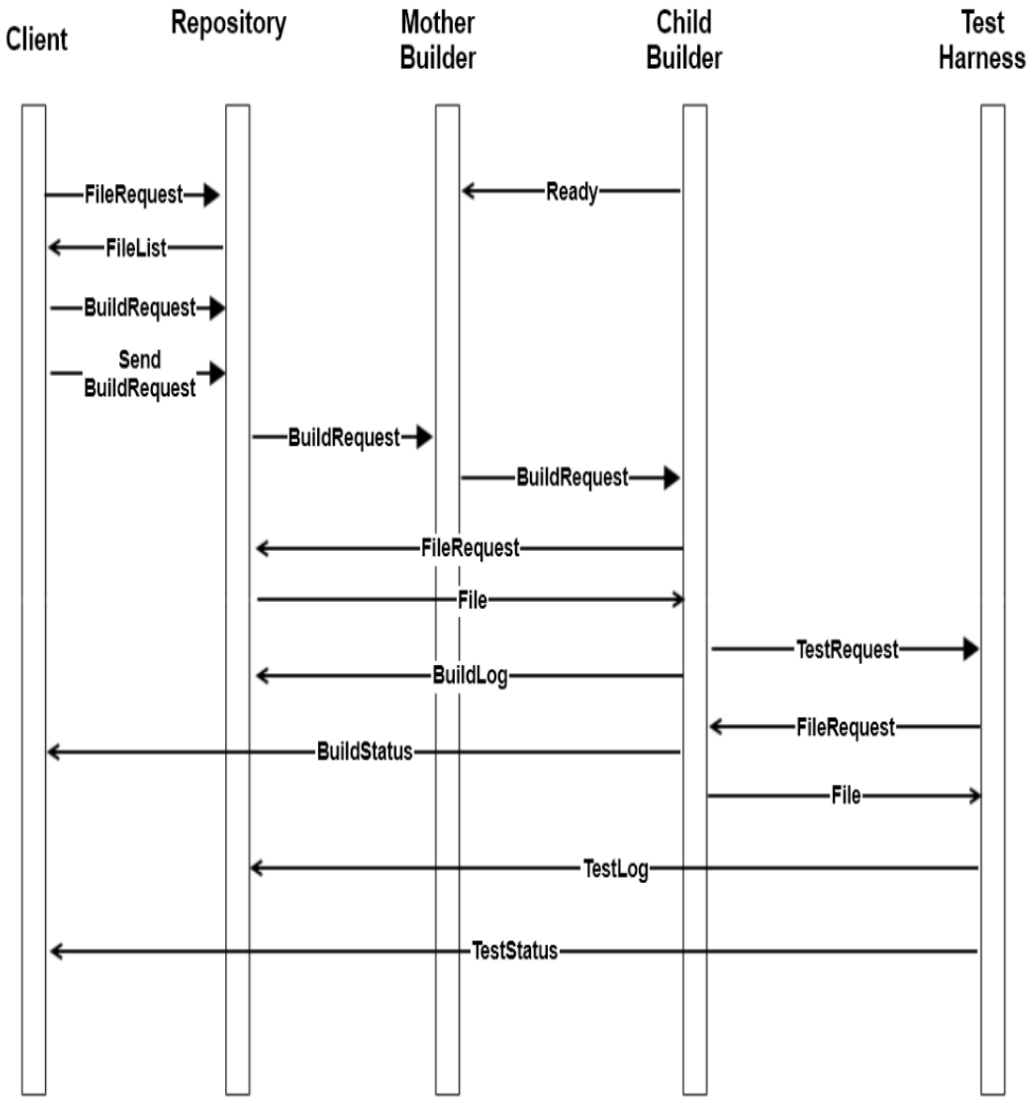


Figure 5 Sequence Diagram for Message Communication

Message interaction between components:

1. **Client-Repository:**
   - Buildrequest- The main build request file
   - Sendbuildrequest- Command to begin build
   - Filerequest- Command to show code files or existing build requests

