Remote Build Server

Operational Concept Document

Version 2.0

CSE681-Software Modeling Analysis
Fall 2017

Author: Kaiqi Zhang
Instructor: Jim Fawcett
Date: 12-06-2017

Index

1 Executive Summary

This Operational Concept Document is about the design of Remote Build Server – an essential part of Software Collaboration Federation. The federation provides several different services to support continuous integration for the development team.

The collaboration system contains 4 different modules:

- Client with GUI
- Remote Repository Server
- Remote Build Server
- Remote Test Harness

This development will create a Remote Build Server, capable of building C# and C++ libraries, using a process pool to conduct multiple builds in parallel.

The implementation is accomplished in three stages:

- Project #2, implements a local Build Server that communicates with a mock Repository, mock Client, and mock TestHarness, 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.
- 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.
- Project #4, completes the build server, which communicates with mock Repository, mock Client, and mock TestHarness, 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
- using a single message structure for all message conversations between clients and servers
- managing EndPoint information for Repository, BuildServer, and TestHarness

All of 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, TestHarness, and Federation Client. Building these other Federation parts is beyond the scope of this development.

2 Changes and Deficiencies

2.1 Changes as my design evolved

2.1.1 Supports multiple tests in one request

In the Project #4 design, user can repeating select code files and add test to the build request structure. Thus, user can send multiple arbitrary test requests in one request message.

2.1.2 Use process pool for parallel building

In the new design, the build server using a process pool to spawn a specified number of child builders. The mother builder can decide which child builder to execute the building according to the balance situation. In this mechanism the build server implemented parallel building to handle high volume of build requests.

2.1.3 Store build requests on repo

When user create a new build request and send it to repository, the repo will save the request as a file with timestamp as name. So that user can browser and resend the same build request later. This design avoid user from creating duplicated requests.

2.1.4 Open and preview files on client GUI

In the client GUI, the client can get file list from repository, and user can double click on the files to preview the xml requests or logs. It's a convenient design to promote the user experience.

2.2 Design and Implementation Deficiencies

2.2.1 Endpoints are hard-coded

In current design, the ports and endpoints of client and servers are still hard-coded in source code. So, there is no way to modify them after compilation. This makes it difficult for administrators to manage the system.

A smarter way is store configurations in XML file. And client or servers ready the information from the XML file at startup.

2.2.2 Crashed child builder detection

In current design, if one of child builder crashed for any reason, the mother builder won't do anything to it. To enhance the stability, we should detect the running status of child builders and kill if any process crashed and became freezing.

2.2.3 Java and other language building

Currently the build server uses MSBuild for compilation, so it can only build C# and C++ source codes. In real life uses, it is very meaningful to add supports to build and test for other languages like Java.

3 Introduction

Software industry has been growing very quickly since 1970s. Nowadays a typical software company may develop tens to hundreds software projects every year. A typical commercial software product may contain up to 1 million SLOCs in 1 thousand packages, and hundreds of developers, managers, QAs, and customs could be involved in.

In the progress of software market growing, many development and management concepts were introduced, for example, Waterfall Model, Agile Software Development, etc. which makes the collaboration process an important role in the entire project development process. On the other hand, Continuous Integration has been very common in the current development process. It provides the following values: reduce risks; reduce human interfere; promote transparency; and enhance team connection.

To make the Continuous Integration process and project collaboration feasible, a Software Collaboration Federation is an essential tool for the software development team: developers submit and compile their work every day at the server; QAs can execute test tasks on the server; also, managers can monitor and follow up the progress on it.

3.1 Obligations

The main responsibility of the system is to store code submitted by developers and handle test requests from users. The main responsibility of Build Server is to build test libraries and test drivers based on build requests and source code files sent from the Repository.

- Execute tasks in a sequence of operations
- Accept files and store in temporary directory
- Build code into libraries
- Generate build report
- Automatically execute test process

3.2 Organizing principles

The organizing principles are to provide Client, Repository, Test Harness, and Build Server at the top level. And separate the program into isolated packages depend on single obligation principle. The Executive package call functions of different packages and execute tasks in a fixed sequence of operations, e.g. Executive controls the entire operation logic.

3.3 Key Architectural Ideas

The key idea is to implement a program to build source code into libraries. In this project, the system is constructed with C# language and .NET framework with Visual Studio 2017. The build engine is based on Microsoft Build Framework. With this framework, we can easily build Microsoft format solutions or projects in C# or C/C++.

4 Users and Use Cases

4.1 Uses of Different People

4.1.1 Developers

Code developers use this system to advance their work every day. Every day they use clients to submitted code to the Repository Server. Then they build them into libraries and do some basic tests like Construction Test or Unit Test.

Multiple clients connect to the same server, so separated environment should be provided to each developer. Also, their requests should be executed in parallel so that each person don't wait for too long time.

4.1.2 Quality Assurance

QAs or testers execute their job after code developers submitted their work. They can write test cases and upload them to the Repository Server. The test cases produced by QAs are much more comprehensive than those by developers. Then they send test requests to do Unit Tests, Integration Tests, Regression Tests, or other system tests.

The test procedures will be executed by server automatically. After the tests, QAs will get reports from the server. Thus, their work can be more efficient.

