Remote Build Server
Operational concept document

By Biao A (SUID:266139645)
Instructor: Dr. Jim Fawcett
DATE: 6th Dec 2017

Index

1. Executive Summary

This document aims to illustrate some important aspects of Remote Build Server. It will also show some connections between Remote Build Server and other major parts in this system or the users.

In the process of developing a big software, we need to partition code and test them one by one. For this purpose, a federation which contains Repository, Build Server, Test Harness and Client is needed. The aim of this project is to create a Remote Build Server, an automated tool that builds test libraries for the whole federation.

Users of Remote Build Server are mainly TAs and graders, Developer and Quality Assurance and other parts of this federation. Each of them may need different functions. For example, Repository needs to send test requests and test files. Also, Repo need the logs written by Remote Build Server. Other vital uses are listed in section 3.

There also exists some critical issues need to be discussed and given solutions.

- Performance
  Performance of Remote Build Server plays a very important part in this system. Every test files need to be built before being sent to Test Harness. So, the speed of building determines the whole schedule, even determines what time this software is completed. Consequently, the performance represents the economic profit.

- Define a qualified message structure
  Because this build server and other part in remote, we need WCF to provide message transmission. As all parts in this federation use one WCF service, so it is necessary to define a single message structure that works for all need in whole federation.

- Complexity
  In my mind, complexity in RBS may represent two things. The first one is about the structure of whole system and the second one is about the interface to users. The former is important to developer and the latter means a lot to users.

- Reliability
  Reliability is important in every system which make sure system will not easily crush. For example, when user incorrectly use the GUI, it should not crush but return a warning message.

- Managing root, port information
  When facing a large federation, the amount of root, port and some other information will be large. Inserting the path code in each project is a bad idea because it will reduce the maintainability.

- File Transfer
  In this system, file transfer is very common. A common issue is that the program will use a file before the completion of its transmission.

2. Introduction

2.1. Application Obligations

This part presents some major obligations of Remote Build Server. Other additional functions or uses designed for other parts in this federation will be listed in the later part of this document.

As its name shows, the major function of the Remote Build Server is to build the test libraries. For this purpose, the function to build libraries is needed. Also, it should be able to receive test files or build requests from Repository and send test libraries and test requests to Test Harness. In addition, a process pool is needed to deal with large number of requests. Build Server needs to assign requests to process pool. Actually, the Build Server split to two parts in this project: Mother Builder and Child Builder. The jobs of Mother Builder are to receive request and send it to a Child Builder and Child Builder build dll files and send it to TestHarness. The detailed obligations will be present in section 5.

2.2. Organizing Principles and Key Architectural Ideas

Nowadays, to create a new software or system, the amount of the code might be very large, even to million lines of code. To make sure this new software implements successfully, we need to test codes, but, due to some reasons (such as the capacity limitation of computers and time schedule), testing the whole software is unrealistic. To solve this problem, it is necessary to partition codes to proper size that we can handle and test them before developers put them in the software baseline. For this purpose, we need a system or a federation which has enough capacity to test these codes. This federation has four vital parts: Client, Repository,

Build Server and Test Harness. The Remote Build Server will be mainly described in this document.

The main process of this federation is: First, Client will build and send request to Repository and command Repository send build request to Build Server. Second, Build Server will build dll files and send logs to Repository for Client to look. If build success, dll file will be sent to Test Harness to test. Third, as same as the Build Server, Test Harness will create log and send it to Repository.

In this project, each function above and other functions displayed later could be consider as an activity. Each activity is a part of whole system and their connection determines whether the running of this system is healthy or not. In this design, each activity will be assigned to one package. In the end the whole project will have a clear structure.

3. Users and Uses

This part displays some mainly users of Remote Build Server and some functions designed for these users.

