Operational Concept Document
For
Build Server

By
Butchi Venkata Akhil Rao
SUID: 463365540

Submitted to
Professor Jim Fawcett
Date: 12-06-2017

Contents

Figures

1. Executive Summary

This document contains the Operational Concept Document of Build Server. Build Server is a tool that enables Continuous Integration. Continuous Integration is a technique, in which developers integrate code into a shared repository. This integration is done several times when doing a project. By integrating frequently, we can detect where things went wrong and can find the errors easily. In this project, build server is capable of building only c# libraries.

Build Server is developed for performing the following activities:

- Build Server builds the test libraries, which are based on build requests and code received from the repository.
- After successful building of test libraries, the libraries are sent to the Test Harness along with the test requests.
- Build logs are sent to the repository.

By using Build Server, we will have the following uses:

i. **Continuous Integration:** Probability of committing mistakes is less, because of the fact that build server allows us to integrate code regularly into a repository and check for errors.
ii. **Machine Independence:** The work of a developer is not affected due to the change in Workstation because Build Servers integrate builds made from any machine.
iii. **Scripted Builds:** Builds are automated. The automated builds include compiling source code into binary code, dividing binary code into various packages and running automated tests.
iv. **Scripted Tests:** Build Server also catches test errors of a product by using test scripts.
v. **Deployment:** Deployment of the code is made easy by the use of Build Server.
vi. **Multiple Environments:** Build Server allows the builds to be created in various programming languages on different Operating Systems.
vii. Code can be altered i.e. some code can either be added or deleted or the existing code can be changed without actually affecting the previously developed code.

Main components of a Build Server are:

i. **Build Core:** Build Core is responsible for building packages from source code.
ii. **Version Control System(VCS):** VCS stores source code of all packages.
iii. **Distribution Build System:** This creates the final product.
iv. **Web User Interface:** Web UI is responsible for interaction between developers and the build environment.

Coming to the architecture, the Build Server needs to perform the following activities:

- Test requests must be accepted
- Parse the request
- Creation of temporary directory
- Accepting the files
- Build
- Creation of build log and sending that to the repository
- If build is successful, then it has to send the libraries to Test Harness
- In case build fails, information must be given to the client.

Work for this entire project can be divided into 3 main parts:

a. In the first part, a build server which communicates with mock Repository, mock Client, mock TestHarness is developed.
b. Prototype for Message Passing Communication is developed in the second part along with a process pool. Process pool communicates between the parent builder and child builder.
c. Third part is about implementing all the above-mentioned functionalities by using the work done in first and second parts. Here, a full-fledged build server is developed.

Below mentioned are some of the critical issues, which we need to consider while developing Build Server:

- Security of the Build Server
- Request Handler (Handling multiple requests)
- Building the code in different programming languages
- Complexity of the Build Server

2. Introduction

Build Server is an automated tool that enables Continuous Integration and builds test libraries. Below mentioned are some of the Obligations, Organizing Principles and key architectural ideas.

2.1 Obligations

The main duties of the Build Server are:

- Build Server has to receive test requests and sources from repository.
- Then, the Build Server needs to build test libraries and submit them to the Test Harness.
- Test requests must be sent to the Test Harness along with test libraries and build logs must be sent to the repository.
- Build Server also should maintain a log for all errors (if any occur) and warnings.

Other important obligations include:

- Checking out the repository
- Triggering the build
- Triggering the tests

2.2 Organizing Principles

- For developing Build Server, we are going to use .Net Framework and Visual Studio 2017.
- The main principle we use in developing Build Server is to dedicate a package for every important functionality of the Build Server.
- Package can be defined as the thing which organizes a set of related classes and interfaces.
- Other important point is that we have to make sure the Build Server we develop must solve all the critical issues mentioned.

2.3 Architectural Ideas

- The key architectural idea is to design Build Server using Model View Controller architecture.

Other architectural ideas are as follows:

- Build Server must parse the xml test request files.
- It must generate log file where details regarding errors and warnings are stored.
- Build Server also should create a temporary file directory, where the required source code can be saved.
- It also should be able to command the Test Harness to test code on receiving a test request.
- Dynamic Link Libraries (.dll files) must be created for each test request.
- Other main idea is that the Build Server must be able to handle multiple test requests.

The main packages to be included for performing the above-mentioned functionalities are:

i. Test Harness
ii. Blocking Queue
iii. Mother Builder
iv. Repository

Above mentioned packages are the most important ones but there are some other packages as well. We will be discussing all those packages in the "partition" section below with the help of a package diagram.

3. Users, Uses and Possible extra features

In this section, we are going to discuss about the possible users of the Build Server, processes they can do while using Build Server and the possible features, which can be added later to the application.

3.1 Users

Users are the ones who are directly or indirectly involved with the development or usage of the Build Server (in this case).

3.1.1 Developers

- Developers are the main users of Build Server.
- Build Server helps developers to share their code with others by sharing in the repository.
- Extra features to the Build Server can be added by developers.
- Problems with the Build Server are also identified by developers.

3.1.2 Managers

- Manager is responsible for work done by the group of developers (who are responsible to the manager).
- They are also responsible for delivering good quality product within a specified time frame to the client.
- With the help of Build Server, the manager can keep track of the progress of project.
- Managers use build logs to track the work done by each developer, hence keeping track of the whole project as well.

3.1.3 Quality Assurance

- Quality Assurance Team makes sure that there are no defects in the final outcome of the product.
- They check each module and find defects (if any present) and rectify them.
- Quality Assurance Team also needs to collaborate with developer team and the manager to enhance the functioning of the application.
- They also need to collaborate for providing any temporary solutions, if the application doesn’t work all of a sudden.