4.1.3 Managers

Managers use this system to monitor the progress of everyday works. They can coordinate resources and make decisions based on the submit and test result. They can also find out critical delays or problems at the early stage. Thus, the risks of the entire project can be reduced significantly.

4.1.4 TA/Instructors

Because this project is a course project, the TAs or Instructors will run the program to check whether all requirements have been meet. They will also check if the program runs fluently and do not crash or contains serious bugs. To make it convenient for grading, the program should provide an interface to show how it runs tasks by sequence, as well as how each requirement is meet.

4.2 Uses of Different Organizations

4.2.1 Personal Developers

For personal developers, the project won't be very large, maybe just contains tens packages. Each day several hundred lines of code are modified and uploaded. The system may be used for once or twice. The storage space for code and CPU resource used for compilation and test won't be a significant problem. Usually the system can be deployed at local machine, and only one active user exist at the same time in the system.

But the system can still help developer to keep everything on track. Also, the automatic testing process will save a lot of energy for the developer.

4.2.2 Small Development Groups

For small groups of teams, the compilation and testing tasks are still in small scope. But multiple users may use the system both in temporal scale and spatial scale. People's works begin to rely on each other. Modification and test request to a same package could be executed by different users at the same time. The group may not need an authentication system because they work openly and closely.

4.2.3 Medium and Large Organizations

For medium and large organizations, a considerable quantity of users may use the system at the same time. Thus, the work load of the server could be very high. And as the progress of development advanced day by day, the number of libraries stored in the server may grow to a very large size. Under this situation, the server should still be able to provide service to the users in an acceptable time.

Second, each user plays a different role and is configured to a specified authority in the system. So the authority management and security problem should be considered seriously in the system design.

Additionally, users of the system may come from different location, work in different platform. So, the system must be deployed in an ultra-stable environment, and is accessible in all kinds of circumstances. The system should also be very secure to defend attacks from all kinds of enemies, as the value of the information in the server could be very significant.