3.1. TAs and graders

In the context of this course, TAs and graders uses the project to evaluate student performance for this course. They will check requirements, evaluate structure and design, and analyze code metrics.

function design for this user:

- Clear demonstrate the code automatically

- Write many prologue and comment
- Clear show which requirement I have implement

3.2. Client, Repository and Test Harness

Build Server needs to collaborate with the other parts in this federation, so the whole system can run healthily. Client needs to check the notifications or logs send by Build Server. Repository needs to store the logs of build result. Test Harness needs the command from Remote Build Server to execute. Also, it requires the test libraries built by Remote Build Server. With the commands and test library, it will run test, sends notifications to Client and sends test logs to Repository.

function design for this user:
- Send and receive files and message
- Build dynamic linked library
- Create logs

3.3. Developer

To test every line of code, developer should make sure the Remote Build Server runs well. When something wrong (such as some building error happened), Remote Build Server should send notes to Developer. Developer can browse logs in Repository later.
In reality, amount of the test codes could be very large. After Developer send the code and test request to Repository, Remote Build Server might build test libraries all night waiting for Developer to check logs next day. In this process, developers do not want to see that building has stopped because the Build Server has crushed.

function design for this user:
- A GUI for developer to use

- Build Server need to have high reliability. It will be discussed in critical issues.

3.4. Quality Assurance

Quality Assurance's duty is to make sure that everything is going well. It includes a lot of meanings. QA will be submitting test request messages for large bodies of code. It needs Build Server to be fast enough. Also, QA need to see the logs or notifications.

function design for this user:

- Create logs and notifications
- Fast enough (using process pool)

4. Application Activities

Some activities diagrams are displayed in this part which show some important activities related to Build Server.

4.1. Activity Diagram for Whole Federation

The diagram below shows some high-level activities in this federation. Some detailed activities will be shown later in this section.

First, Executive creates components using process class. Actually, this step will only be operated when demonstrate this project, because different parts in this system may not be local. Then, Client browses file list in GUI and selects the files to build a request. After that, Client can send requests

to repository to store the request. Second, Client chooses the number of child BS and commands Repository send request to BuildServer. When Build Server receives the request, it will assign it to child builder (using process pool). Then, child builder will parse the request and ask repository for files. When the building ends, if succeed, child builder will send log to repository and send dll file to TestHarness. Concurrently, it sends a ready message to Mother Builder being ready for next job. Second, when has received the request, TestHarness will ask child builder for files and create a appdomain to do test. It will create a log for this test and send it to repository.