2. **Repository-Build Server:**
   - Buildrequest- The main build request file

3. **Build Server- Child Builder**
   - Buildrequest- The main build request file
   - Ready -State message of child builder

4. **Child Builder- Repository**
   - Filerequest- Command to send code files to build
   - File- The main code files
   - Buildlogs- The build log files

5. **Child Builder- Test Harness:**
   - Testrequest- The main test request file
   - Filerequest- Command to send test libraries to test
   - File- The main test libraries

6. **Test Harness- Client:**
   - Testresults- Status of the test

7. **Test Harness- Repository:**
   - Testlogs- The test log files

9.5 Class Diagram

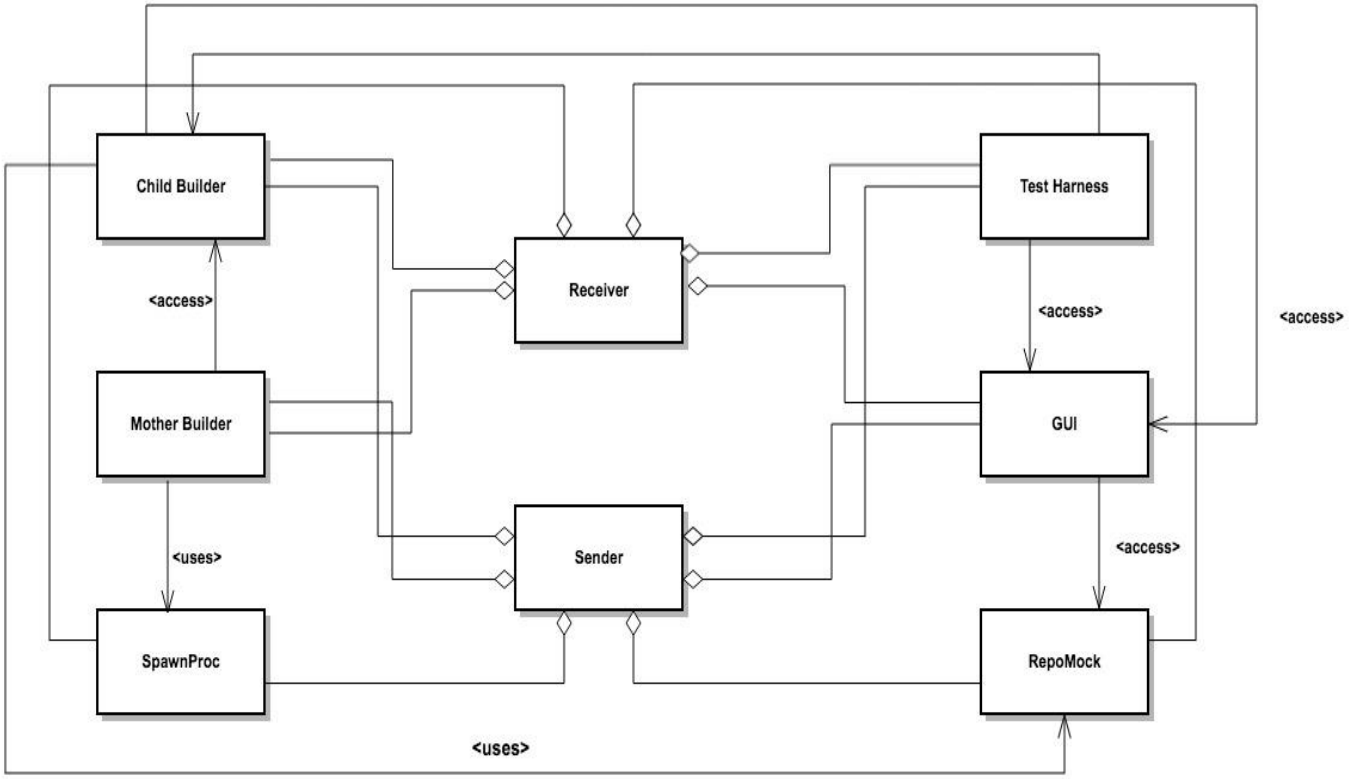


Figure 6 Class Diagram Remote Build Server

The Main Classes of our Application are:

1 **Mother Builder**- to implement the mother builder functionality
2 **Child Builder-** the core build server of the application
3 **SpanProc**- to implement the process pooling
4 **Sender**- to implement sender message
5 **Receiver**- to implement receiver message
6 **Test Harness**- to implement testing functionality
7 **GUI-** to implement user interface
8 **RepoMock**- to implement repository features

10. Possible Inadequacies and Errors of Commission

The plausible places where system might jam are as follows-

- **GUI** -It might happen GUI fails to load, in this case the user is not able to send build command and process can't be started.
- **Message Transfer** -the messages might not get delivered, in that case the communication between modules might fail and so the intended processing might not happen. Care should be taken that the port numbers are correctly assigned.
- **File Transfer** - Files might not get transferred to right destination, due to wrong paths or inaccessible paths. Proper care should be taken for the sending and receiving of messages which involves file exchange. For this receiver should send proper acknowledgement to sender stating that its ready to receive the files.
- **WCF** -Exceptions such as abrupt channel close or connection timeout might happen. In such cases message passing is halted and thus application also stops. Sometimes connection doesn't get established and this might lead to failures in communication.
- **Process Pool** -While running a managed application, the pool of threads will be created the first time the code accesses it. the pool is associated to physical process, same where the application is running, it is a functionality available in the .NET infrastructure to run several applications (called application domains) within the same process. In these cases, one unreliable application can hamper the functioning of the rest within the same process because they all use the same pool.
- **Test Harness**- The test harness needs the proper DLLs to carry its testing process, so if the DLLs are missing or are misplaced it might happen that testing generates wrong results.
- **Process Kill**- The application might throw warning of "Unreachable Code Detected" if the processes are not properly closed. Hence, all the processes should be closed with utter caution.

11. Changes to the Core Concept as Design Evolved

The design has evolved through various stages. As we developed the remote build server we successfully implemented a client-server structured application. Our core concept was to support continuous integration for which we developed a core build server. This server provided the functionalities to build code files into **DLLs**.

For this we used:

1 **Local Mock Repository**- It supports the file storage and retrieval, being the prime file manager, it holds all code files, build requests and test request files along with the log files. It's the prime database for the application. It sends the build request files to core build server. It serves core build server by sending it the code files to build. It also stores the build logs sent to its core build server.

2 **Local Mock Test Harness**- this handles the complete testing procedure, it is responsible for testing the **DLLs** created by core build server. It parses the test request sent to it by core build server. It obtains the test libraries and successfully generates the test results. It also interacts with mock repository by sending it test logs.

3 **Core Build Server-** the main build server of our initial application. It parses the build request sent to it by repository. Further it obtains the code files to build from the repository. It also generates the build logs and sends it to repository.

4 **Test Executive-** in our initial application, test executive is our client. It didn't have a GUI to support user friendliness. This test executive was responsible to send command to repository to start the build process.