5 Application Activities

5.1 Activity Diagram of Top Level

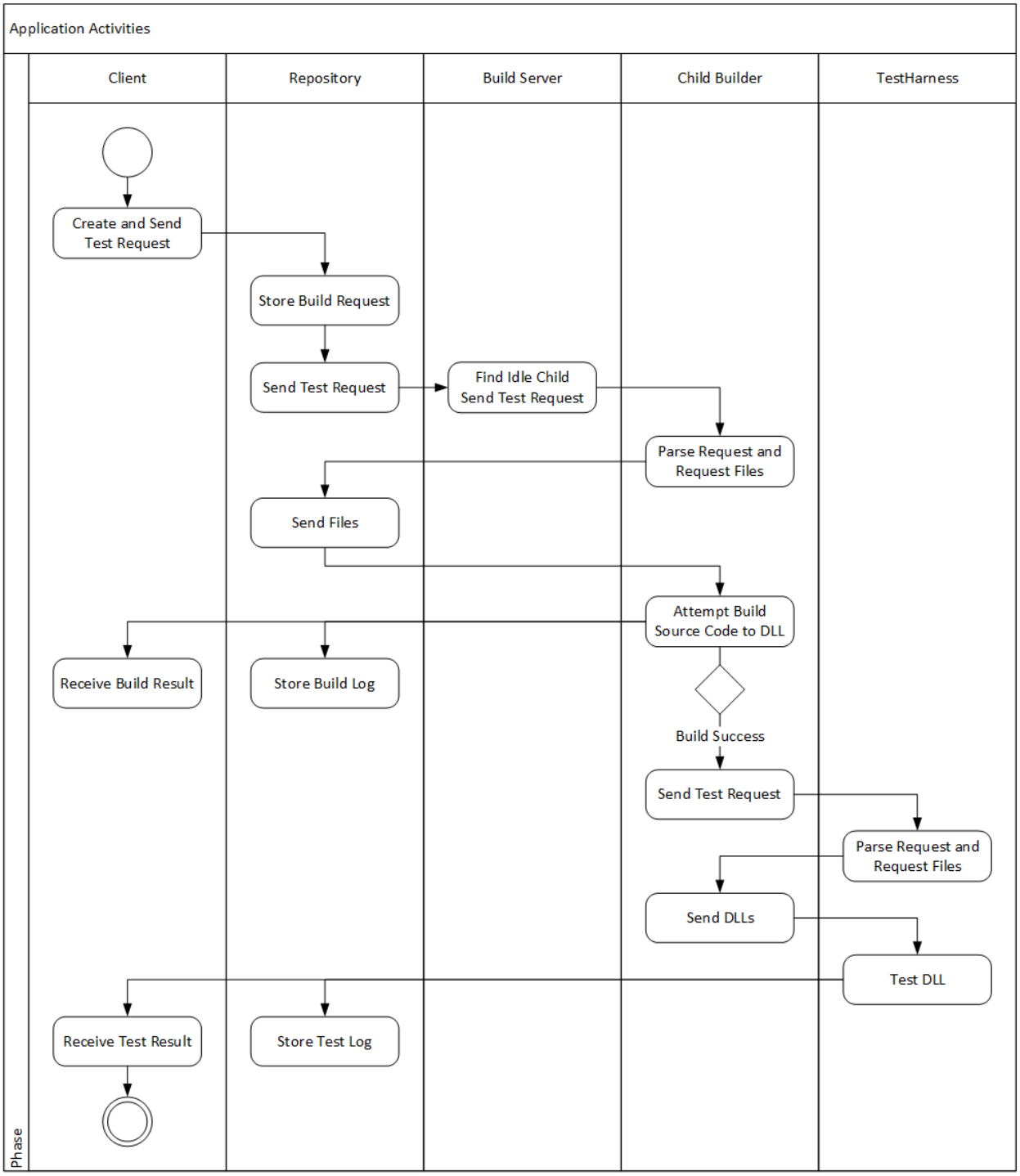

Figure 1. Activity Diagram for Top-level

The above swimming pool activity diagram shows the overview working flow of the entire software collaboration system. In the four swimming lanes are client, repository, build server and test harness.

1) First, client create test request and sent it to the repository server. The tested file and test driver source files may already exist on the repository. User can browser to files to build, and select target files to generate XML requests.
2) Then client create XML build request and send it to the repository to start build and test procedure.
3) When repository received build request, it saves it to XML file. Then file name consists the timestamp for user to distinguish. Then the repository sends the request to build server (mother builder).
4) When request is sent to mother builder, it looks for idle child builders and send the request to the idle child builder.
5) The child builder parses the test request to local format. Then request the repository for source code files.
6) Then the repository prepares the required files and sent them to the build server with the original test request.
7) After child builder accepted source code files, it attempts to build the source code and send the report to repository and client.
8) If build succeeded, it sends the DLLs to the test harness. Otherwise, it notifies the client of the failure.
9) When the test harness received command and libraries from the build server. It loads the libraries and execute tests and generate test report.
10) After the tests finished, the test harness sends the report to the repository. It also sends notification to the client with the test result.

5.2 Activity Diagram of Builder Server

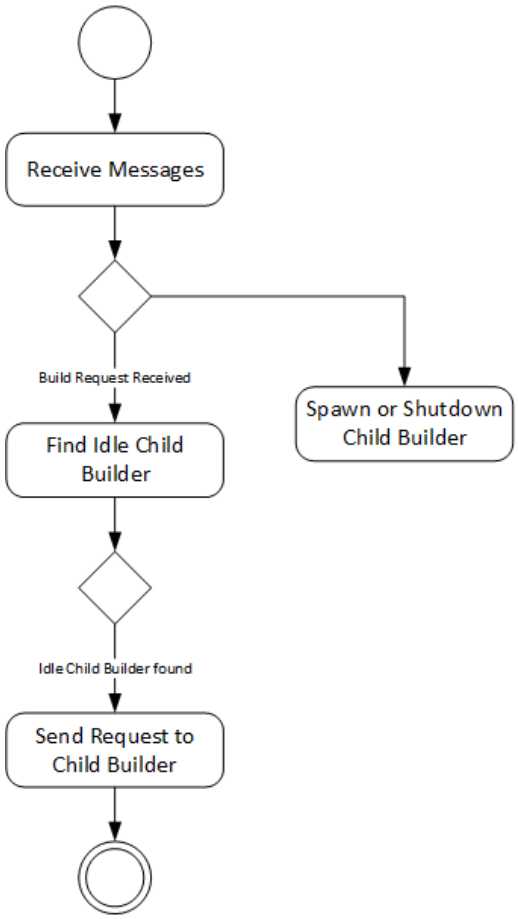

Figure 2. Activity Diagram for Build Server (Mother Builder)

The above diagram shows the working procedure of Build Server (Mother Builder)

1) The Mother Builder waits for WCF Messages from repository and client. The client may command to spawn a specified number of child builders, or shutdown all created child builders.
2) The repository may send build requests to Mother Builder. If a build request received, the mother builder looks for idle child builder from the list. If no idle child builder exists, it keeps waiting until any child builder is ready.
3) Then the mother builder sends the request to the idle child builder.