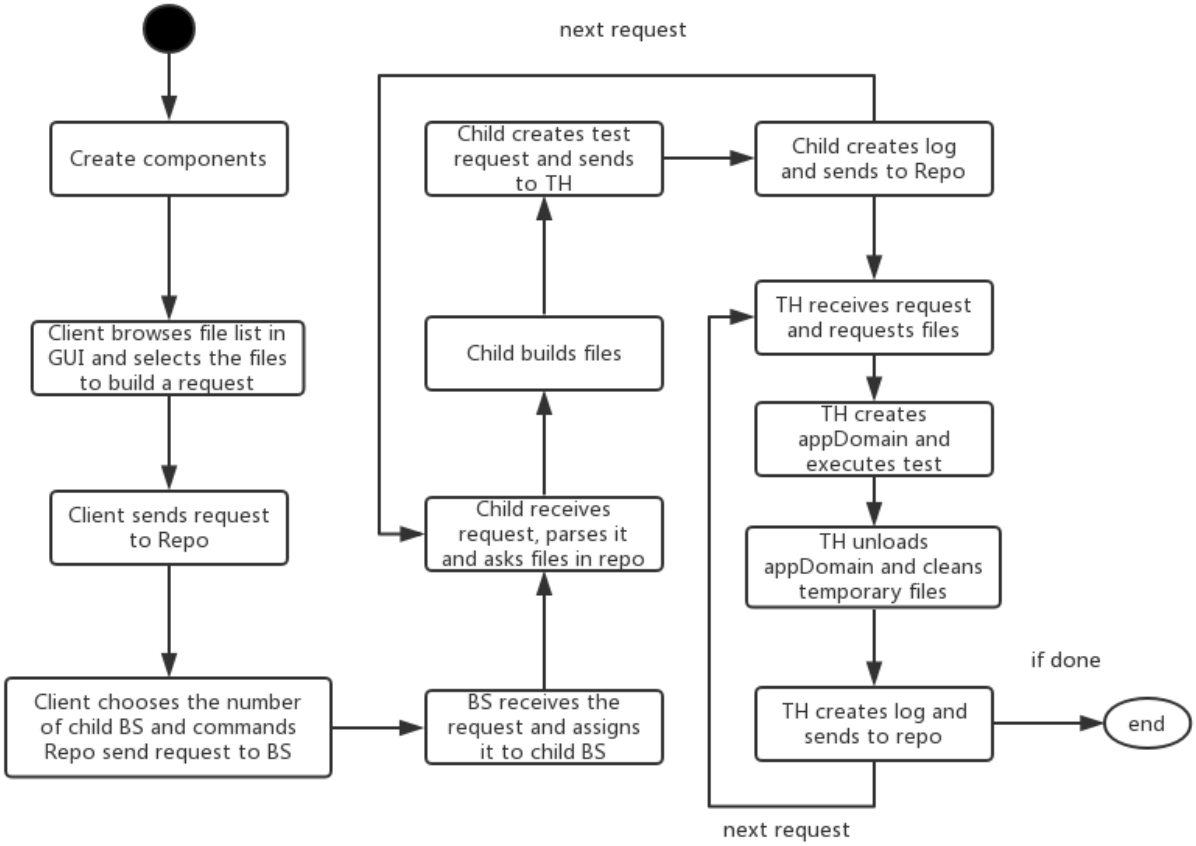

Figure: Package diagram for whole federation

4.2. Model and Activity Diagram of Process Pool

The build server may have very heavy workloads just as we mentioned. We want to make the throughput for building code as high as is reasonably possible. To do that the build server will use a "Process Pool". That is, a limited set of processes spawned at startup. The build server provides a queue of build requests, and each pooled process retrieves a request, processes it, sends the build log and, if successful, libraries to the test harness, then retrieves another request.

Malformed code may cause one of the processes to crash, perhaps by a circular set of C++ #include statements which overflow the process stack. This however, won't stop the Builder, which simply creates a new process replacement, and reports the build error to the repository. Note that the process pools will need to communicate with the mother Builder process.

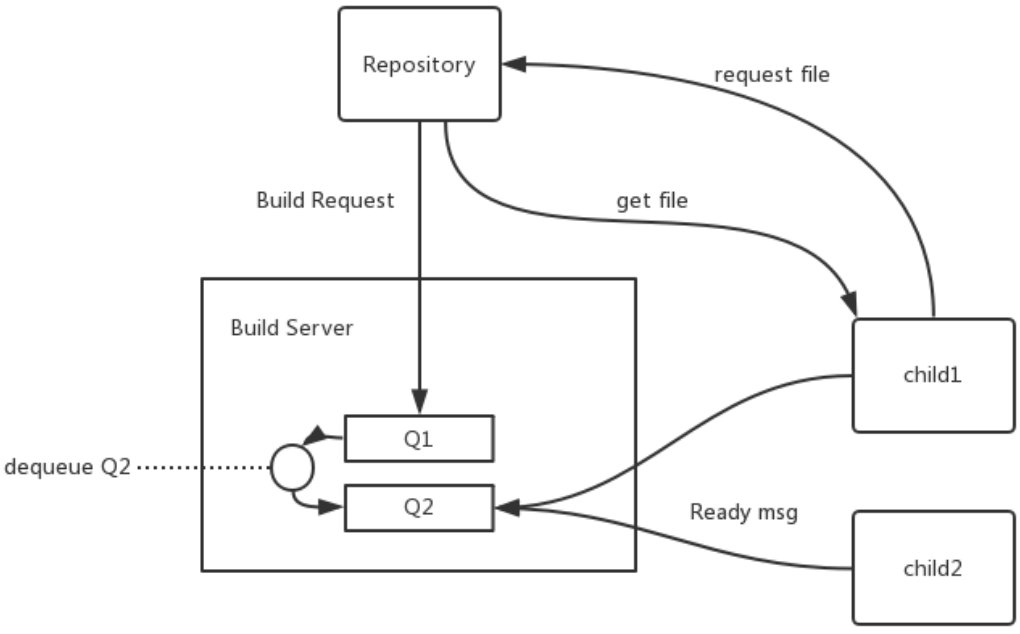

Figure: Process Pool Model

In this Figure, on command of Client, Repository send build request to the Q1 in Comm of Build Server. When Build Server get a build request from its Q1, BS will dequeue Q2 (a queue storing the ready message of child build server) and assign build request to child build server. When get the build request, the child will ask Repository for file. Then, a ready message will be sent by child when it has completed its work. The activity diagram is shown below.