3.1.4 Instructor and Teaching Assistants

- Instructor and Teaching Assistants will test the Build Server and check whether all mentioned functionalities and requirements are implemented while developing the Build Server.
- They also verify whether the Build Sever is feasible and check whether any extra features can be added.
- They inform the developers whether any changes are required to be made.

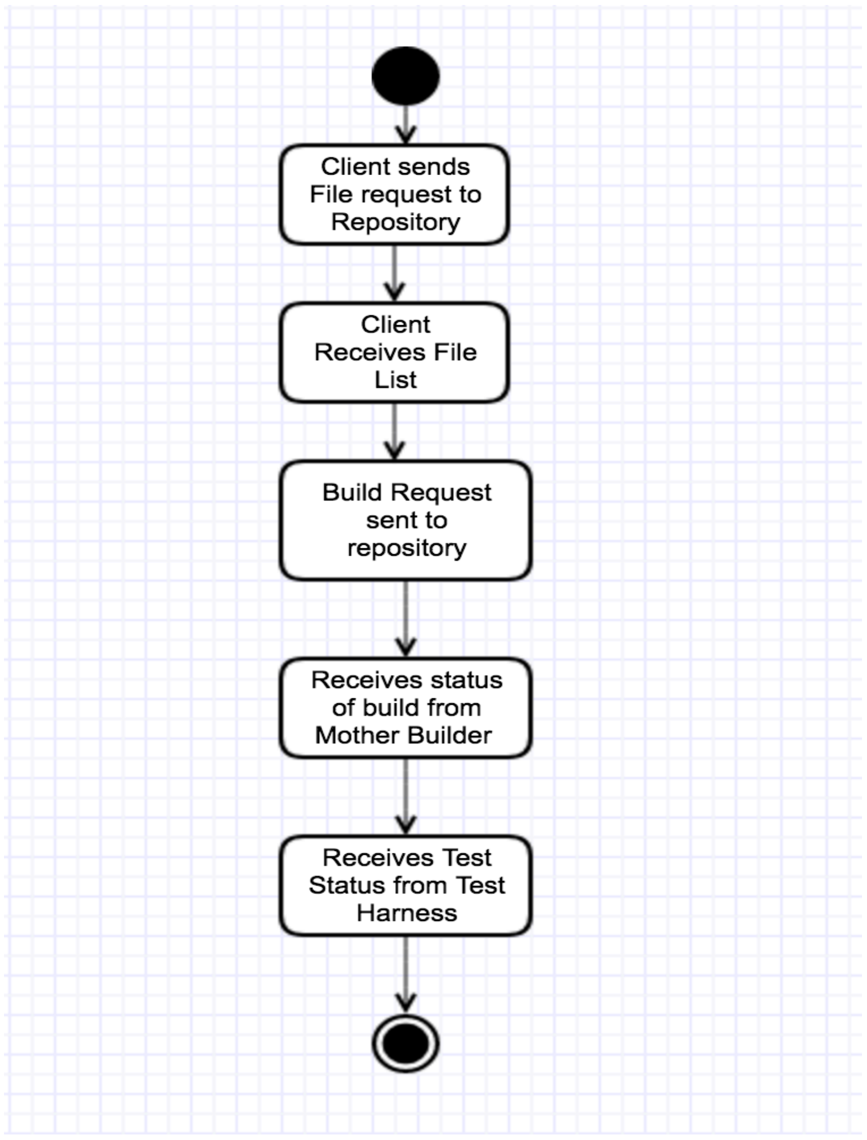
3.2 Possible Extra Features

- Handling multiple request processes at a single time.
- Multiple users can be able to deploy their respective source codes in parallel.
- Interactive Graphical User Interface.