5.3 Activity Diagram of Child Builder

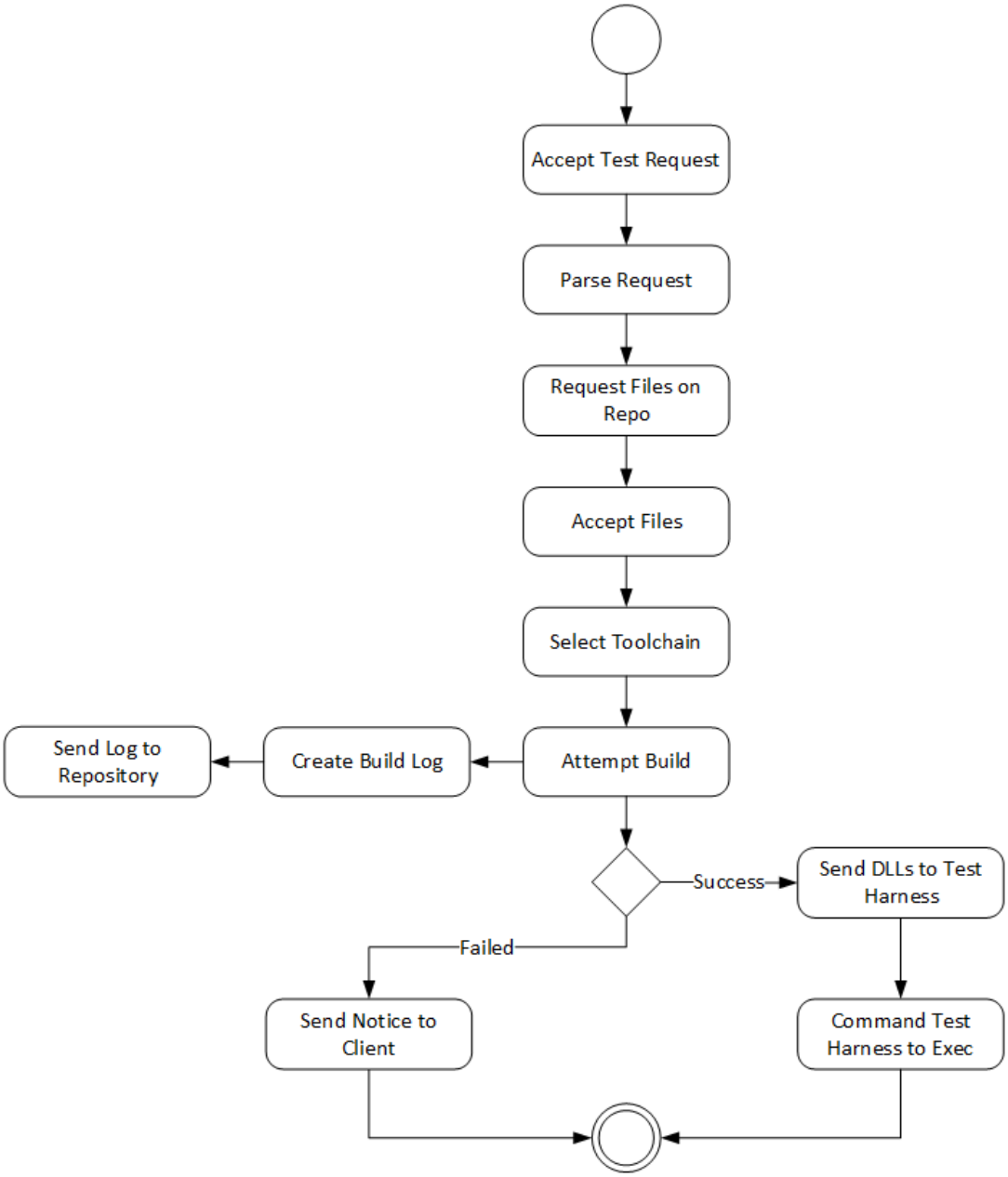

Figure 3. Activity Diagram for Child Builder

The above activity diagram shows the working procedure of Child Builder.

4) First, the build server accepted test request.
5) Then, it parses the request to see how each test is configured.
6) After that, it sends requests to the repository server for files in the build request and then receive the files.

7) After all files have been received, the builder prepares system environment and select the correct toolchain according to the content of request.
8) Then the builder attempt to build the project and generate build report.
9) When the build process finished, the build server notices the client with result.
10) If build succeeded, it sends the libraries to the test harness and commands the test harness to execute the test procedure.

5.4 Message Flow

This diagram represents the socket message flow between client and servers.