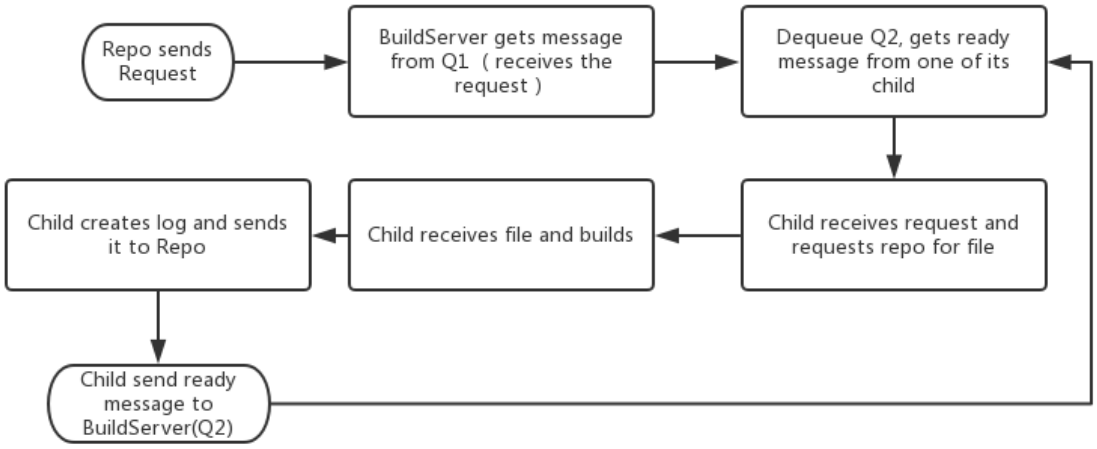

Figure: Activity Diagram for Process Pool

4.3. Message Dispatcher

In this project, for communication, Repository, Client, Test Harness, Build Server and Child Builder all have a message dispatcher. The main duty of message dispatcher is to recognize message and handle it with right functions. A message dispatcher can be simply implemented with a Dictionary<Msg.command, Action<Msg>>. The Msg.command defines the type of processing needed for a message, and the Action<Msg> defines the processing used to handle that message. When get a message, dispatcher will map this message to a function by its Command.

