Remote Build Server (OCD)

Project 4

Sachin Basavani Shivashankara (SUID: 267871645)
CSE-681: Software Modelling and Analysis (Fall 2017)
12/6/17

Instructor: Dr. Jim Fawcett

Contents

Table of Figures

Executive Summary

"Complex software architectures are extremely hard to manage, not only in terms of the architecture process itself but also in terms of getting buy-in from large numbers of stakeholders. This, in turn, requires a very disciplined approach to identifying common architectural components, and management of the commonalities between federated components — deciding how to integrate, what to integrate, etc."[1]. The pattern/architecture that enforces this approach is called a "Federated Software Architecture".

The advantages that result from a good enterprise architecture bring important business benefits like,

- Lower business operation costs
- More Agile organization
- Improved business productivity
- A more efficient IT operation
  - Lower software development, support, and maintenance costs
  - Increased portability of applications
  - Improved interoperability and easier system and network management
  - Improved ability to address critical enterprise-wide issues like security
  - Easier upgrade and exchange of system components

So, what is Federated Architecture? Federated Architecture is expected to deliver high flexibility and agility among independently cooperating components and at the same time reduce complexity significantly. A federated architecture should be considered for problems with the root cause of unmanageable complexity. This approach is applicable for decoupling and decentralizing projects where a central one-fits-all approach cannot be applied. This approach is especially applicable for long-term migration projects.

When a new code is created for a system, it is built and tested in the context of other code. As soon as all the tests pass, the code is checked-in and becomes a part of the current baseline. There are several services necessary to efficiently support this continuous integration. In this course, we will be developing the concept for and creating, a federation of servers each providing a dedicated service for continuous integration using the federated architecture pattern.

We will be focussing on "Build Server"- an automated tool that builds test libraries. Along with the Build Server, the system consists of other modules with their own set of responsibilities. This structure supports continuous integration which is an excellent practice to follow when developing software. The subsystems of this federated structure are:

- **Repository:** Holds all code and documents for the current baseline along with their dependency relationships. It also holds test results and 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.

1. TOGAF Version 9 Enterprise Edition.

- **Test Harness:** Runs tests concurrently for multiple users based on test requests and libraries sent from the Build Server.
- **Client:** The primary interface into the Federation. Provides GUI to submit code and test requests to the Repository.
- **Collaboration Server:** Provides various management capabilities like remote meetings, shared digital whiteboard, shared calendars etc.

This project is a result of piecewise development of different parts implemented in Projects 1, 2 and 3.

In Project 1, we built a prototype to understand how to build files programmatically. Used MSBuild to build the files. In project 4, process class is used to build the files as process class gives the flexibility to build code files of different programming languages (implemented only for C#). Also, MSBuild needed project configuration file (.csproj) for builds which is eliminated by using process class.

In Project 2, the implementation had most of the functionalities needed for a Build Server (similar to the objective of project 4), but was designed to run only on a local machine. Every part was started as one single process. Even though the project successfully built and tested the code files, it wasn't very flexible (which is explained in the next paragraph).

Project 3 was all about building the foundation for the parts required in project 4. The main objective was to understand how to use Message Passing Communication (MPC) with Windows Communication Foundation (WCF) and to understand and use the concept of process pooling and lastly to provide a Graphical User Interface (GUI). All these provide flexibility like operating all the parts of the federation remotely and on different machines, ability to handle large loads, provide an easy to use interface to increase productivity (which were absent in the implementation of project 2).

Project 4 develops on the prototype created in project 3. It includes MPC between different parts of the federation using WCF, implements process pooling in the Build Server to handle heavy loads when large number of build requests are received, provides a GUI, adds a mock Repository which handles file storage and transfer and provides a test harness which executes the libraries built by the Build Server.

In this document, we discuss the concept, package structure and the organizing principle of the whole federation, some design highlights and concept, activities and users of each of the subsystems in detail and in particular the Build Server as implemented in project 4. It also discusses about some of the critical issues related to flexibility to build files of more than one programming language, scaling (issues when the size of the system grows), ease of use, security etc. which affect the key operations of the federation.

Introduction

This document is an Operational Concept Document (OCD) focussing mainly on the design aspect of the project/tool-Remote Build Server. This tool is a federated architecture that allows interoperability and information sharing between de-centrally organized modules that provides facilities to build projects, run tests, record build/test status and log execution details efficiently. This tool with some enhancements (like versioning or checking-in, checking-out files, querying for files etc.) can be used to make software development and testing easier.

Organizing Principle

Some of the guiding principles behind the project are:

- When a new code is added, or existing code is modified, the whole system must be tested to make this change a part of the baseline. This must be done with minimal effort even for a big system.
- The tool/project should support the above requirement. That is, the tool should support continuous integration and automated testing. This is achieved using modules like Client, Repository, Build Server and Test Harness.
- The modules must communicate effectively among themselves to increase productivity.