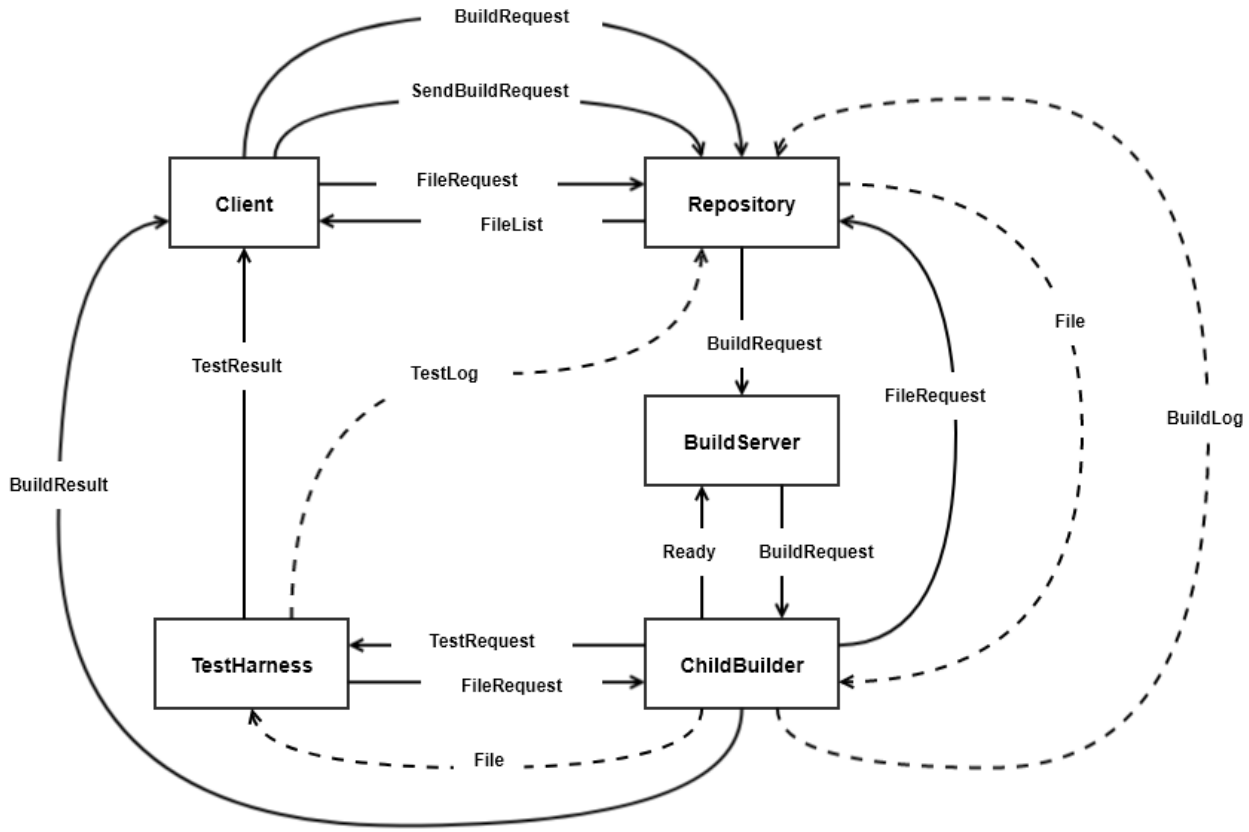

Figure 4. Message Flow

The name, source and destination, purpose, and content of each message are listed as below.

Name	Source	Destination	Purpose	Contents
BuildRequest	Client	Repository	Send XML string	BuildRequest
SendBuildRequest	Client	Repository	Command	Command
FileList	Repository	Client	Return file list	File list
FileRequest	Client	Repository	Get file list	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 file
BuildResult	ChildBuilder	Client	Notification	Build status
TestRequest	ChildBuilder	TestHarness	Send XML string	TestRequest

FileRequest	TestHarness	ChildBuilder	Get file	File name
File	ChildBuilder	TestHarness	Send file	file contents
TestLog	TestHarness	Repository	Send test log	Log file
TestResult	TestHarness	Client	Notification	Test status

6 Partitions

6.1 Top-Level Structure

The entire system is separated into 4 modules: Client, Repository, Build Server, and Test Harness. In project2, the Client is a simple executive, and the Repository and Test Harness is mock.

Below is the top-level block structure diagram of the system.

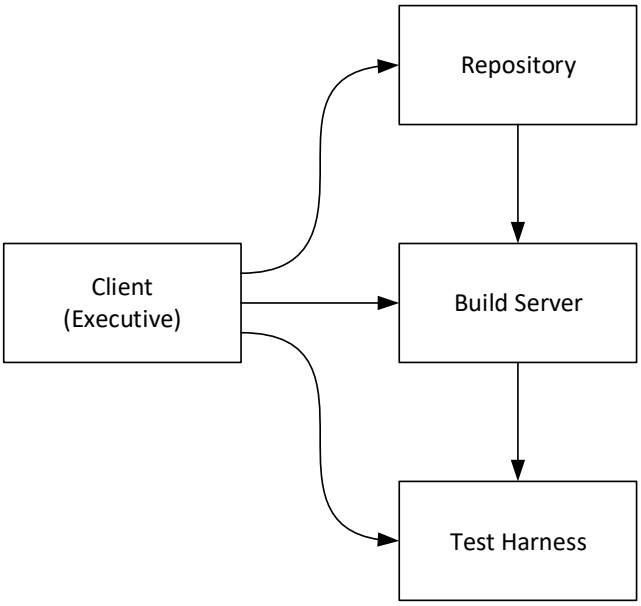

Figure 5. Block Structure of Top-Level

- Client

The user's primary interface into the system, serves to submit code and test requests to the Repository. Later, it will be used to view test results stored in the repository.

- Repository

Holds all code and documents for the current baseline, along with their dependency relationships. It also holds test results and may cache build images.

- Build Server

Based on build requests and code sent from the Repository, the Build Server builds test libraries for submission to the Test Harness. On completion, if successful, the build server submits test libraries and test request to the Test Harness, and sends build logs to the Repository.

- Test Harness

Run tests, concurrently for multiple users, based on test requests and libraries sent from the Build Server. Clients will check-in, to the Repository, code for testing, along with one or more test requests. The repository sends code and requests to the Build Server, where the code is built into libraries and the test requests and libraries are then sent to the Test Harness. The Test Harness executes tests, logs result, and submits result to the Repository. It also notifies the author of the tests of the results.

6.2 Package Diagram

The Build Federation is divided into different packages based on function decomposition. Each package is responsible for one specified task.

This section describes the functions and responsibilities of each package. Below is the package diagram of entire system.