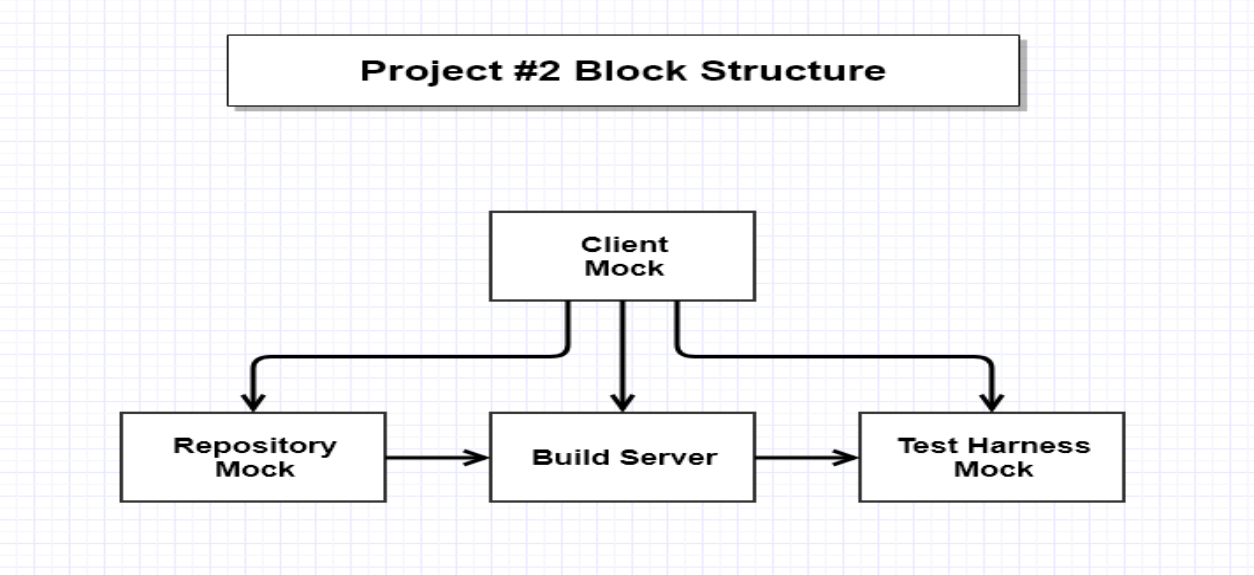


Figure 7 Block Structure of Core Build Server

Thus, as we discussed our initial prototype involved a stand-alone application which supported local packages for its procedure. Initially we implemented just core build server which successfully built and tested code files sequentially.

As we moved on further stages we made significant changes in the application. Now it supports multiple build servers running at the same time along with a well-formed GUI. We also implemented the message

passing communication in our application so that a flawless and reliable interaction can be handled for all he remote components of remote build server

For this we changed:

1 Repository- We made repository to run on different process. It now has a well-versed communication channel implemented using WCF. All the interactions like file requests and file transfers now happen via message passing only.

2 Child Builder- Our core build server now turned into child builder. There are multiple child builder now running on different processes. All the functionalities of core build server as inculcated in each child builder thus supporting simultaneous builds and faster builds. It also has a well-versed communication channel with which it interacts with mother builder, repository, GUI and test harness. It sends notification to GUI now.

3 Mother Builder- We added a new component called mother builder. It runs on a different process and maintain its own process pool of child builders. It has blocking queues which maintain ready state of all child builders and a blocking queue to hold build requests sent to it by repository. This component is responsible to batch each build request to respective child builder. It also has a well-versed communication channel with which it interacts with child builders and repository

4 Test Harness – Our test harness is now running on different process to support multiple tests. It has all the functionalities as it previously had alongside It also has a well-versed communication channel with which it interacts with child builder, repository and GUI. It sends notification to GUI now.

5 GUI- we have now developed a user friendly and an interactive GUI for our application, it helps user to browse through build requests, create new build request by selecting code files. It also shows build and test logs for user satisfaction. Further the status of build and test procedure is displayed so that user is well informed about whole procedure.

6 WCF- We have modified whole interaction foundation of our application by implementing the message passing communication method for all the components. This helps in increasing the reliability, robustness and increases the speed of whole process of building and testing.

7 Process Pool- It is a new addition to our core concept. Since we are trying to achieve CI process, so we need to run all the components on different processes. It helps in implementing multiple builds concurrently.