Structure and Partitioning

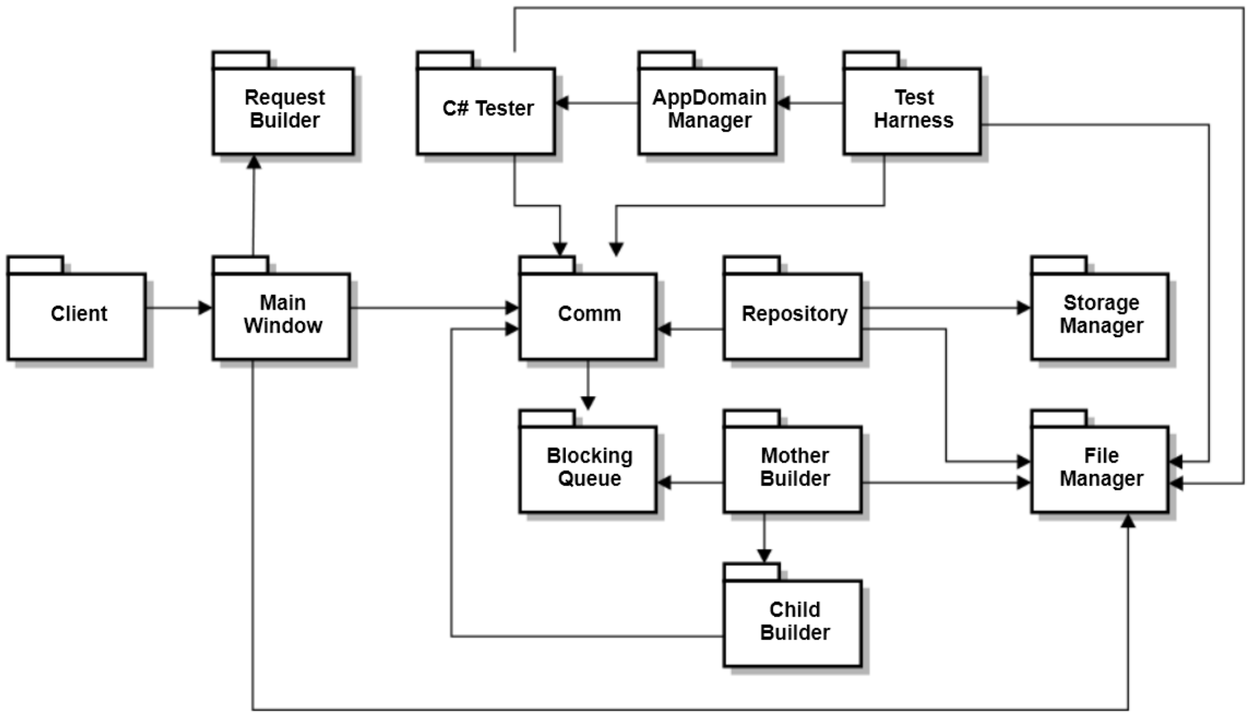

Figure 1: Structure Diagram[2]

[2] Referring the solution provided for mid-term (modified as per my implementation)

- Client: Entry point into the system. Gives option for the user to make selection of the files to be built and tested. Provides option to see the status of the builds and tests. Also, provides option to view the logs of the builds and tests.
- Main Window: Back-end of the GUI. Aggregates Comm package. Has all the processing needed to display files on the GUI. Calls Request Builder package to generate an XML based on the selections made by the Client (on GUI). Has the processing to display all files in the file storage area.
- Request Builder: Generates an XML based on the inputs made on the GUI.
- Comm: Provides communication service, which is used by all the parts in the federation (Window, Repository, Mother Builder, Child Builder, Test Harness). Has Sender and Receiver packages inside it to provide facilities to send and receive messages. Uses Blocking Queue which provides a thread-safe queue.
- Blocking Queue: Provides thread-safe blocking queue which guarantees that the queue is free of race conditions. Hence, supports concurrent message handling without deadlocks.
- Repository: Provides storage for all the files in the federation. The test code files that has to be built, the build request xml's, build and test logs. Aggregates Comm package.
- File Manager: Provides facilities for file transfer between different parts of the federation.
- Storage Manager: Handles request from the Repository for file transfer and storage.
- Mother Builder: Manages the child builders spawned. Receives build requests from the Repository which it transfers to the available child builders. Aggregates Comm package. Uses Blocking Queue to maintain the queues for build requests and ready messages from the child builders.
- Child Builders: Generates the libraries from the files received (from the Repository) based on the parsed build request. Aggregates Comm package.
- Test Harness: Loads and executes the libraries received from the child builders. Aggregates Comm package.
- App Domain Manager: Provides facilities to create new application domains.
- C# Tester: Tests the libraries generated for C# files (option to build C++/ Java files not provided in the child builder, planned for future enhancements).

Overall Application Activities

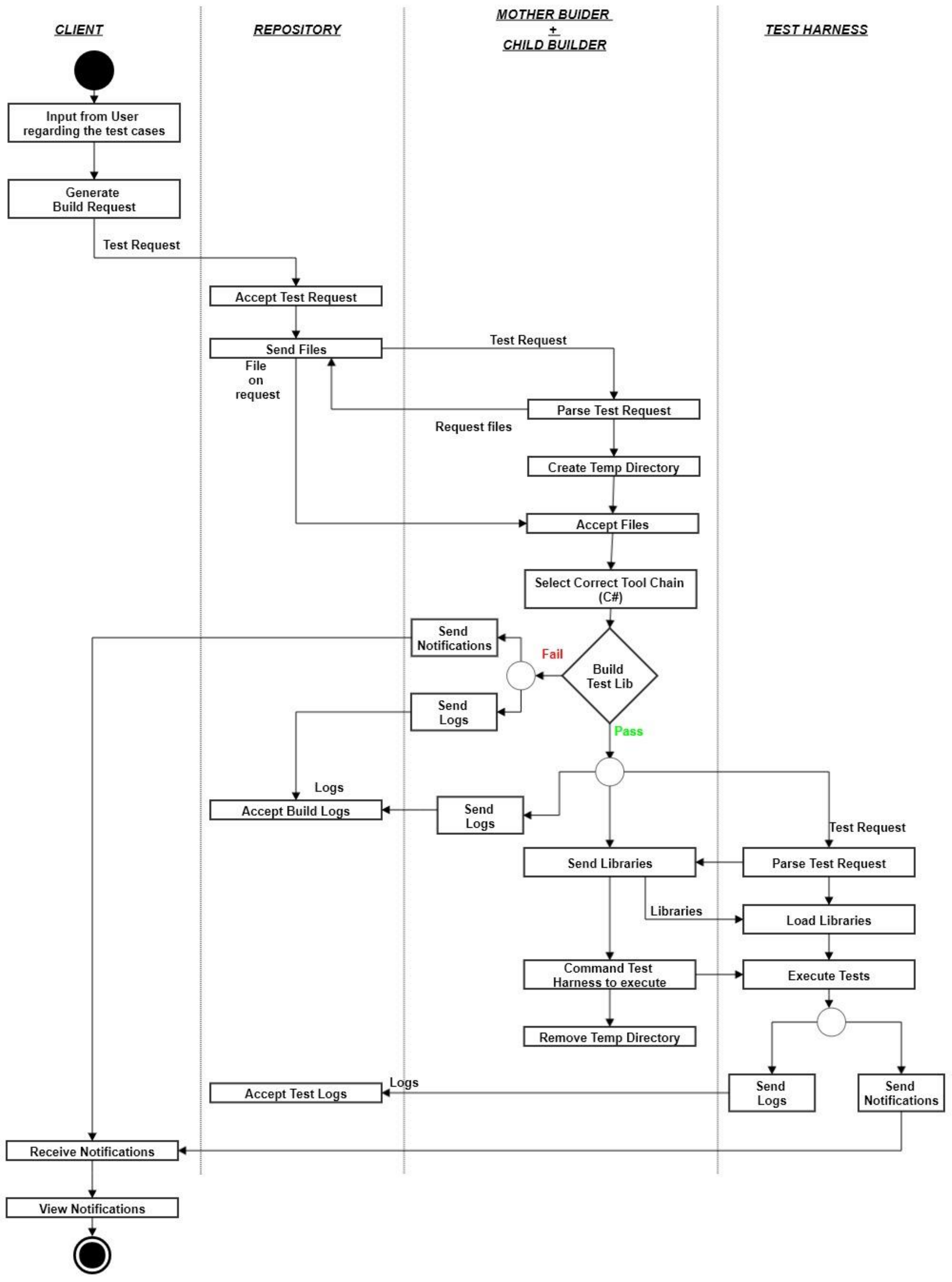

Figure 2: Overall Activity Diagram

Description:

- The user of the tool uses the GUI to select the code files to be tested.
- The Client generates the Build Request based on the selections. And, then sends the Build Request to the Repository.
- Commands the Repository to process the Build Request.
- Then, the Repository takes over. It accepts the Build Request from the Client.
- The Repository then forwards the Build Request to the Mother Builder which then forwards it to the Child Builders (which is free/available).
- The Child Builder which receives the Build Request parses this and requests the Repository for the code files for build.
- The Child Builder creates a temp directory and accepts the code files into this directory.
- By selecting the correct tool chain (only C# implemented), the Child Builder attempts to build the test code files.
- If build fails (due to a missing part or wrongly set properties etc.), the Child Builder sends a notification to the Client of the failure. Also, sends the build log to the Repository.
- If build passes a DLL (library) is generated, the Child Builder then notifies the Test Harness regarding the library. Sends a notification to the Client of the success. And, sends the build logs to the Repository.
- The Test Harness then requests the Child Builder for the library.
- The Child Builder then sends the library to the Test Harness.
- The Test Harness after receiving the library loads it and tries to execute it.
- And then sends a notification of the test status to the Client and sends the test log to the Repository.
- Client will be able to retrieve all the logs and view them too (could not implement in the end due to time constraint).

Design Highlights

Below are some of the important design aspects implemented in the project. The decision to implement these in the project has made the operation of the whole federation more productive, more efficient, highly flexible and highly scalable.

Message Passing Communication (MPC)

Message Passing Communication is an immutable interface where the sending application decides how to configure messages, how to handle notifications, how to interpret the received messages etc. The architecture implemented in the project is similar to HTTP like messaging where the messages are dispatched to the registered communicators between endpoints. This MPC implemented with queues and WCF supports asynchronous operation which is a very efficient way for the servers in a federation to communicate among themselves.

In any system, a client sends a request and the server acknowledges appropriately. In a synchronous system, after sending a request the client must wait for the response, putting on hold the other processing. If the operation to be performed by the server is a long running one, the client has to wait for a long time resulting in lot of CPU cycles spent on waiting. This can be overcome by using Message Passing Communication (with WCF and Queues). MPC supports asynchronous operation where a client can send a message to a server any time and does not wait for a response. After sending a message it can go about sending more messages to different servers or do other processing. And, the server responds to the message when it is available to do so. To do this, MPC establishes a channel between the two processes (using WCF, explained in the next section) to communicate by sending messages.

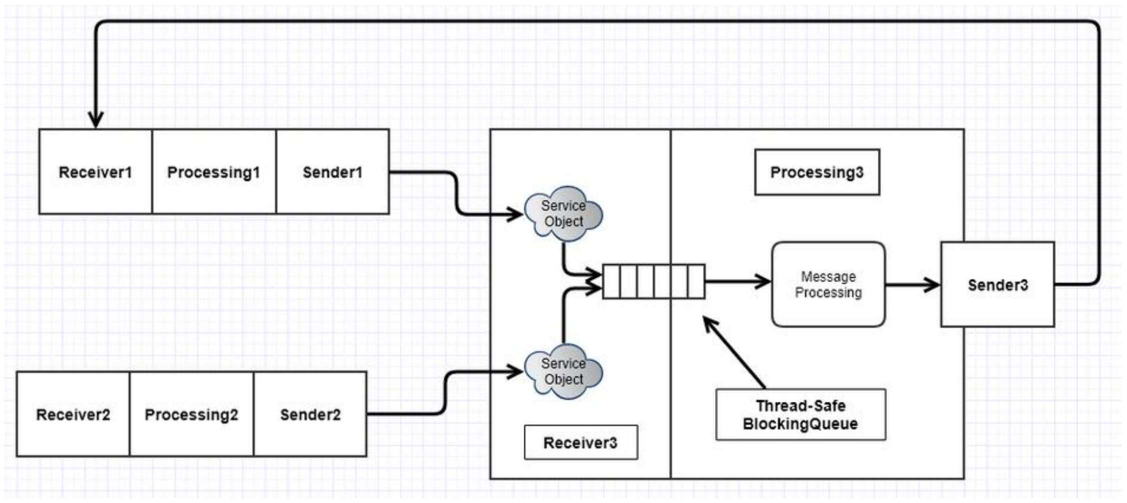

Figure 3: Message Passing Communication with Queues and WCF[3]

Components in MPC:

- **Messages:** A message is a class containing data members which are serializable. A message typically contains a *From* address- if the receiver needs to respond, a *To* address- the

[3] Figure taken from "Message Passing Communication" blog by Dr Jim Fawcett.

endpoint/destination, *type* of message- if it's a request or reply etc., a *command*- what is the operation needed, *arguments*- if additional data is to be provided. More fields can be provided depending on the application.

- **Queues:** Both the client and server implement queues to support MPC. The queues implemented are thread-safe blocking queue. Thread-safe queues guarantees that the queue is free of race conditions and avoids deadlock when accessed by multiple threads. This is achieved using "Monitor" and "locks". Multiple clients can send messages concurrently to the receiver, and the receiver accepts these message into its receive queue. A thread at the receiver's end keeps pulling out messages from the queue (or blocks/waits when there are no messages) and processes them.

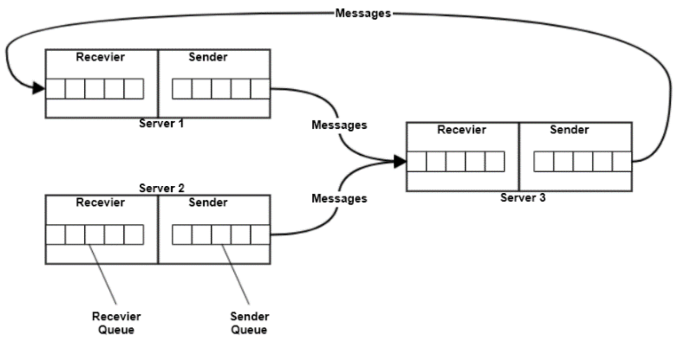

Figure 4: Thread-blocking Queue with MPC

- **WCF communication:** (Will be explained in detail in the next section) MPC uses the facilities provided by the WCF to send messages to the designated endpoints. WCF provides the protocol, channel and the endpoint required for successful transmission of a message. In this project, we use Basic Hyper Text Transfer Protocol (HTTP) to transfer messages. The messages are sent via a channel, which is established based on the endpoint specified in the message. WCF also serializes the message (which is a class with data members which are serializable) and wraps this message in a SOAP wrapper (Simple Object Access Protocol) and transmits this message via the channel.