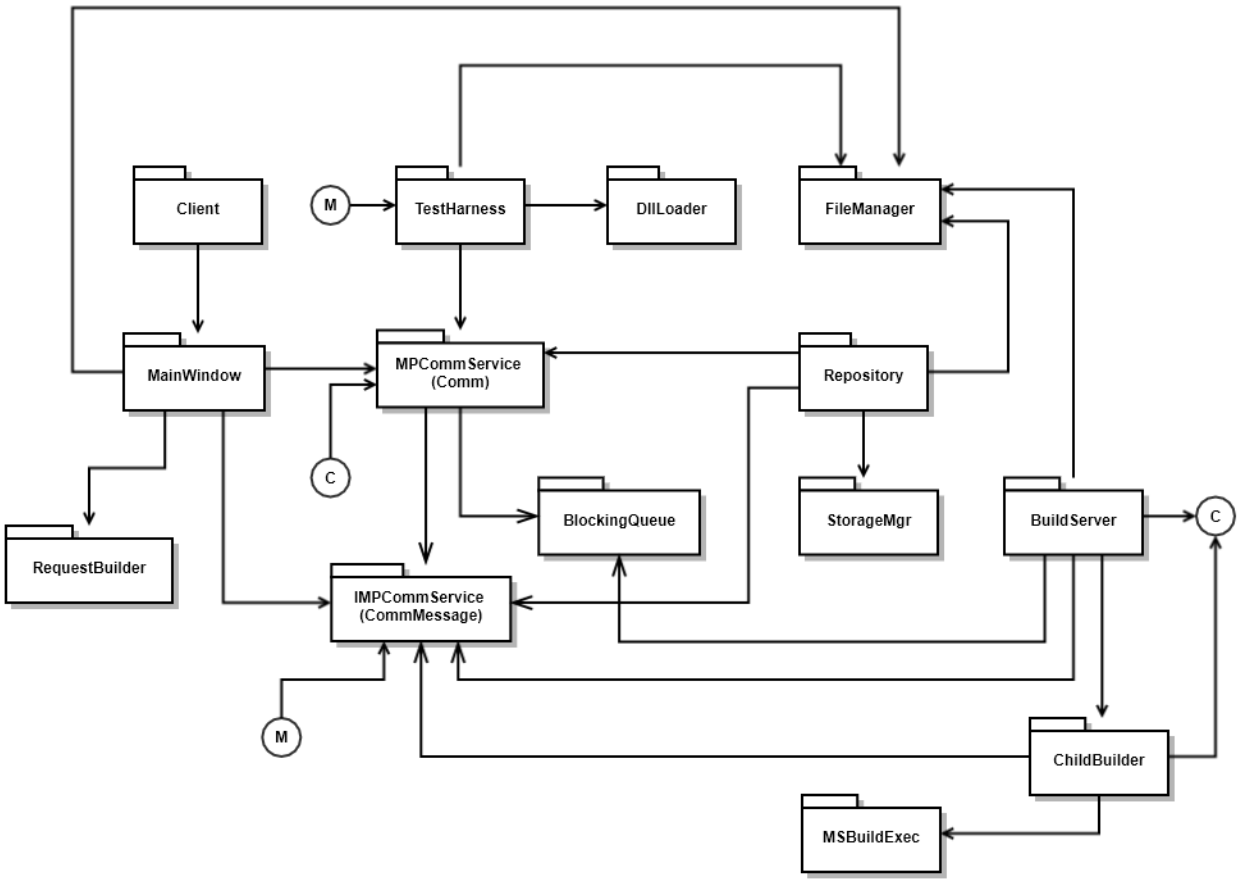

Figure 6. Package Diagram

6.2.1 Client

This package is the executive package of the client. It controls the main flow of the program and direct different packages to perform tasks in a specified sequence. It also prints logs to the console to demonstrate whether all requirements have been met.

6.2.2 MainWindow

The graphic user interface (GUI) of the client. It provides an interface for user to operate the build federation.

The client can send command to build server to create a specified number of processes or shutdown all processes in process poll. It can get and display all the files on repository server. It can create build requests depending on the code files user selected. It can open and view log files on the repository server. It can also receive the notification from build server and test harness to see the build and test results.

6.2.3 RequestBuilder

This package is responsible to generate build request from selected code files. The test request contains meta information like: author, timestamp, testDriver, testTarget, etc. And one TestRequest may contains multiple TestElements.

A typical test request is like below:

```
<?xml version="1.0" encoding="utf-16"?>
<TestRequest>
  <author>Kaiqi Zhang</author>
  <dateTime>9/7/2017 9:49:14 PM</dateTime>
  <tests>
    <TestElement>
      <testName>Test1</testName>
      <buildTool>MSBuild</buildTool>
      <buildConfig>Test1.csproj</buildConfig>
      <testDriver>td1.cs</testDriver>
      <testCodes>
        <string>tf1.cs</string>
        <string>tf2.cs</string>
        <string>tf3.cs</string>
      </testCodes>
    </TestElement>
  </tests>
</TestRequest>
```

6.2.4 Repository

The mock repository stores files for the build federation. This package receives build requests from client and send it build server. Other nodes can also store and retrieve files on it. For example, child builder retrieves source code files from it and send build logs to it for storage; test harness send test logs to it for storage; client retrieves requests and logs from it.

6.2.5 StorageMgr

The StorageMgr package manages the code, build request xml and log files on the repository server. It can browser the directory, create directory or clean the files.