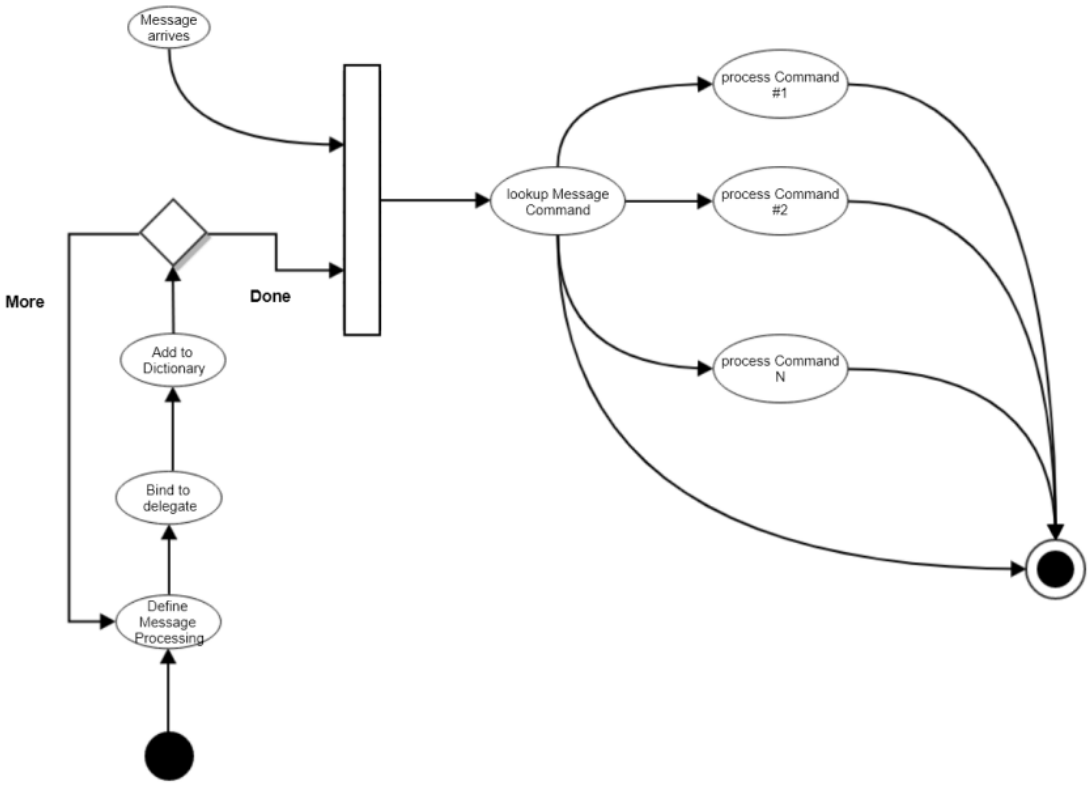

Figure: Activity Diagram for Message Dispatcher (from MT1Q7)

5. Partitions

Partition is the core idea of a design. By establishing this package diagram before writing coding, developers will get a blue print of this system, which is helpful. The graph shows the main packages and their interaction.

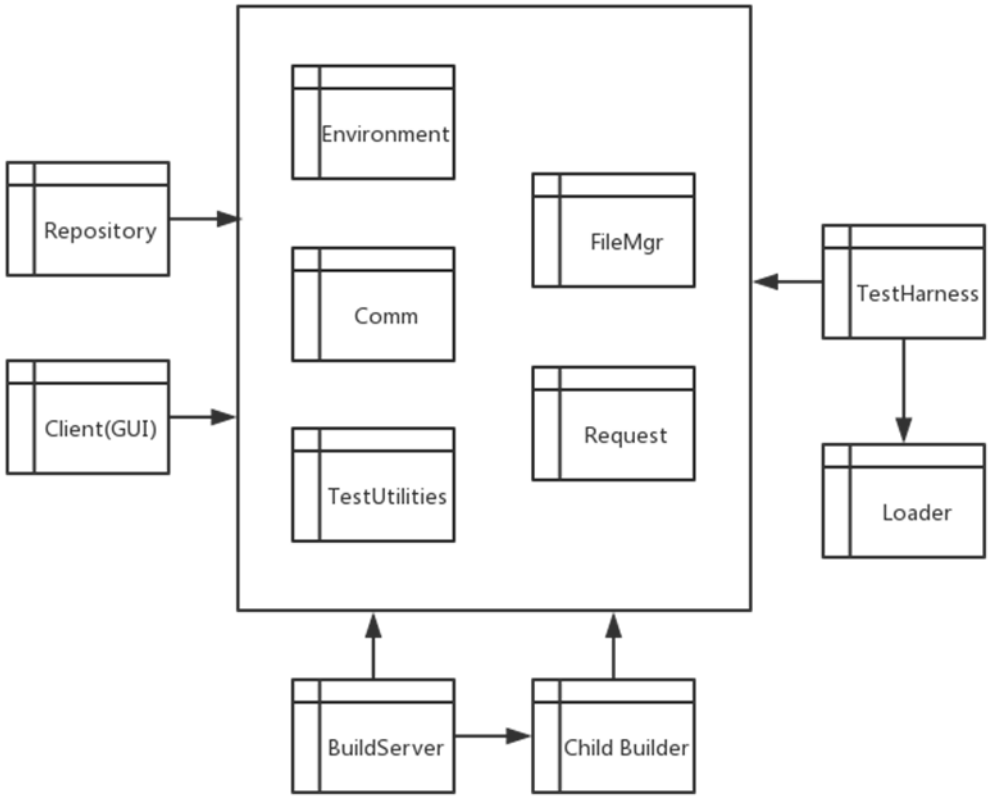

Figure: Package diagram for this federation

For ease to understand, I draw this diagram like this, but I have ignored some details. The Repository and BuildServer do not have connection with Request, because they do not need to parse the request.xml. In my design, they just transfer .xml files to another place. Also, the Comm package contains some important subparts (I will show it later).