Below is all the communication that happens between different parts of the federation.

From	To	Purpose
Client	Repository	Send list of code files to display on GUI
		Request for build logs in Repo
		Request for test logs in Repo
		Command the Repo to send Build Requests to the Mother Builder
	MotherBuilder	To kill the spawned child builder processes
Repository	Client	Send code files after request
		Send build logs after request
		Send test logs after request
	MotherBuilder	Send Build Request
	ChildBuilder	Send the code files after request
MotherBuilder	ChildBuilder	Send Build Requests

Figure 5: MPC all communication

From	To	Purpose
ChildBuilder	MotherBuilder	Ready messages when free
	Repository	Request for code files to build
		Send build logs
	TestHarness	Send test request
		Send library
TestHarness	ChildBuilder	Send the library
	Repository	Send test logs
	Client	Send test result notification

Figure 6: MPC all communication (contd.)

Below table provides summary of all the messages used in the project to support the communication shown above.

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

Figure 7: MPC all messages

Windows Communication Foundation (WCF)

Windows Communication Foundation is a framework that provides software services on machines, networks and across the internet. WCF provides a unifying model for all of these. WCF supports sending data from one service endpoint to another.

Some of the important terms in WCF are:

- **Contracts:** A contract is an interface declaration which define the behaviour of the service. Its an interface defining the services rendered.
- **Service:** It’s a construct that exposes one or more endpoints.

- **Endpoint:** Endpoint is a construct where messages are sent or received. It comprises of a location (an address), a communication mechanism (binding) that describes how messages should be sent. i.e., how the endpoint communicates with the world.
- **Channel:** A concrete implementation of a binding element. A binding element just represents the configuration, but, channel is the implementation.

- **Service Operation:** A procedure defined in the service's code that implements a functionality for an operation. These operations are exposed to clients as methods.
- **Service Contract:** Service Contract ties together multiple related operations into a single functional unit. The contract is defined by creating an interface.
- **Operation Contract:** An operation contract defines the parameter and return type of an operation.
- **Message Contract:** Message contract describes the format of the message.

The diagram shown below shows how WCF is used in the project.

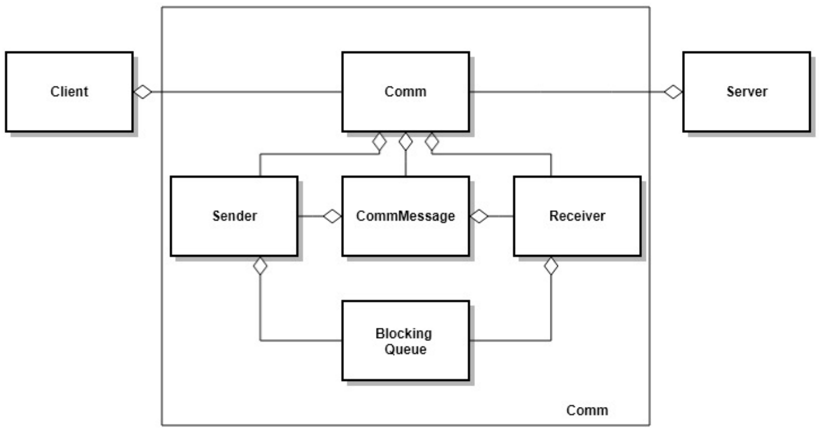

Figure 8: Comm

- The interface provides the service contract and the operation contract needed for WCF
- CommMessage is the class which provides data contract and provides data members which are serializable. These are the messages which gets passed around the parts of the federation.
- Receiver class derives from the service contract interface and provides implementation for the operations. This class provides the service instance.
- Receiver creates a host at the given port and starts listening at that port.
- Sender class uses the operations provided by the service (receiver).
- Sender creates a proxy (channel) with interface of remote instance.
- Blocking Queue provides a thread-safe blocking queue. This is implemented with Monitor and Locks.
- Both, Receiver and Sender uses Blocking Queue to provide thread-safe blocking queue.

- Comm class aggregates both the sender and receiver to provide communication between the different parts of the federation.
- Client and Server (every part in the federation) aggregates an instance of Comm to enable communication between the rest of the system.
- We can also transfer files as blocks using the proxy as the medium of transfer.

Windows Presentation Foundation (WPF)

Windows Presentation Foundation (WPF) provides a rich Graphical User Interface (GUI) for the federation. We use Extensible Application Mark-up Language (XAML) to implement the appearance of the application. WPF provides facilities for buttons, Data display and selection, Dialog boxes, Text boxes, Navigation, Layout etc.

Using a GUI over other alternatives is beneficial as,

- Using a GUI eliminates human errors while checking for dependencies.
- There is no need to know the complex internal functionalities when an operation is to be performed.
- Provides good readability.
- All the necessary files are easily accessible (through proper implementation). That is, every file related/needed to the project will be accessible on a single window.

Snapshots shown below is the GUI provided in the project. Some of the facilities provided are:

- Displays all the files present in the Repository storage area.
- Provides option to select files to generate Build Request XML.
- Provides option to test the generated (and existing) build requests.
- Provides option to view the status of the builds and tests (could not implement due to time constraint)
- Provides option to view the logs generated from these builds and tests (also present in the Repository storage area). (Could not implement due to time constraint)