4. Application Activities

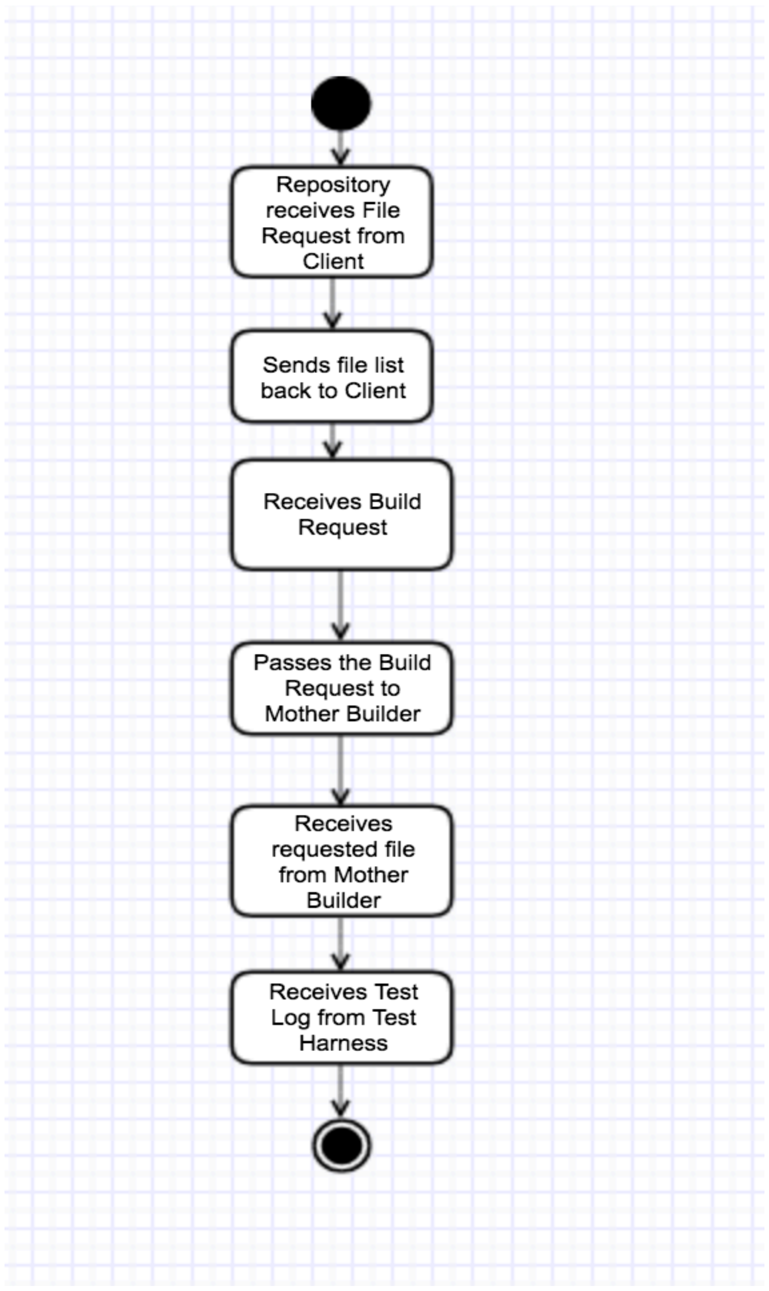
In this section, we are going to see the work flow of various components in Build Server with the help of activity diagrams.

4.1 Activity Diagram for Client:


Client sends File request to Repository
Client Receives File List
Build Request sent to repository
Receives status of build from Mother Builder
Receives Test Status from Test Harness

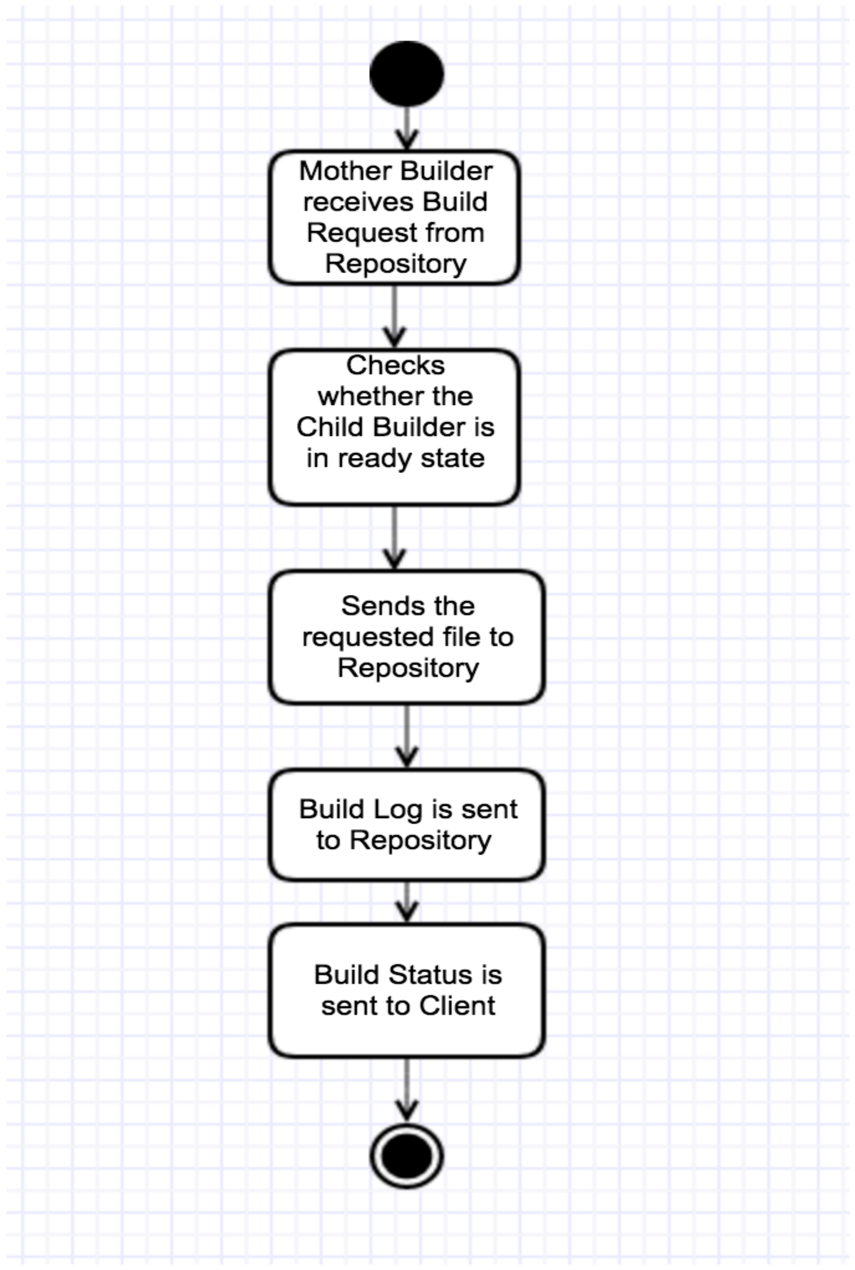
- Client sends file request to repository.
- Client receives the requested files.
- The build request for file is sent to Repository.
- Build status received from Mother Builder.
- Test status is received from Test Harness.