6.2.6 BuildServer

BuildServer (Mother Builder) creates a process pool to distribute build tasks. It spawns a specified number of child builder processes, and use blocking queues to deliver build requests to idle child builders, so that it implements a parallel building mechanism.

6.2.7 ChildBuilder

ChildBuilder do the real building process. It parses the build request, and requests for code files from repository server. Then it calls build engine according to the config information in the build request to execute the building process. After each test element building finished, it sends test request to the test harness and notify the client of the build result. It also stores the build logs to file and send to repository for storage.

6.2.8 MSBuildExec

This package uses MSBuild to build C# source code on Windows. It reads the build configuration from csproj file, and build with .NET framework.

It's a good and easy to use build engine for this project.

6.2.9 TestHarness

TestHarness loads and test DLL files, and generate test logs. In this package, it can parse the TestRequest and retrieve DLL files from child builder. Then it calls DllLoader to do the test process. After test process finishes, it stores the log information to file, and send the log file to repository. It also notify the test result to client after that.

6.2.10 DllLoader

DllLoader can load and test C# DLL files. It uses the reflection feature of C#. It loads and reads the meta information from CLR codes and find out the test interface. Then executes the test interface to test and get test result.

6.2.11 FileManager

After getting the test requests, the server needs to create temporary directories and copy files. And at the end of the entire process, the server needs to delete the temporary files and directories. The FileManager package is responsible to manage the directory/file creation and deletion.

6.2.12 BlockingQueue

This blocking queue is implemented using a Monitor and lock, which is a decent solution for multi-thread accessing. In this project, blocking queue is used as a message buffer for WCF communication. It is also used for build server to maintain the ready status of child builder processes.

6.2.13 MPCommService

This package implements the details of WCF communication interface. Processes can send and receive requests and commands, post or get files using this package via WCF.

To define the behaviors of each request or command, developers only need to use a delegate structure to add dispatchers to the service.

6.2.14 IMPCommService(CommMessage)

This package defines the Data Contract and Service Contract used for WCF communication. It is an interface of MPCommService.

In this package, a unified message structed is defined that contains information like To and From address, Command string, List to hold arguments, etc. With this message structure, processes can easily communication with each other via WCF service.

6.2.15 TestUtilities

This package provides some helper functions shared by many other packages. E.g. string format. Most of the methods is static.

7 Critical Issues

This section discusses about several critical issued that have important influence on the system usability and stability. It also discussed the potential solutions to the problems.

7.1 Multi-Language and Multi-Platform Building

In the practical utilization, different programming languages will be used in different projects. Also, the deploy target could be different for each request. Some programs run on Windows, some runs on Linux, and some on iOS, even Embedded OS. The compilation toolchains and dependent libraries may change one by one. The system should be able to switch between different requirements.

Solution: For each build task, the required building toolchain and system environment should be reported in the request. For example, a Linux C++ build task should ask a Linux compilation toolchain; a embedded C++ build task should ask a cross-compile toolchain.

On the other hand, to make the build environment clean before each building, we can copy a specified Virtual Environment based on each build requests.

7.2 Scaling Build Process for High Volume of Build Requests

In many circumstance, a build requests contains multiple packages to be built. Also, several build requests or test requests could be sent to the server at the same time by different users. It's unreasonable to build packages in single thread one by one because some heavy tasks may occupy the resources for too long time.

Solution: To build packages parallelly, Thread Pool or Process Pool could be introduced to solve the problem. Build tasks are created and inserted into the pool, then the system decided how many tasks run parallelly and the sequence.

In my implementation, a Process Pool is used to spawn specified number of Child Builders. The Build Server will find an idle Child Builder and send the request to it. When Child Build finished its job, it sends a Ready message to notify the Build Server.

7.3 Message Structure for Message Conversations

According to the message flow design, there are many kinds of message types between client and servers. To simply the message structure, it's reasonable to design a combination message structure that meets all the requirements of communication.

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

7.4 Managing Endpoint Information

For client and servers to communicate with each other, each program needs to know the ports and endpoints of others. Of course, we can hard-code the information into the source code. But it makes difficult for system administrator to manage the configuration.

Solution: Store endpoint information in XML file resident with all clients and servers. When client or servers start up, they can load the information from XML file. So that the ports and endpoints can be easily configured after deployment.

8 Reference

1) http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Project1-F2017.htm
2) http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/Note-SizeMatters.htm
3) https://en.wikipedia.org/wiki/Continuous_integration
4) https://en.wikipedia.org/wiki/Agile_software_development
5) https://msdn.microsoft.com/en-us/library/dd393574.aspx
6) https://travis-ci.org/
7) http://ecs.syr.edu/faculty/fawcett/handouts/webpages/blogTesting.htm

9 Appendix. Project Builder Prototype

9.1 Prototype Design

This prototype is a small program to demonstrate programmable builds. It loads a Visual Studio Solution (C#) file from a specified path and build the solution. It also writes the build report to the console and log file.