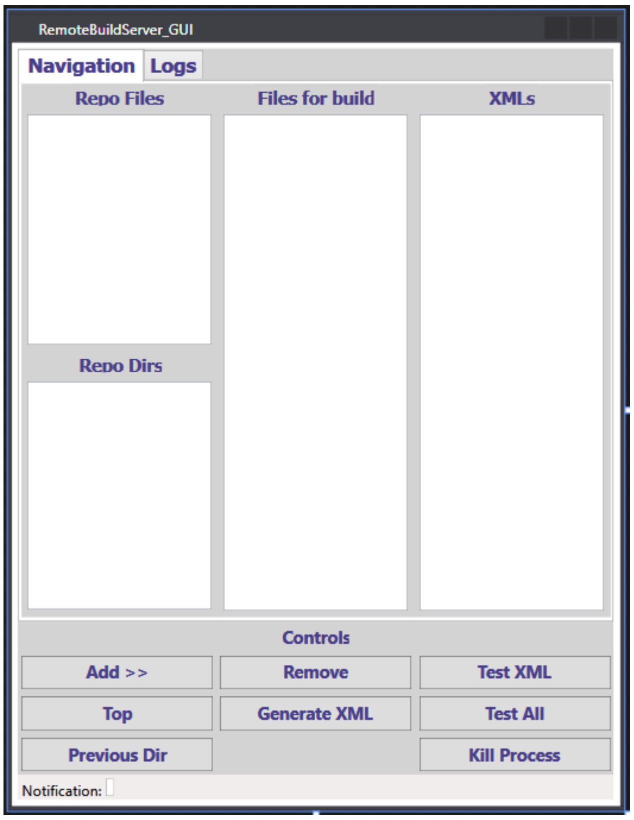

Figure 9: GUI 1

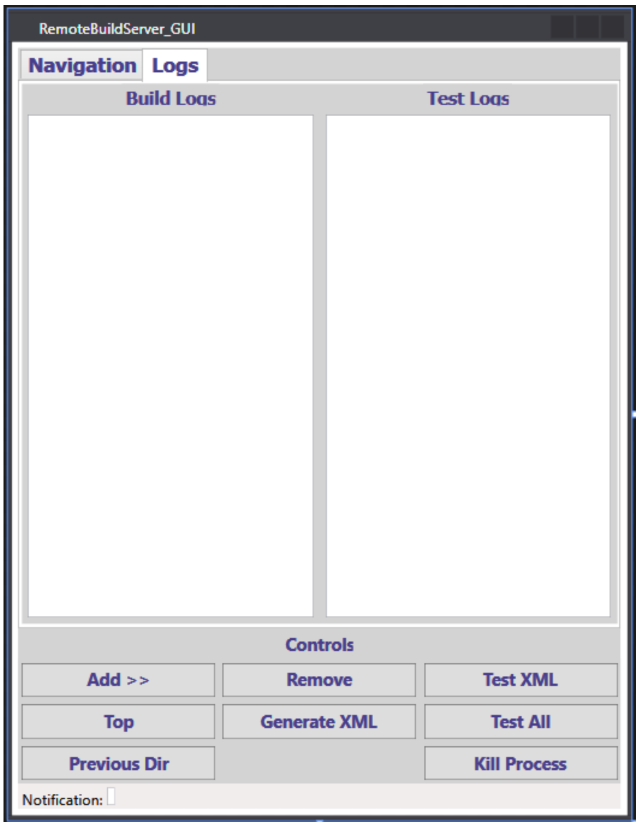

Figure 10: GUI 2

Process Pool

In a large system where many changes are made and checked-in everyday by different developers in a company, there will be lot of testing performed to ensure these changes are integrated well and don’t cause failures. Hence, the builder will be flooded with these build requests which has all the code files where changes have been made and it’s the builders responsibility to build these files and generate libraries. In case of large number of incoming requests, it would be productive to have multiple builders performing the builds so that any particular builder is not overloaded by a flood of requests.

The frequently used technique to achieve this is “Thread-pooling” where new threads are created, and each thread picks up a build request, performs the build, and dies, and this process is repeated. This technique is quick and easily managed, but the main disadvantage with thread-pooling is that when a thread crashes it brings down the whole builder system (the thread-pool which creates new threads for builds) and hence, the whole system stops. Therefore, we use a technique called “Process-pooling” which eliminates the disadvantage of thread-pooling.

Processes might be a little slow to spawn, but the advantages it offers greatly outweighs the disadvantage. Like, these spawned processes don’t expire, they expire only when it crashes (but another process can be easily spawned, like in thread-pool). And, even when it crashes, only that particular process goes down and not the whole system. Hence, it is advantageous to use process-pooling.

Below diagram shows how the process pooling is implemented in the project.

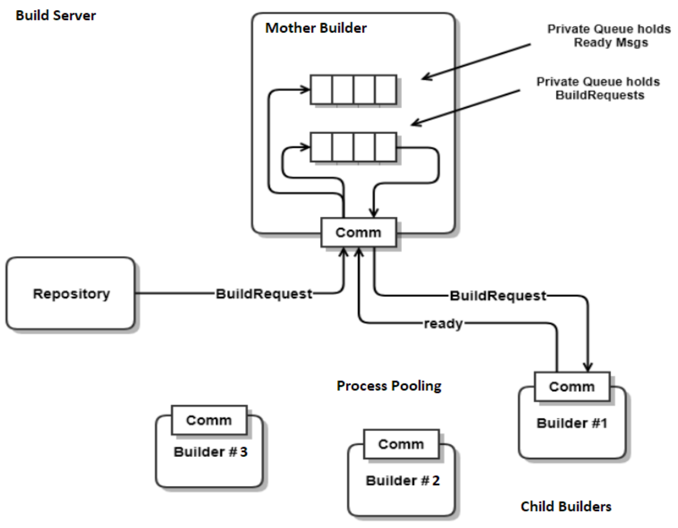

Figure 11: Process Pooling

Below class diagram shows the components needed for process pooling.

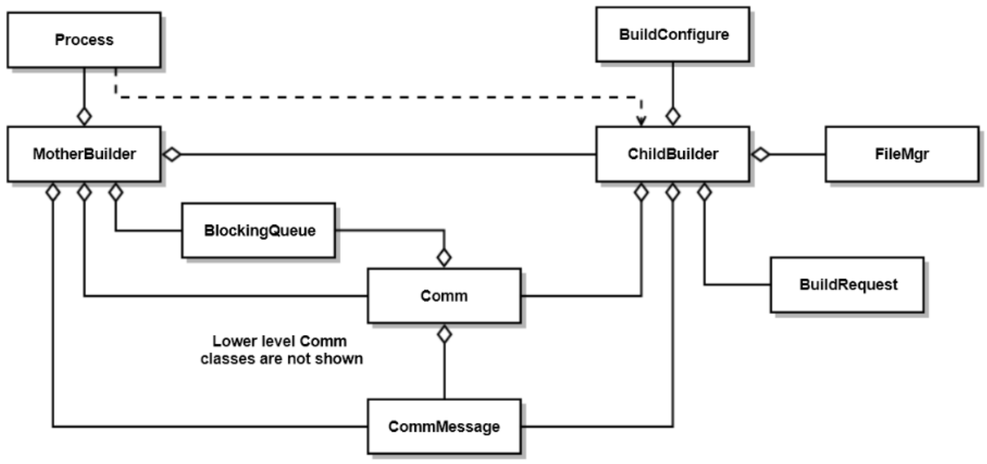

Figure 12: Process pooling class diagram[4]

Here, the Mother Builder is responsible for spawning the process and holding a reference to it. Mother Builder receives Build Requests from the Repository which it then forwards it to the child builder which is available. To manage all this, Mother Builder maintains 2 blocking queues. One for the incoming build requests from the Repository and another for Ready messages sent by the Child processes saying they are ready for building. These queues are synchronized such that if any of the queue is empty, no message is handed over to the Child Builders. This is an elegant system where there is minimal management needed. As the child processes are in a different process, the spawned child processes cannot make function calls and thus communicate with rest of the federation using Comm.