4.2 Activity Diagram for Repository:


Repository receives File Request from Client
Sends file list back to Client
Receives Build Request
Passes the Build Request to Mother Builder
Receives requested file from Mother Builder
Receives Test Log from Test Harness

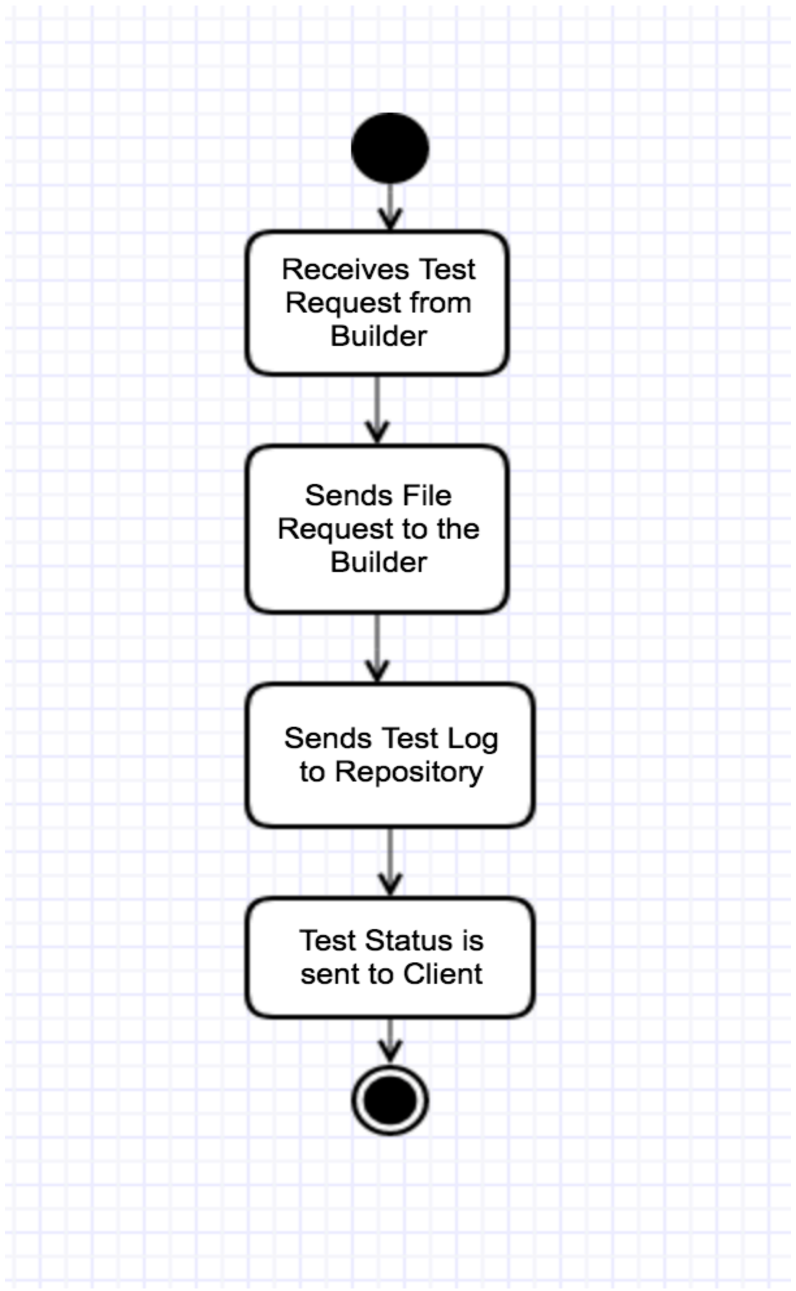
- Repository receives File Request from Client.
- File List is sent back to the Client.
- Build Request is received.
- Build Request is passed to the MotherBuilder.
- Requested files are received from MotherBuidler.
- Test Logs are received from Test Harness.

4.3 Activity Diagram for MotherBuilder:


Mother Builder receives Build Request from Repository
Checks whether the Child Builder is in ready state
Sends the requested file to Repository
Build Log is sent to Repository
Build Status is sent to Client

- MotherBuilder receives Build Request from Repository.
- Allots each request to a ChildBuilder.
- Requested file is sent to the Repository.
- Build Status is sent to the Client.