The target solution file path is hard-coded into the source file. You can also provide a path for the file log. The current test target project is at "BuildStorage/HelloWorld/HelloWorld.sln".

```
54      // target solution path
55      string slnPath = @"../../../BuildStorage/HelloWorld/HelloWorld.sln";
56      // target file log path
57      string logPath = @"../../../LogStorage/log.txt";
```

The build engine is based on MSBuild (Microsoft Build Engine). Several Microsoft.Build classes are imported.

```
16  using Microsoft.Build.Construction;
17  using Microsoft.Build.Evaluation;
18  using Microsoft.Build.Execution;
19  using Microsoft.Build.BuildEngine;
20  using Microsoft.Build.Framework;
21  using Microsoft.Build.Utilities;
```

The solution building configuration is chosen with the GlobalProperty. By default the configuration is "Debug | Any CPU".

```
35      Dictionary<string, string> GlobalProperty = new Dictionary<string, string>();
36      GlobalProperty.Add("Configuration", "Debug");
37      GlobalProperty.Add("Platform", "Any CPU");
```

Two loggers are registered to the building process. One outputs result to console, the other outputs result to a log file specified by the user.

```
30      // logger log to console
31      ConsoleLogger cl = new ConsoleLogger();
32      // logger log to file
33      FileLogger fl = new FileLogger() { Parameters = @"logfile=" + logfile };
```

If run SolutionBuilder.compile() and build succeeded, the function will return "Success", otherwise it will return "Failure";

9.2 Outputs

The figure below shows the output result of running builder prototype:

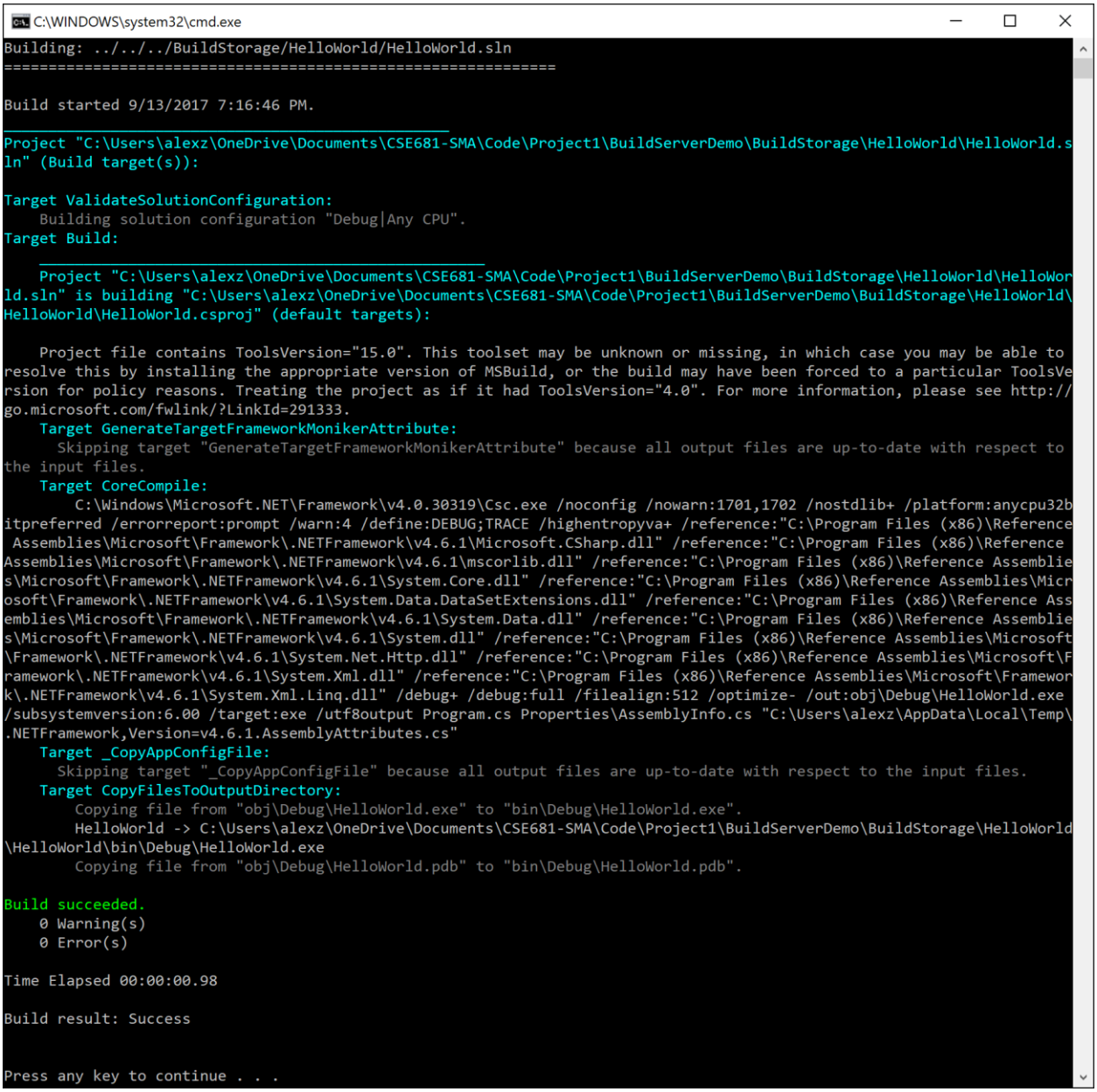

Figure 7. Builder Prototype Output