The only case that needs special attention is when any child builder process crashes. As the Mother Builder is holding onto references to the child processes, when a child process crashes, Mother Builder simply removes the reference of that process and creates a new process. Another simple yet elegant solution.

Build Server

Concept

This is the main focus of the project. The main purpose of a Build Server is to build projects based on the code files from the build request. This Build Server along with other systems like Repository, Test Harness provides facilities for Continuous Integration. It enforces the practice of merging all developer working copies to a shared baseline. Continuous Integration supports systems building only from the repository, this way, build packages are reproducible and traceable.

A Build Server:

- Simplifies the developer's workflow and reduce the chance of mistakes and breaking in production (due to some stray DLL's on their individual machines etc.) as it replicates the target environment. That is, it avoids "but it works on my machine" issue.
- Can be easily automated.
- Sets up builds for different versions for different stages of the application (not implemented in project).

Uses and Users

The Build Server uses the build request and the associated code from the repository and builds the test libraries. The Test Harness accepts the libraries from the Build Server to run the tests on them.

QA and Test team use the build logs generated by the Build Server to document the build report.

Structure and Partitioning

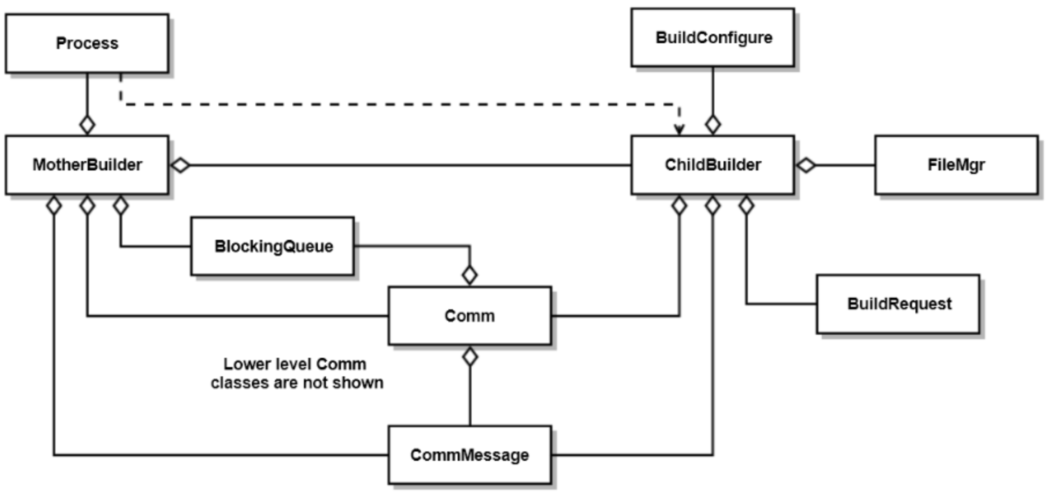

Figure 13: Package Diagram of Build Server

Above diagram shows the partitioning of the classes in the Build Server. Mother Builder spawns the pre-set number of child builder processes, where each child builder is a full-fledged builder which has the capability to build files. Mother Builder and Child Builders uses Comm to communicate between themselves and rest of the federation.

Below diagram shows how the Build Server is structured. That is, how Mother Builder and Child Builder interact with each other.

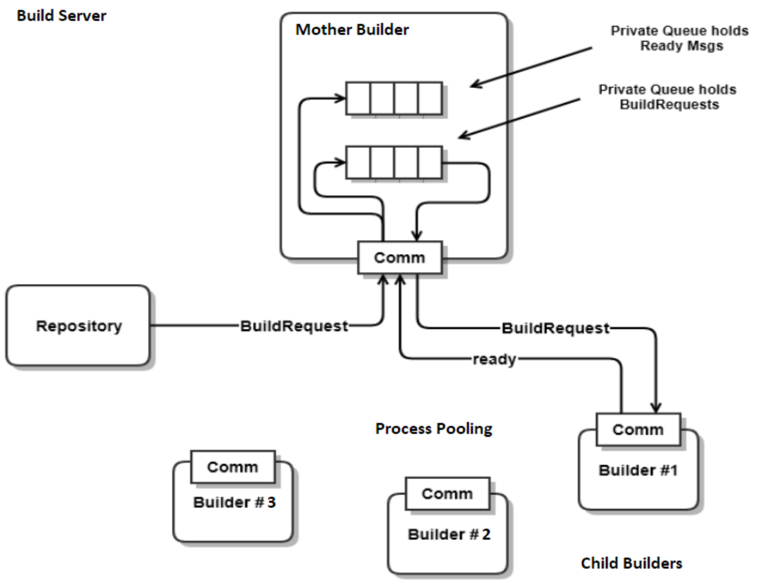

Figure 14: Build Server Structure

Here, all child processes (Builder #1, Builder #2, Builder #3) are different processes but have the same functionality. That is, they receive the build requests from the Mother Builder and then after parsing that requests the Repository to send the appropriate files. These, Child processes also send a message to the Mother Builder with a "ready" status that they are available for accepting build requests. Therefore, Mother Builder maintains two blocking queues, one to hold the incoming build requests from the Repository and another to hold the ready messages from the Child Builders. How this is synchronized is that there are two getMessage() function which pulls out the messages from the two queues mentioned above. Only when there is some message in both the queues, the message will be dispatched to the child builders (analogous to AND gate in Digital Electronics).

The implementation done in project 2 didn't have this process pooling, hence was incapable of handling large loads. Additional advantages this process pooling offers is that if a process crashes, it only brings down that particular process (unlike thread-pooling which brings down the whole build server during crashes). Also, as the Mother Builder maintains a reference to all the processes that it has spawned an event handler recognises the event of the crash and notifies the Mother Builder, the Mother Builder the removes that reference and just spawns a new process.

Activity diagram of the build server is shown in the next page.