4.4 Activity Diagram for TestHarness:

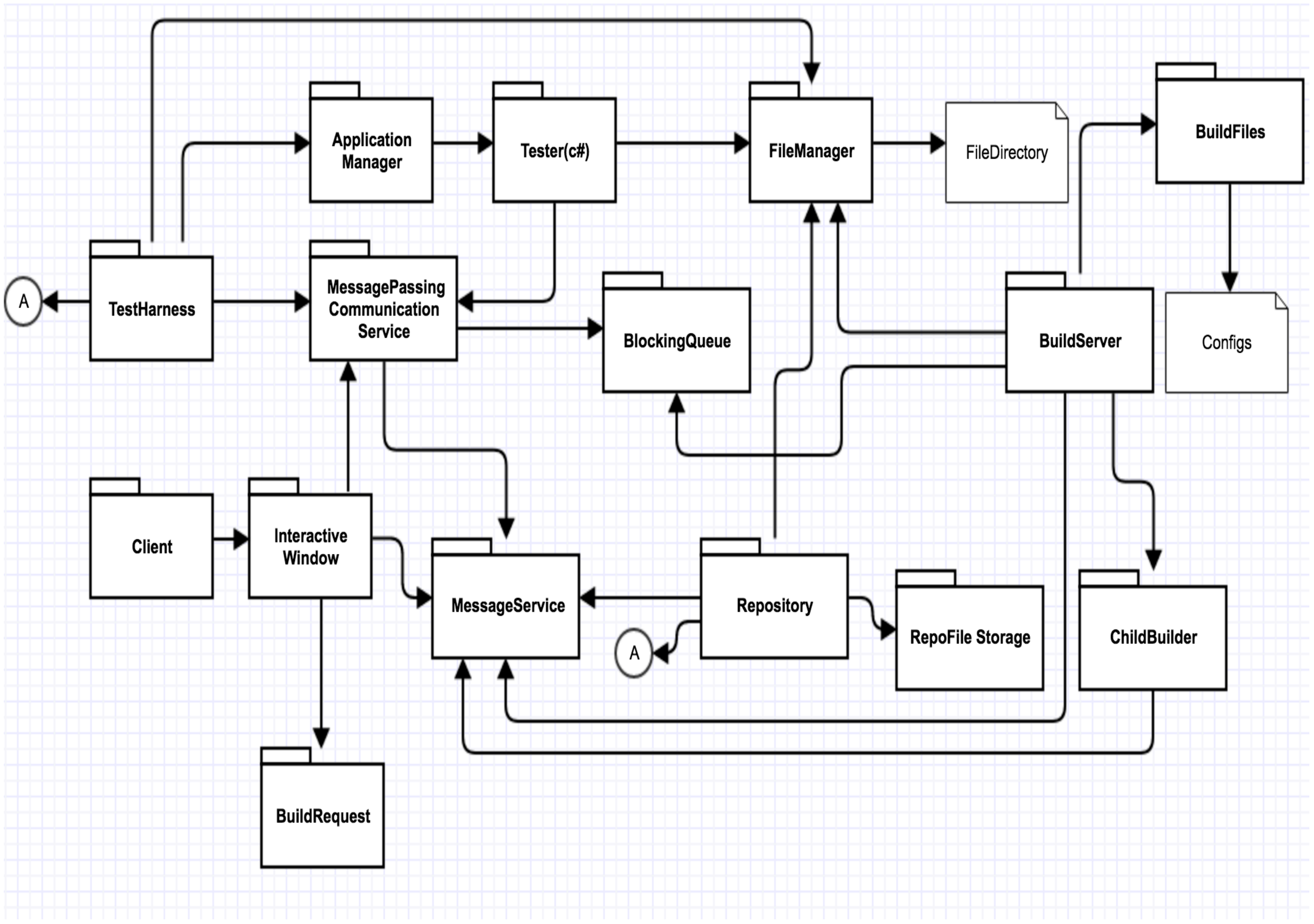

Receives Test Request from Builder
Sends File Request to the Builder
Sends Test Log to Repository
Test Status is sent to Client

- Test Request is received from Mother Builder.
- File Request is sent to the Builder.
- Test logs are sent to the Repository.
- Test Status is sent to the Client.

5. Partitions

For Build Server to work the way we want, we need to split the work to be done by it into packages. Each package will perform a unique function, which helps us in achieving our target for the total functionality of Build Server.

5.1 Package Diagram



5.2 Packages Description

i. **TestHarness:**

Test Harness receives files, performs testing and sends the files after testing to the File Directory.

Uses of Test Harness include:

- Automating the test process
- Executing the test cases
- Debugging

ii. **ApplicationManager:**

Application manager contains the files which are to be tested and it passes the files on to tester. It receives files from TestHarness.

iii. **Tester:**

Tester tests the files and sends them to the File manager.

iv. **FileManager:**

It manages all the functionalities for a given file such as adding a file, deleting it, copying, etc.

v. **MessagePassingCommunicationService:**

This is the main communication package and acts like service contract for TestHarness. Communication Service is responsible for the interaction of client with various functionalities of our Build Server application.

vi. **Client:**

Client is nothing but the end user package.

vii. **Interactive Window:**

In this window, various options are mentioned which when selected perform activities of this application.

viii. **BuildRequest:**

This package builds the requests based on the selection from client.

ix. **MessageService:**

Message Service provides the medium for client, repository, builder to interact with each other.

x. **BlockingQueue:**
Blocking Queue implements a generic thread safe queue. Used for communication between Build Server and other important components.

xi. **Repository:**
The source code metadata information is stored here. Repository sends the information to File Manager, which saves it in the file directory.

xii. **RepoFileStorage:**
This package stores the files which are received from the repository.

xiii. **BuildServer:**
Build Server package creates Dynamic Link Libraries (.dll) for the files and sends them to Build Files package for parsing. Build Server checks whether the files that are needed are present in Cache memory or not. If the files are present in the Cache memory, they are retrieved from the Cache Manager. If the files are not present in the Cache memory, they are retrieved by the Build Server from File Manager.

xiv. **ChildBuilder:**
Child Builder builds the child processes from the processes received from Build Server.