Packages and their interactions:

- Comm

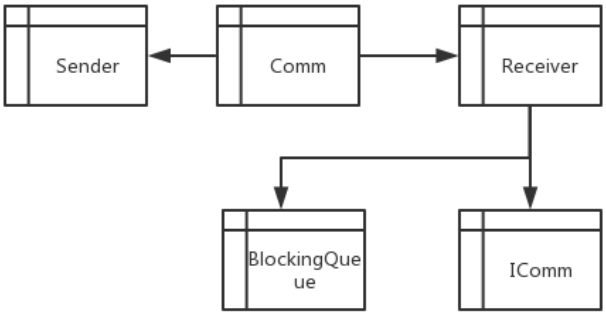

Figure: Detailed Package diagram for Comm

- This package includes four class and one interface which are all dedicated to information transmission. IComm is an interface including the service contract and data contract of WCF. Comm integrate the Sender and Receiver, which makes sending and receiving message more convenience. BlockingQueue is a class including a thread safe queue, which is used in message transmission. One can get message from this queue only when someone put a message in it. The Comm package is used by all parts in this federation which use WCF.

- Request
  This package mainly deals with two kinds of request: build request and test request. It contains functions like creating a xml file, parsing a xml file and loading a xml file. It is used by Client, Child Builder, and TestHarness.

- FileMgr
  This package allows other parts to get information about its local files. For example, repository use it to get the file list and send this list to Client. Also, this package provides function to delete file. TestHarness

can use it to clean temporary files it created. It is used by Client, Repository, Child Builder, TestHarness.

- Environment
  This package has no function in it. It contains all path, root and other information. With this public data structure, you can easily manage these kinds of information. It is used by Client, Build Server Repository, Child Builder and TestHarness.

- TestUtilities
  This package includes some functions which will be used many times in this whole project such as printing a title. So, I package them all for the ease to manage and use. Some functions of it can help me easily to demonstrate this project. It is used by Client, Build Server Repository, Child Builder and TestHarness.

- Repository
  Repository mainly provides function to store files such as logs, requests, and test files. Almost every part in this federation send files to it. Also, using its message dispatcher, it can receive some commands from other parts. For example, Client can browse its files through GUI by sending a message to repository and repository will send a file list to it.

- Client
  This package includes a GUI for user to control all this federation. By using this GUI, you can browse the files in repository, select some of them to build a request and send them to Repo. Also, you can use GUI to set number of child processes and start the process pool.

- BuildServer and Child Builder

  These two are the core of the whole federation. BuildServer receives the build request from repository and assigns the work two child processes. Then the child process will request repository for test files. When the building ends, if succeed, Child Builder will send log to repository and send dll file to TestHarness.

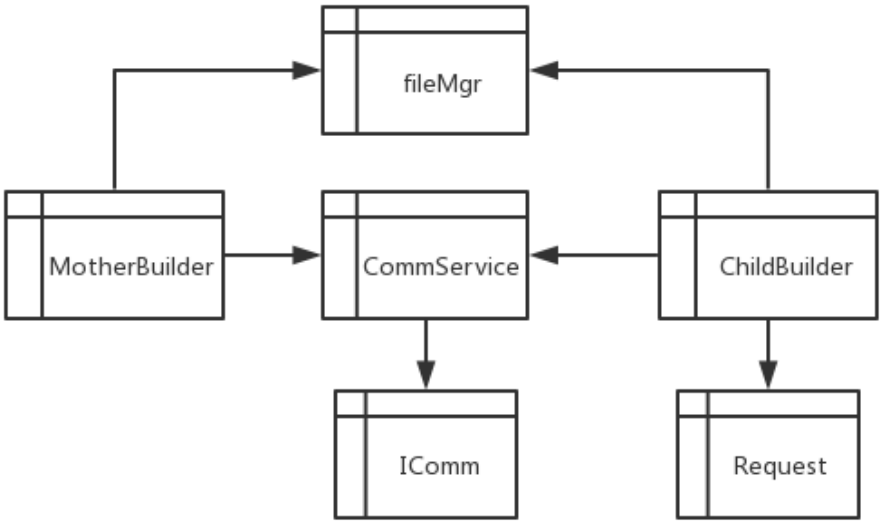

Figure: Detailed Package diagram for Process Pool

- TestHarness and Loader

  In this project, TestHarness use appdomain to test. Appdomain can avoid test harness crush, because it is running in the isolated space. Loader is responsible for loading the test files from Child Builder. It will load the dll files and using reflection to invoke the method in it and return value. Actually, loader is a dll file, too. TestHarness loads loader to test each file it received and sends logs to repo.

6. Critical Issues

Some of issues in this part showed up when I wrote this project, so I list them here.