Application Activities

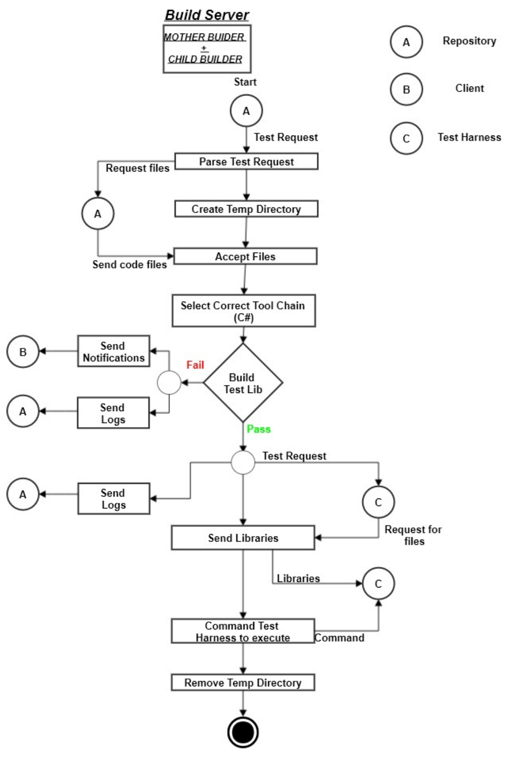

Figure 15: Build Server Activity Diagram

The activity diagram given above shows the activities done by the Build Server to accomplish the task.

Given below is the description of the activities:

- **Accept build request:** Accept build request from the Repository.
- **Parse build request:** Analyse the build request, make a list of files needed for the build and request the Repository for the code files.
- **Create temp directory:** The Build Server caches the code files so that it can look up its cache, the next time it is building code, before requesting to send the required code.
- **Accept files:** Accept the required files from the Repository.
- **Parse build request to select correct tool chain (C#/ C++/ Java):** Parse the test request to select either the C# compiler or C++ compiler or Java compiler (implemented only for C# tool chain).
- **Attempt to build test libraries:** Tries building the test libraries from the received files.
- **Send notice to client:**
    - **If build fails:** If build fails due to a missing part or wrongly set properties etc. sends a message to the client of the failure.
    - **If build succeeds:** send a message to the Test Harness regarding a library available for test.
- **Send libraries:** Receives a request from the Test Harness to send the libraries, acknowledges the request by sending it.
- **Command Test Harness to execute tests:** Send notification to the Test Harness to execute the test.
- **Remove temp directory**

Repository

Concept

The main purpose of a repository is to store a set of files, as well as the history of changes made to those files and to place all software artefacts required to build a project. In this project, we implement a mock repository which holds all code and documents for the current baseline, along with their dependency relationships and also holds the build and test result.

The Repository implemented in Project 4 is a mock repository, hence doesn't implement all the points mentioned above. That is, it differs from a fully functional repository as it does not provide facilities for versioning or checking-in/ checking-out files, querying the repository contents etc.

In a version controlled repository, the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. It is also preferred for changes to be integrated rather than for multiple versions of the software to be maintained simultaneously. The mainline or the baseline should be the place for the working version of the application.

Uses and Users

The main purpose of a repository is to store a set of files. It stores baseline code and test logs. The user can upload and download files (code and test logs) from the repository using the GUI provided by the client. Through the client the repository keeps every user of the tool aware of the latest changes in the repository.

Whenever a developer needs to test/use some code, they need that file and all its associated files to successfully build it. Instead of checking through the references of the file manually, the repository can use the metadata to easily find the dependant files and download them all along with the required file.

Project managers can use the repository to check progress of the project by analysing the files checked into the repository. Also, when a customer reports an issue for an older version, codes and logs for that version can be easily assessed to take corrective actions.

The QA team can access and analyse build and test logs of any version to analyse and maintain the quality of the code.

Test team can easily extract the build and test logs to document the test report.

Application Activities

The purpose of the repository is to store a set of files (codes, logs etc.). And give access for data transfer to different parts of the system. The Repository does the following tasks to facilitate transfer of the required files:

- Accept Build Request from Client.
- Forwards the Build Request to the Mother Builder in the Build Server
- Receives a request from the Child Builder to send the files needed for the build.

- Accepts build logs from the Child Builders after the build
- Accepts test logs from the Test Harness after the test.
- On command
    - Sends build or test logs to the Client for viewing

Client

Concept

The main purpose of a client is to provide access to different servers of the system. That is, the client provides a GUI (Graphical User Interface) to interact with the the Collaboration Server. Another important purpose of the client is to view test logs and build logs. The advantage of using a GUI is that, the selections made in the GUI will be internally translated and given to these subsystems as input, thereby avoids manually keying in the input thereby avoiding human errors.

Uses and Users

The client provides an easy way to access the different servers of the application using a GUI. It initiates the repository to send the test requests and its associated files to the build sever based on the selections made on the GUI. Client can also be used to view the build logs and the test logs generated from the Build Server and Test Harness. Client gives an easy and interactive way to peek into the application and to monitor the health of the application.

Developers can use the client to interact with the repository by checking-in the files for testing, or can check-out the required files to make enhancements (not implemented in the project). They can also use the Client to view the test results of their code files.

QA and test teams can use the client to extract the build logs and test logs to document the test/build reports. They can also use it to monitor and maintain the quality of the code.

Project managers can use the Client to check the progress of a project. They can use old test issues and test results to assess any complaints from the customer. They can also use these files to educate the team of any old issue and its resolution.

Client can be used by the TA’s to check if the requirements of the project have been fulfilled.

Application Activities

The client is responsible for the initiation of the testing and to view the results after the process is complete. After it initiates, other subsystems of the application take over to successfully complete the process. Following are the tasks done by the client:

- Creates build request.
- Sends the build request to the Repository.
- Commands the Repository to process the build request.
- Accept build and test logs from the Repository.

Test Harness

Concept

A Test Harness is a collection of software and test data configured to run a test program unit. Test Harness is software constructed to facilitate Integration Testing and it allows for automation of tests. Test Harness executes test suites of test cases and generates associated test reports. The benefits of having a Test Harness are:

- Increased productivity due to automation of the testing process.
- Repeatability of subsequent test runs.
- Increased quality of software components and application.

A Test Harness has no knowledge of test suites, test cases or test reports. Those things are provided to the Test Harness by the associated framework like Build Server and Repository.

Test Harness runs 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 then sends code and requests to the Build Server which then builds libraries. The test requests and libraries are then sent to the Test Harness. The Test Harness executes tests, logs results, and submits results to the Repository. It also notifies the author of the tests of the results.

Uses and Users

Test Harness uses the libraries received from the Build Server to perform testing. Test Harness provides the facility to perform Integration Testing of an application in an environment for which it has been developed. As the Test Harness tests the complete system even when only a single component is modified, it increases the quality of the system. That is, it allows for thorough testing of the application even for a small modification.

Developers make use of the facilities provided by the Test Harness to avoid rework as the whole process can be automated, thereby, reducing cost and time to the company.