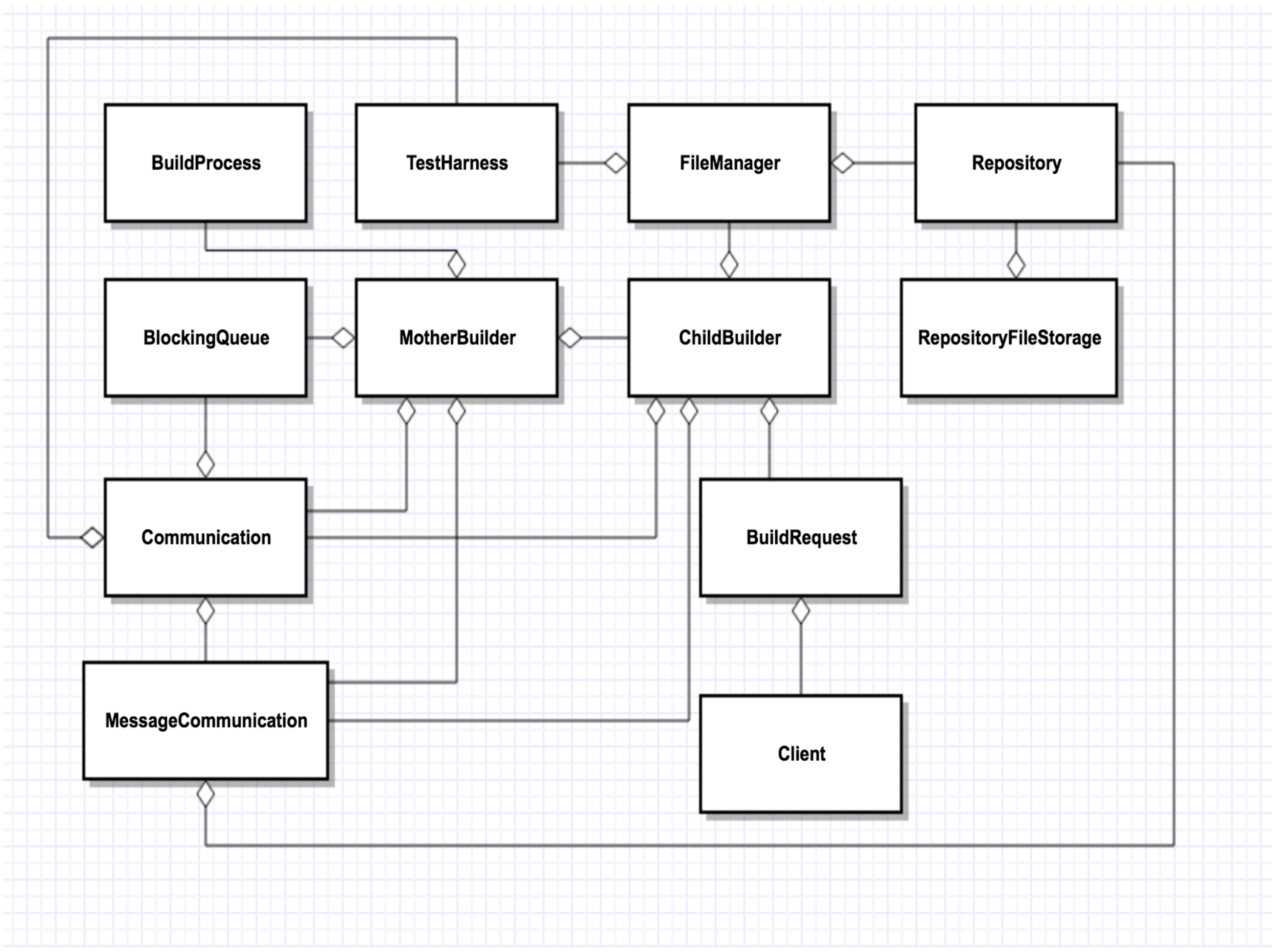
xv. **BuildFiles:**
This package stores all the files received from build server.

xvi. **File Directory:**
This is not a package but a storage path, where we can retrieve the stored files. File Directory sends the files to File Manager when asked by it.

xvii. **Configs:**
This file keeps track of the packages which are needed to be installed for setting up the required environment.

6. Class Diagram:

In this section, various important classes are identified and presented in form of a class diagram.



6.1 Classes Description:

i) Client:

Client asks for build requests.

ii) TestHarness:

Test Harness receives files, performs testing and sends the files after testing to the File Manager.

iii) Repository:

Stores the source code information.

iv) **RepositoryFileStorage:**
Receives and stores the files received from Repository.

v) **MotherBuilder:**
MotherBuilder passes the build requests to ChildBuilder.

vi) **ChildBuilder:**
Loads files and checks them with build request from repository and coverts them into libraries before sending them to TestHarness for testing.

vii) **FileManager:**
Creates temporary directory for the files it receives from TestHarness and repository.

viii) **Communication, MessageCommunication:**
They enable the communication among Repository, TestHarness, MotherBuilder and ChildBuilders.

ix) **BuildRequest:**
Parses the build request message.

x) **BlockingQueue:**
BlockingQueue is used for communication between Build Server and other important components and it implements a queue which is generic and thread safe.

7. Critical Issues and Possible Solutions

i. If test requests are sent to the Build Server in large numbers at a time, Will the Server execute all the requests without being slowed down?

Solution:

We can make Build Server execute all requests at a time without being slowed down by using the concept of processing multiple test requests at a time. But, this requires high level architecture.

ii. If the environment used by developers is different from the environment used in Build Server, you will be having trouble. How will you manage this?

Solution:

For solving this problem, Build Server maintains a configuration file, which installs all the packages required for running the file on Build Server.

iii. How will the Build Server execute when the C# code you wrote is inter dependable on few CPP libraries?

Solution:

Build Server uses a tool called toolchain which converts the CPP modules into C# format. So, there is no problem even if the C# code is dependent on CPP libraries.

iv. In some cases, source file changes in the repository and now, it doesn't match any file in the cache memory or file manager. How will you solve this issue?