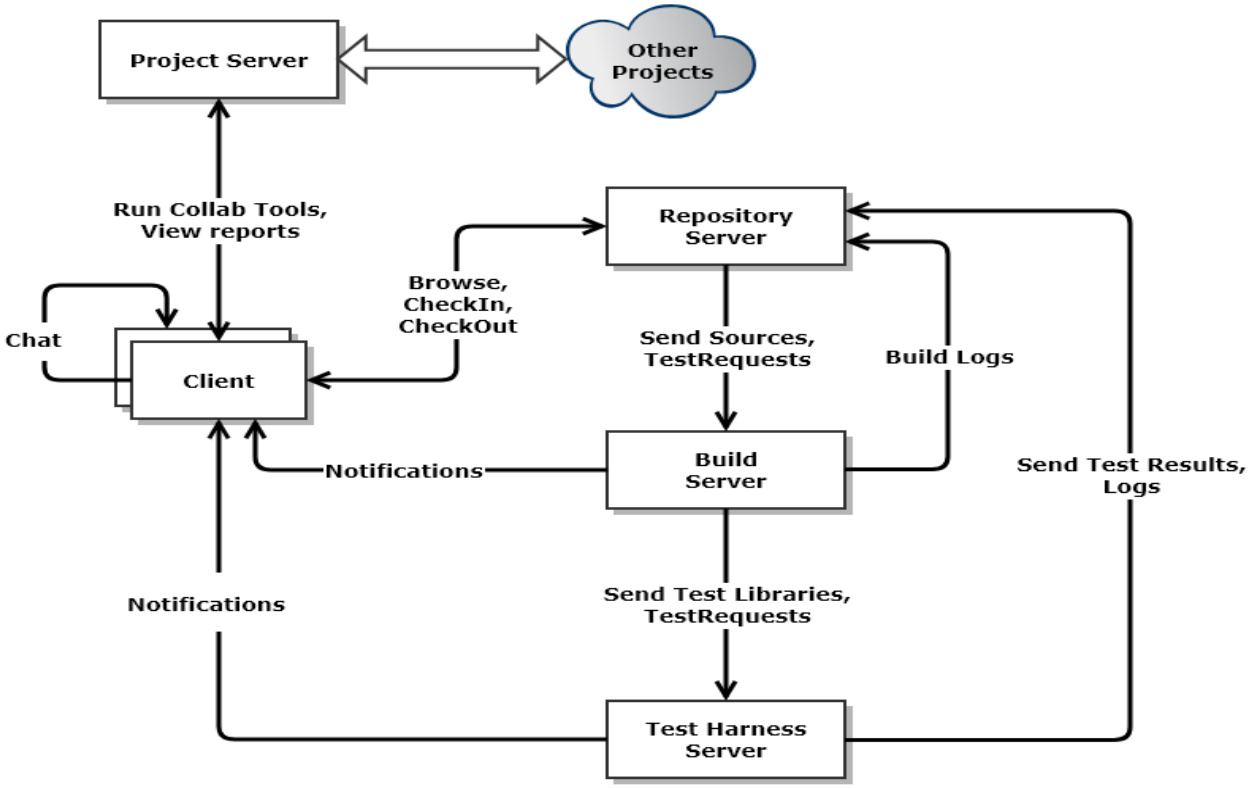


Figure 8 Collaboration System

12. Deficiencies the Project Incorporates

The project incorporates few deficiencies which are:

1 PORTS- The ports used for the communication of different components of our Remote Build Server is currently hardcoded, instead an XML File should be ideally given so that on startup all the components load using the information stored in this XML.

2 VERSIONING- The Repository is our efficient file manager system currently. But the system lack versioning, due to there's no history for a file that is maintained. In current system files are overwritten rather than maintaining different versions of same file.

3 CROSS PLATFORMS- In the current system we have a support to build and test only c sharp i.e. .cs files. It lacks the support for c++ or java files. Hence its limited to certain language only.

4 GUI- In current application, the GUI has features to create and send build request, it lacks the platform selection functionality. The GUI shows status of build and test after completion of process, it doesn't constantly notify the user.

13. Conclusion

The Core Build Server which was developed in Project #2 was successfully imparted into the Communication Channel as a remote child builder along with the other child builders. Considerable efforts were put in the further projects where the prototypes are built for the other parts of the system (project #3) and include the services over the prototypes and build the Remote Build Server. The Core Build Server is the primary server that has the maximum functionality and we divided the development of the project into 4 steps - the final one giving out the product. Since the Build Server is developed separately in project #2 we can estimate the time properly and it won't be an issue. Similarly, the Final project uses the prototypes developed in Project #3 and Core build server. Since we don't deal with unnecessary packages or any form of data because of the proper planning, we can reduce the production cost.

Thus, our final product is a remote build server which works on a well-designed foundation of WCF communication channel and Process Pooling services. Our final product is fully capable to support continuous integration by simultaneously building and testing the code files. We have successfully achieved the goal of supporting large systems in the continuous integration process by developing a system of build servers and test harness.

14. Output Screenshots

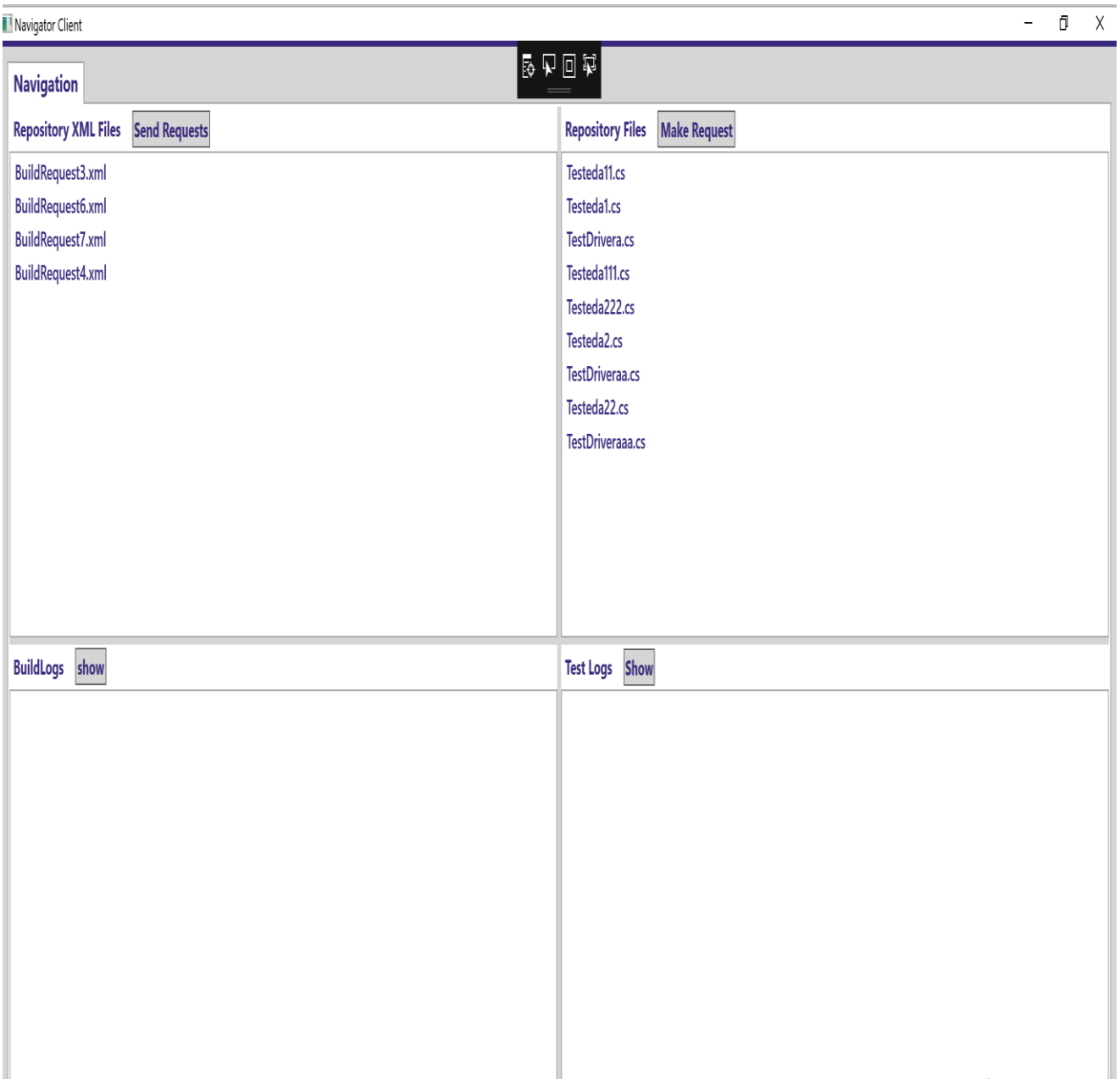


Figure 9 The GUI of the Project

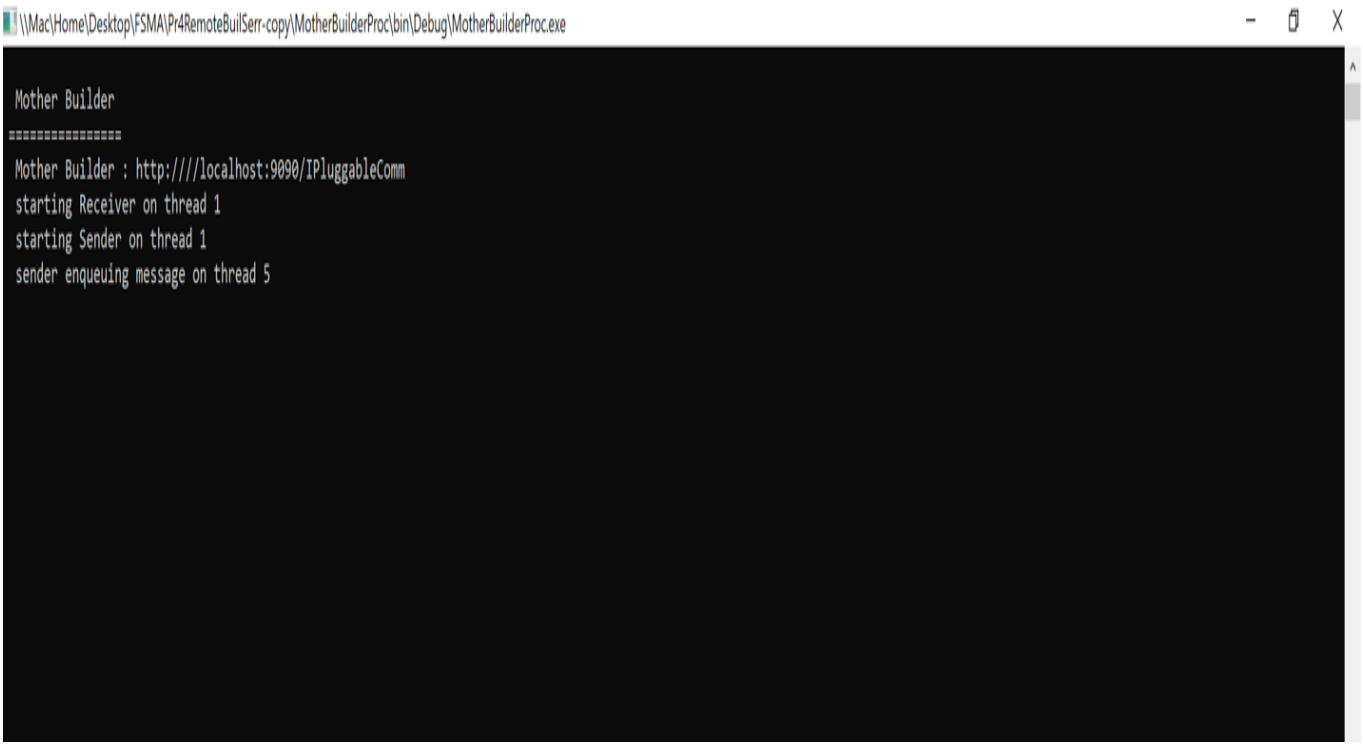


Figure 10 Screenshot of Mother Builder Process

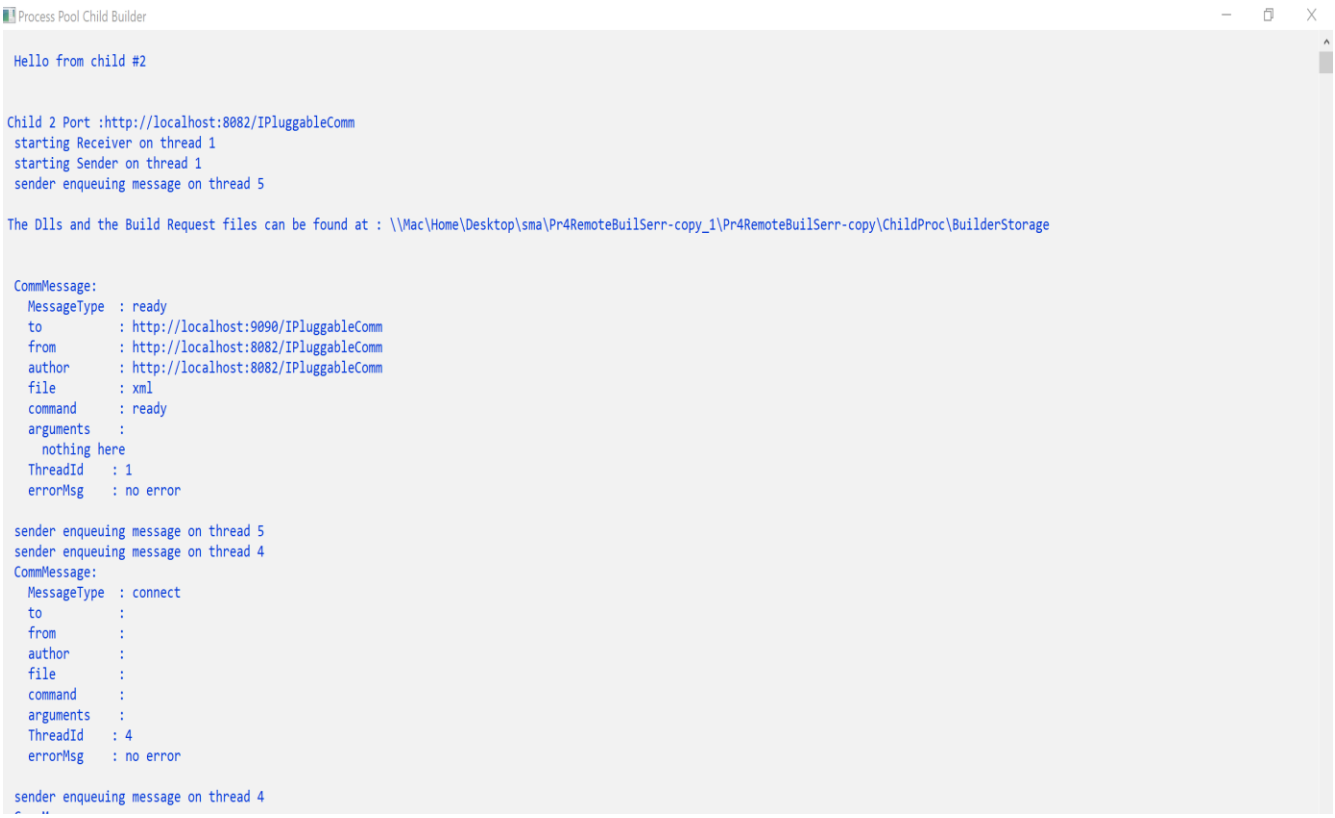


Figure 11 Screenshot of Child Builder Process

15. References

- **https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/threading/thread-pooling**
- **http://www.c-sharpcorner.com/UploadFile/1d42da/threading-pooling-in-C-Sharp/**
- **https://martinfowler.com/articles/continuousIntegration.html**
- **https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/BestProject1s/RonakBhuptani.pdf**
- **https://en.wikipedia.org/wiki/Versioning_file_system**
- **https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/midterm/MTF17/MT3F17-InstrSol.pdf**
- **https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/midterm/MTF17/MT2F17-InstrSol.pdf**