QA team can monitor the quality (standards) of the application by analysing and documenting the test logs provided by the Test Harness.

Project Managers can also use these logs to monitor the progress of the project by knowing which tests are working well and which are failing.

Application Activities

Test Harness is the subsystem which executes the tests. Based on the libraries received from the Build Server, the Test Harness loads and executes the tests. Given below are the steps involved:

- Receives test request from the Build Server after a successful build.
- Test Harness parses the test request and requests the child builder for the library.
- After receiving the library, loads and executes it.

- Sends the test status notification to the Client.
- Sends the test logs to the Repository.

Critical Issues

Issues on a higher/broader level (not specific to an implementation):

1. **Productivity:** For a large system, it is of at most importance that the throughput of the system is high. Developers who check-in the changes made in the code should not have lot of additional manual work to be performed to these changes. **Solution:** The main purpose of this tool is to support continuous integration and automated testing. All the dependant systems of the modified system must be thoroughly tested to ensure no breakage in the system exists. To perform all these tasks, the subsystems provide facilities like a GUI for easy inputting, automatically including the dependant files (from the metadata), programmatically generating the test data etc. that reduce rework for the developers and hence saving time and revenue for the company. Thereby, increasing productivity.

2. **Communication (goes together with productivity):** As the size of the application increases, the number of issues also increase. So, it is extremely important that the appropriate teams are notified of the issues so that corrective actions can be taken. **Solution:** To ensure that all the users of this tool are on the same page, so that, depending on the communicated message a process is initiated that everybody involved in the process agrees on and benefits everyone. So, the communication system between the subsystems must be effective and efficient.

3. **Security:** Security is a major concern for large or small companies. If proper precautions are not taken they will susceptible to intrusions. **Solution:** When the Test Harness runs a malicious library (either from an intruder or inadvertently by a user of another application), the application might get corrupted and the security (or normal operation) of the system may be compromised. Therefore, to prevent malicious DLL's from being executed we need to ensure the Test Harness gets the libraries only from the Build Server which builds files checked-in to the Repository by verified users. Verified users being developers, managers, QA/test team etc. i.e. only people with an approved clearance (from the company or that particular project).

Issues specific to the implementation (Design Issues):

1. **Communication message:** The system can turn out to be complicated if many forms of messages are used. Every part in the federation needs to know all the types of messages that are being used and needs to know how to handle messages of each types, resulting in a complex system. **Solution:** A single message structure is defined, which is the only message that gets passed around. The message has a From and To address, a command argument, list of strings to hold files names and a body to hold logs.
2. **Heavy loads and Complexity of Build Server:** During end of project, many developers will be constantly checking-in changes, this leads to increased testing. This may result in the Build Server being heavily loaded thereby bringing down the productivity. So, to handle large loads we implemented process pooling (explained in the previous sections) at the Build Servers end. This shouldn't result in a system which is difficult to design. **Solution:** We move all the processing (parsing the build request and requesting the files, building the files, generating the

logs etc.) to the Child Builders side and keep the Mother Builder activity to a minimum and by using two blocking queues. Mother Builder just maintains two queues and accepts build requests for the Repository which it just forwards to the child builders. The only (and the main) job is to ensure that the queues are synchronized. That is, a build request will be sent to the Child Builders only if it has a ready (saying it is available) message from the Child Builder. Therefore, there is no processing needed by the Mother Builder to prioritize build requests. And, process pooling comes with its own set of advantages (explained in the preceding sections). Therefore, this is a simple yet elegant solution.

3. **Configuration (not implemented, future enhancement):** The application of the tool is limited if it only builds C# files. The application must be versatile enough to be able to build codes of multiple programming languages. **Solution:** This can be done by implementing .NET environment class and .NET process class. By setting the appropriate environment variables needed for the particular build and by selecting appropriate tool chain commands, code files of different programming languages (ex. C++, Java) can be built.
4. **Managing Endpoint information (not implemented, future enhancement):** An endpoint is needed for any server to send a message. **Solution:** One of the ways of doing it is by storing all the endpoint information in an Xml file and provide a copy of this file to all the servers. When these servers start-up, they load this Xml and gets the endpoint information of all the servers in the federation.
5. **Multiple Clients (not implemented, future enhancement):** When there are multiple users working on the same project, a clash can occur when two or more users try to check-in the same code at the same time. This will result in code being overwritten. **Solution:** Single ownership methodology can be implemented which gives ownership of the file (when checked-in) to a single user and preventing all other users from checking-in the same file.

Conclusion

In conclusion, this tool with some enhancements can be used for Continuous Integration, which is an important part of software development. That is, when new code is created for a system, the tool supports to build it and test it in the context of other code which it calls, and which call it. As soon as all the tests pass, the code can be checked-in and it becomes a part of the current baseline. The Build Server along with other systems like GUI, Repository, Client, Test Harness and Collaboration Server efficiently support this process.

Summarizing the operation of the tool. The user of the tool makes selections on the GUI (Client) depending on the files that needs to be tested. This generates a Build Request and this Build Request is sent to the Repository for storage. But, this Build Request can be further, on command, be sent to the Build Server for building. The Build Server on receiving it, parses and requests the Repository for the code files. Once it receives the code files, the Build Server builds them and sends the log to the Repository. If the build is successful, the build generates a library which is sent to the Test Harness to be tested. The Test Harness then tests the library and sends the test status to the Client (GUI) and the logs to the Repository. The tool also provides the user to view the build and test logs. This is the tool operates. The whole process is automated in such a way that once a user makes selections and clicks a button, depending on the time required for the build, the build and test logs will be available for viewing.

Some design strategies like Message Passing Communication, Windows Communication Foundation (WCF), Windows Presentation Foundation (WPF) and Process Pooling have been implemented in the project for several reasons like remote operation, easy to use interface, ability to handle heavy loads etc. With some enhancements like versioning, querying, ability to build code files of different programming languages etc. the tool becomes more versatile and helps the user use it more effectively.

The whole system is divided into packages which can interact with each other to effectively perform all the tasks as per requirements. This OCD explains the concept, uses, users, the package structure and key activities of each package with relevant diagrams like package diagram and activity diagram. The document also discusses few critical issues and its implications along with the solutions.

References

1. Wikipedia: 1, 2.
2. TOGAF Version 9 Enterprise Edition
3. Stack Overflow: 1, 2, 3.
4. CSE-681 SMA resources: Instructor’s (Dr. Jim Fawcett’s) resources on the website.
5. Message Passing Communication- Blog by Dr. Jim Fawcett
6. Mid-Term review points and solutions (by Dr. Jim Fawcett) for Fall 2017.
7. Project 4 Helper.