Solution:

The Build Server regularly checks performs integration of code. So, if the source file is changed, then the files related to it in the cache memory or file manager are also changed.

v. How can you define a message structure for all messages which are used in the Federation?

Solution:

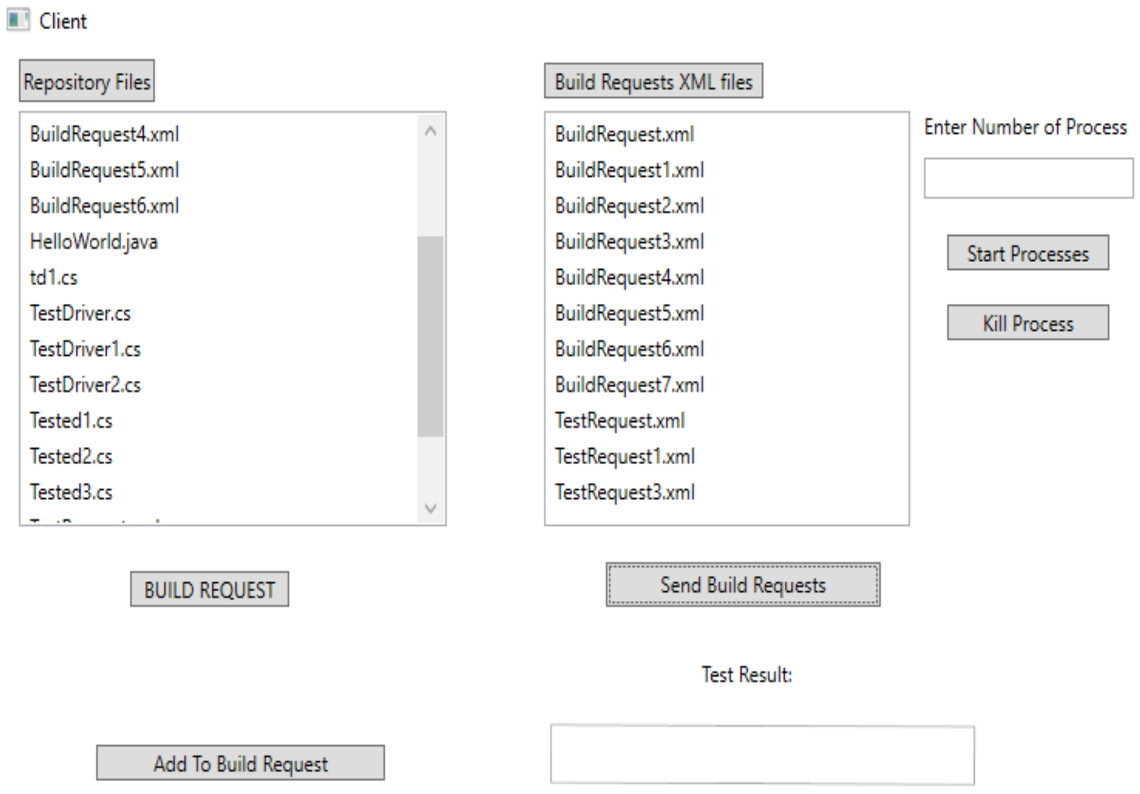
The basic message structure used in build server contains details related to addresses ("To" address and "From" address). This structure uses list of strings to contain file names and logs are stored in a string body.

8. Appendix

In this section, a few outputs for the project are mentioned.

8.1 Outputs:

a) GUI Output Screen:

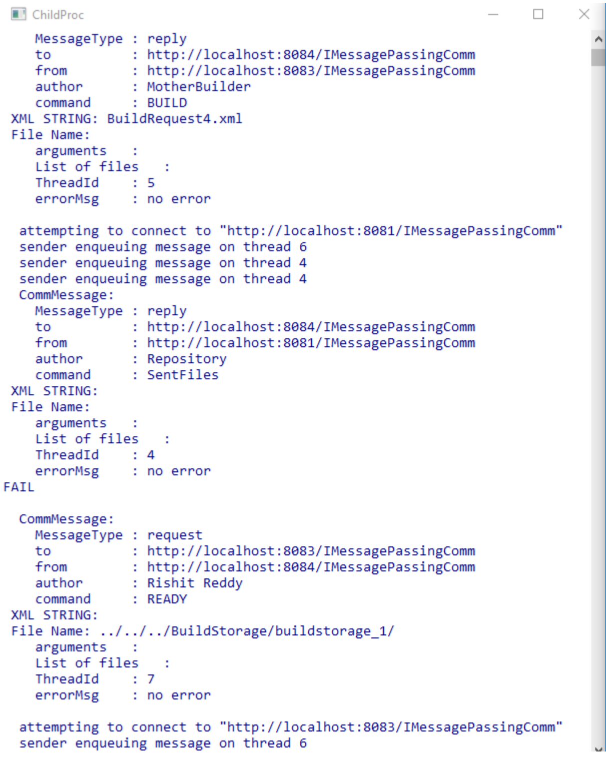


BuildRequest4.xml
BuildRequest5.xml
BuildRequest6.xml
HelloWorld.java
td1.cs
TestDriver.cs
TestDriver1.cs
TestDriver2.cs
Tested1.cs
Tested2.cs
Tested3.cs

BuildRequest.xml
BuildRequest1.xml
BuildRequest2.xml
BuildRequest3.xml
BuildRequest4.xml
BuildRequest5.xml
BuildRequest6.xml
BuildRequest7.xml
TestRequest.xml
TestRequest1.xml
TestRequest3.xml

b) Child Process creation:

Here number of Child Processes given is 2.

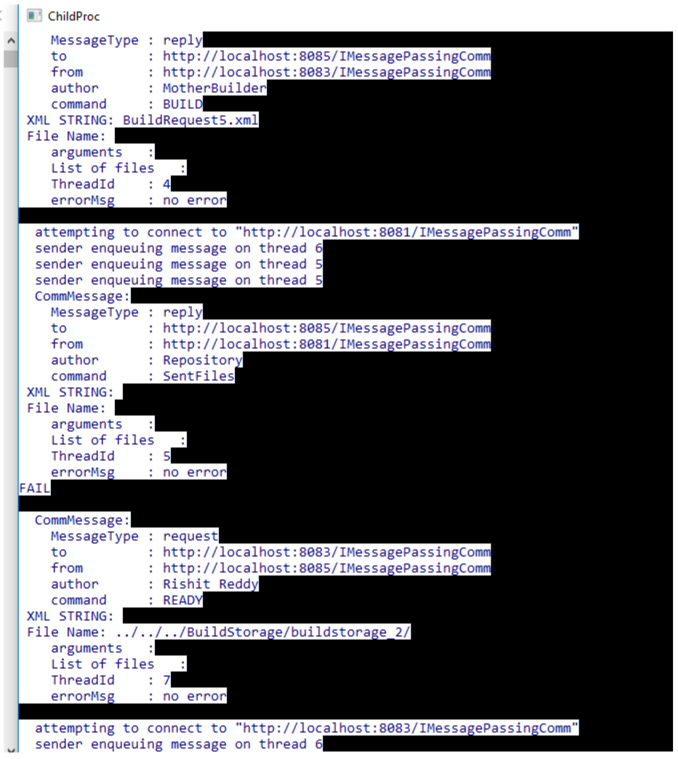


```
ChildProc
    MessageType : reply
    to          : http://localhost:8084/IMessagePassingComm
    from        : http://localhost:8083/IMessagePassingComm
    author      : MotherBuilder
    command     : BUILD
 XML STRING: BuildRequest4.xml
 File Name:
    arguments   :
    List of files   :
    ThreadId    : 5
    errorMsg    : no error

  attempting to connect to "http://localhost:8081/IMessagePassingComm"
  sender enqueuing message on thread 6
  sender enqueuing message on thread 4
  sender enqueuing message on thread 4
  CommMessage:
    MessageType : reply
    to          : http://localhost:8084/IMessagePassingComm
    from        : http://localhost:8081/IMessagePassingComm
    author      : Repository
    command     : SentFiles
 XML STRING:
 File Name:
    arguments   :
    List of files   :
    ThreadId    : 4
    errorMsg    : no error
FAIL

  CommMessage:
    MessageType : request
    to          : http://localhost:8083/IMessagePassingComm
    from        : http://localhost:8084/IMessagePassingComm
    author      : Rishit Reddy
    command     : READY
 XML STRING:
 File Name: ../../../BuildStorage/buildstorage_1/
    arguments   :
    List of files   :
    ThreadId    : 7
    errorMsg    : no error

  attempting to connect to "http://localhost:8083/IMessagePassingComm"
  sender enqueuing message on thread 6
```



```
ChildProc
    MessageType : reply
    to          : http://localhost:8085/IMessagePassingComm
    from        : http://localhost:8083/IMessagePassingComm
    author      : MotherBuilder
    command     : BUILD
 XML STRING: BuildRequest5.xml
 File Name:
    arguments   :
    List of files   :
    ThreadId    : 4
    errorMsg    : no error

  attempting to connect to "http://localhost:8081/IMessagePassingComm"
  sender enqueuing message on thread 6
  sender enqueuing message on thread 5
  sender enqueuing message on thread 5
  CommMessage:
    MessageType : reply
    to          : http://localhost:8085/IMessagePassingComm
    from        : http://localhost:8081/IMessagePassingComm
    author      : Repository
    command     : SentFiles
 XML STRING:
 File Name:
    arguments   :
    List of files   :
    ThreadId    : 5
    errorMsg    : no error
FAIL

  CommMessage:
    MessageType : request
    to          : http://localhost:8083/IMessagePassingComm
    from        : http://localhost:8085/IMessagePassingComm
    author      : Rishit Reddy
    command     : READY
 XML STRING:
 File Name: ../../../BuildStorage/buildstorage_2/
    arguments   :
    List of files   :
    ThreadId    : 7
    errorMsg    : no error

  attempting to connect to "http://localhost:8083/IMessagePassingComm"
  sender enqueuing message on thread 6
```

9. Project 1 deficiencies and changes made for final project:

There are quite a few deficiencies in the way I thought for project 1 and tried solving them for the final project. Some of them include:

- Didn’t mention details about the Communication channel which is very important for our project.
- Process pool was not described properly.
- Activities for each component were not identified.
- Added few key packages to the final project like Message Passing Communication Service and Interactive window for client with message passing.
- In project 4, class diagram is identified.
- Removed unnecessary packages like xml parser in the package diagram which are implemented directly in the larger components.

10. References

1. **http://deviq.com/build-server/**
2. **https://stackoverflow.com**
3. **http://wiki.rosalab.ru/en/index.php/Build_server_structure_and_workflow**
4. **Referenced Professor Fawcett’s solution for class diagram in midterm solutions**