6.1. Performance

As mentioned in executive summary, speeding up RBS is important. Every test need to be built before being tested. When QA or Repo send large amount test to RBS, it is supposed to have the capacity to build them on time.

Solution:

Usually we can use process pool to solve this issue. By dealing with multiple tasks in the same time, we can save lots of time.

6.2. Define a Qualified Message Structure

As all parts in this federation use one WCF service, so it is necessary to define a single message structure that works for all need in whole federation.

Solution:

I use the list below to construct my message structure:

Data Member	Type	Instruction
Source	String	The source of message such as “Repo”
Destination	String	The destination of message
Address	String	The host address of message source
MsgList	List<string>	A list carrying information such as file lists from repo to client.

Body	String	The main body of message
Command	Command(string)	Key value for dispatcher to recognize the message

So, for all message passing in this whole system, this message structure is qualified to do all its job.

Name	Source	Destination	Purpose	Content
FileRequest	Client	Repository	Get file list	Command
FileList	Repository	Client	Return file list	File list
BuildRequest	Client	Repository	Store BuildRequest	Request.xml
SendBuildRequest	Client	Repository	Command	Command
BuildRequest	Repository	BuildServer	Send BuildRequest	Request.xml
BuildRequest	BuildServer	ChildBuilder	Send BuildRequest	Request.xml
ReadyMsg	ChildBuilder	BuildServer	Notification to BS	Ready status
FileRequest	ChildBuilder	Repository	Get file	File name
File	Repository	ChildBuilder	Send file	File contents
BuildLog	ChildBuilder	Repository	Send log	Log.txt
TestRequest	ChildBuilder	TestHarness	Send TestRequest	TestRequest
FileRequest	TestHarness	ChildBuilder	Get file	Command
File	ChildBuilder	TestHarness	Send file	.dll file
DirRequest	Client	Repo	Need dir list to shows	Command
TestLog	TestHarness	Repository	Send string	Log.txt

6.3. Complexity

In my mind, complexity in RBS may represent two things. The first one is about the structure of whole system and the second one is about the interface to the users.

Solution:

For the structure, during the designing, the whole structure should be clear, which means that each package is supposed to have a clear duty and the connection between them must be rational. In detail, codes need to be readable and cannot be too long, which is convenient for the maintain afterwards. Also, the prologues and comments are supposed to be ample to explain every detail, which means a lot for future modifying.

For the GUI, it needs to be well designed. Each button should have clear function and some user guilds need to be prepared if necessary.

6.4. File Transfer

Before using one file, program should make sure the completion of transmission of file. When one part of federation sends a file to another part, you must concurrently send a message to it so that it will know that "some files will come". Under this situation, when the receiver gets the message, it is possible that the transmission of files is uncompleted which will cause exception.

Solution:

There are two solutions: One feasible solution is sending a message to receiver when the filestream is closed. I do not implement this one in my design, because it need to define a new message each time you send a file. Another one is adding a method in TestUtilities which will check files over and over again until its transmission is completed. When someone need to use a file transferred from others, it must call this method.

6.5. Managing root, port information

When facing a large federation, the amount of root, port and some other information will be large. Inserting the path code in each project is a bad idea because it will reduce the maintainability. Each time you use a new folder for files, you have to check every corner of code to fine a path code and change it. One solution is to use an public Environment data structure to contain all information and let all parts in federation add a reference to it. When you want to do some changes, you can only change the content of this public data structure.

6.6. Reliability

Reliability is important in every system which make sure system will not easily crush. Also, reliability means a lot to user experience.

Solution:

It is necessary to use try… catch… syntax, which is easy to see what happened when encounter a problem. Also, process pool and appdomain are good options. They guarantee the partly crush will not affect the main process and the main process can restart them any time. For this project, one important part is WCF communication. To avoid the situation that when the listener is not there, the sender will crush, message senders are supposed to try many times using try…catch…, until they connect to listener or inform user that listening host is not open.

7. Changes and Deficiencies

In project 1, some functions I design are not feasible in pro4, so I delete them from my OCD. I add some new activity diagrams (process pool) and package diagrams (a whole new one) into it because from pro2 to pro4 I gradually get a clear idea about this whole project. There are some deficiencies in my project. Sometimes it will take a long time for file transmission and I have not figured out why. Also, the console may show some useless information.

8. Reference

http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures