CSE Software Modelling and Analysis

Build server operational concept document

Version 1.0

Rohit Kulkarni

9/14/17

Contents

[Executive Summary 1](#_Toc500361121)

[Purpose of the Project: 1](#_Toc500361122)

[Introduction 2](#_Toc500361123)

[Obligations 2](#_Toc500361124)

[Organizing principles: 2](#_Toc500361125)

[Technologies: 2](#_Toc500361126)

[Key Architecture Ideas 2](#_Toc500361127)

[Uses 4](#_Toc500361128)

[Functionality of System 4](#_Toc500361129)

[Users interaction 5](#_Toc500361130)

[Developers and Architects 5](#_Toc500361131)

[Configuration Management 5](#_Toc500361132)

[Quality Assurance and Managers 5](#_Toc500361133)

[Support and Customers 5](#_Toc500361134)

[Teaching Assistants 5](#_Toc500361135)

[Other Software 5](#_Toc500361136)

[Load Analysis and Impact on design. 6](#_Toc500361137)

[Activity Diagram 7](#_Toc500361138)

[Activity diagram explanation 8](#_Toc500361139)

[Design of Build Server 9](#_Toc500361140)

[Package Diagram for Federation Server 9](#_Toc500361141)

[Message Passing Architecture 10](#_Toc500361142)

[Design of the Federation Server Components. 11](#_Toc500361143)

[Mother builder and Child builders Process Pool 12](#_Toc500361144)

[Test Harness 12](#_Toc500361145)

[Important Packages and responsibilities. 12](#_Toc500361146)

[CommunicationObject: 12](#_Toc500361147)

[Design Deficiencies. 13](#_Toc500361148)

[Design Evolution. 14](#_Toc500361149)

[Critical Issues 15](#_Toc500361150)

[Build Server Multi-threaded and Multi-Process Design 15](#_Toc500361151)

[Performance 15](#_Toc500361152)

[Security threat 15](#_Toc500361153)

[Large data transfer 16](#_Toc500361154)

[Large build times 16](#_Toc500361155)

[Cancel long running builds 16](#_Toc500361156)

[Build code of other programming languages? 16](#_Toc500361157)

[References 17](#_Toc500361158)

# Executive Summary

This document is an Operational Concept Document for a Build Server. A build server is a server that is dedicated to compilation of source files into binaries. The build server which we will be targeting in this project will be capable of building C# binaries.

Traditionally, the build process would happen through scripts/ batch files. There are several drawbacks of this method. Firstly, each time a new project is added or an old one is removed from deployment, build script required changes. Dependencies among projects had to be manually managed in the build scripts. Parallel building of independent projects was not possible. To give a big picture, the process was extremely inefficient. This is where build servers come in picture. Build server is a program which solves all the above issues. It automates the build process, works with project and solution files, handles dependencies, gives user the facility to configure pre-build and post build commands and supports parallel build.

This project aims at developing a build server. The development is intended to be done in 4 projects. The first project focuses on creating a high-level architecture and some prototype code for the build server. The prototype code would just demonstrate the building of source files.

The second project then focuses on creating a simple build server, along with mock repository, mock test harness. In the third project, we will be developing a message passing architecture based on the Communication prototype. Along with that a process pool component would be developed. Finally, the 4th project focuses on creating fully functioning build server which would communicate with the mock repository, mock test harness and WPF client through the message passing architecture.

## Purpose of the Project:

The purpose of the project is to build a remote build server. A remote build server system consists of the interaction between different components like a repository, a build server, test harness server. Build server component of the system would be responsible for building C#, C++ projects etc. into dll or static library as per requirement. The repository would be used for storing the different source code files, build logs, test case results, test logs. The test harness component would be used to execute test cases on the built projects. This is the main idea behind the federation server. The architecture of the federation server is discussed in the next section. The servers will communicate using message passing which is a very efficient technique for communication. Security and performance key quality attributes the system should possess. Some of the challenges which will be faced are as follows:

* Scaling the build server for handling huge loads.
* Robustness of the system in case of process failures.
* Support different programming languages.
* Large amount of data transfer (Code bases are huge).

This document starts with a discussion about the architecture of the entire system. Then it focuses on the design of the build server. Moving on it also addresses the issues discussed above and some more and proposes some solutions. In the end, it demonstrates a small prototype to build C# projects programmatically.

# Introduction

## Obligations

The main responsibilities of the application are to help organizations manage big software projects and development. The key features of the federation server can be summarized as follows:

* To manage codebase; store and retrieve file, provide versioning on files, control check-in and check-out policies.
* To build the source code files using language compiler.
* To provide a test suite execution framework to execute test cases.
* A simple client which would allow users to interact with the system.

## Organizing principles:

Some of the organizing principles are as follows:

* All kinds of storage should be handled by the repository server. Diverse types of files, source code projects, test case projects, build logs, test logs should be stored in repository.
* All sorts of communication and data transfer between subsystems will be handled by the message passing framework.
* Every request will be authenticated and authorized before processing. This also includes authenticating each of the software components before exchanging data with each other.
* All the notifications to the client will be delivered through a single service.

## Technologies:

The software will be developed using C#, .NET technologies. WCF will be used for creating the message passing architecture. The use of .net technologies will limit the use of the build server to windows machines. Although mono can be used on Linux machines the analysis of that is out of scope for this project. For development of the build server Visual Studio 2017 will be used as the primary development environment.

## Key Architecture Ideas

The remote build server can be broken down into different subsystems like core build server, the repository, the test harness and a client. These subsystems (components) need to interact with each other to achieve the required functionality. These different interactions can be achieved through a communication technique called as message passing. Message passing establishes communication channels between the interacting subsystems to offer data exchange. The subsystems will send messages to each other as per requirement and get the result back. The message passing infrastructure can be achieved using WCF (Windows Communication Foundation).

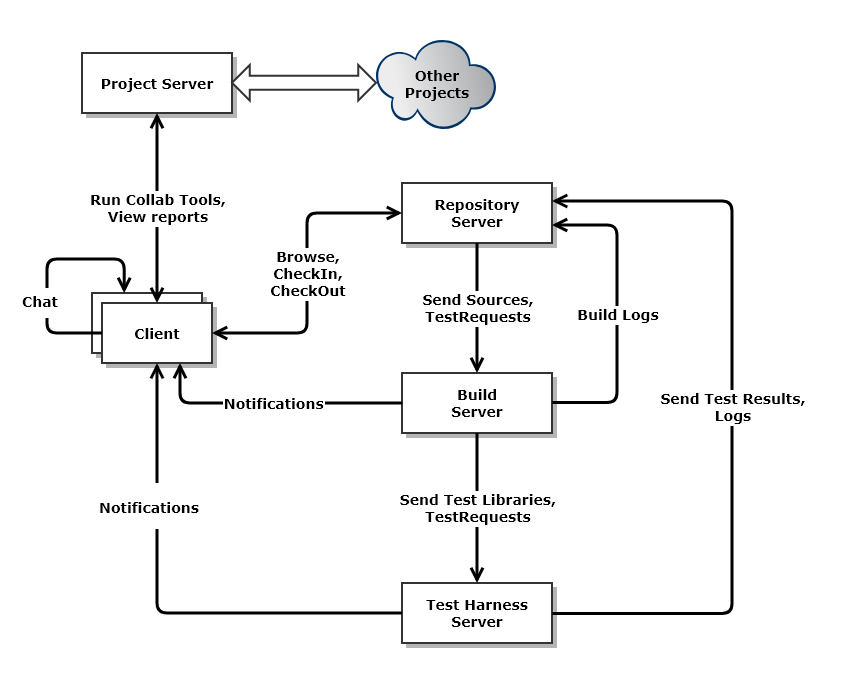


Fig 1: Architecture Diagram.

# Uses

This section describes the high-level functionality of the entire system, explores the different category of users and their interaction with the system, along with the nature of interactions. It also addresses some critical design issues in brief.

## Functionality of System

1. The main function of the build server is to perform building of source code files to binaries. The build server implemented in project focuses mainly on building of C-Sharp code. This code can be compiled into different form: executable, dynamic link library, static library, native code depending on the project type. The build server provides user the option to specify different designed to support different build configurations (debug/ release), target platforms and specify compiler flags. The build server supports multiple build requests at the same time.
2. The build server supports the creation of child builder processes. Server has a process pool component. This component creates child builder processes as per request. The child builder process is the one that handles the compilation request.
3. The build server interacts with the other components of the system like Mock Repository, Mock Test-Harness and the Client. The source code files are requested to the repository. The build logs are saved in the repository in BuildLogs folder. The built libraries and executables are sent to the test harness for execution of test cases. Along with that, the test request file will also be sent to the Test-Harness.
4. The Mock Repository will be responsible primarily for managing the code base. For the scope of this project, repository is acting as a mock repository. Hence it just serves as a file server. Repository will store build requests, build logs, test logs. It will forward the build requests to the build server (Mother Builder Process). The client will request the repository for directory contents. The repository will send a list of files and folders to the client. Client will also save build requests. The build requests will be saved under BuildRequests folder in repository. Repository will accept build logs from the build server and store them in BuildLogs folder. Similarly, it will also accept test logs from the TestHarness.
5. The Mock Test-Harness is a framework to execute test cases in a test suite. It will execute the test cases and log the result (pass or fail) of executing the test case back to the repository.
6. Client is the interface through which the end users can access the entire system. The client will allow the end user to
   1. Create build request, add files and test drivers and save them to repository.
   2. Trigger a build.
   3. Browse repository contents.

Users interaction

The users of the system can be broadly divided into the two categories: End users and other software. This section groups the users somewhat based on similarities of nature of interaction.

Developers and Architects

Developers and architects would interact with the repository server heavily to check-in and check-out files. These operations would be performed heavily on daily basis. Following are the tasks:

* Check-in and check-out files, commit changes.
* Create, merge, rollback change-sets.
* Download latest builds (dependencies) from the repository.
* Upload design and analysis documents.
* Browse the repository; create, update, delete folders on servers.
* Create and upload unit tests.

The repository server needs to be multithreaded, capable of handling several client connections concurrently. Multiple users will be accessing different files or even the same file at any time.

Configuration Management

The configuration managers can be identified as those who manage the daily builds, versioning and deployment of software etc. They would interact with the system for managing software versioning, triggering daily builds, creating code branches, creating deployment packets, managing hotfixes. These users would heavily rely on the build server to create daily deployment. If automatic building is enabled, every check in by a developer would trigger a pre-configured build automatically. If same configuration build is triggered multiple times then there should be a mechanism to queue the builds and execute them sequentially one after the other in the order of the check-ins. Also, the build server should be capable of building independent projects parallelly within a configuration.

### Quality Assurance and Managers

These users will interact with the system to check the status of project. They would mainly use the project server and related tools to keep track of project. Monitoring code quality metrics, tracking project deadlines, tracking incidents and their progress, checking build logs would be some of the operations these users would interact with the system.

### Support and Customers

The support staff and customers would interact with the system for tracking of incidents, daily deployment builds, hotfixes, code base access etc. These users would be interacting heavily daily.

### Teaching Assistants

The teaching assistants will be interacting with the project through the client for grading the project. They would also assess the stability of project by performing ad-hoc testing. The nature of interaction is occasionally.

### Other Software

Other software might also need to communicate with the system. Some of the use cases are as follows accessing code base to run tools like dependency analyzer, code-complexity calculation.

# Load Analysis and Impact on design.

Load analysis really depends on the way the system is used. Some of the different ways system can be used are as follows:

1. One organization may have 500 developers making daily check-ins and check-outs for 5 days a week. They can configure to trigger builds overnight or only at request from management. The main load will be on the repository server while the build server will be only used at certain times of day.
2. Another organization might have over 2000 developers interacting daily with the collaboration server. The organization might configure the server to build the changes and execute test cases before every check-in. This will create substantial build requests and load on repository. This will be an extreme case. One way to solve this issue is to host 2 or even 3 build servers (Scale out).
3. Another case is where the size of the projects to build is huge. The organization might be small having only around 100 developers but a legacy application with over 3000 projects. In this case, parallel building of independent projects will help improve performance.

The above use cases help us draw the following conclusions:

1. Build server should be designed to support multiple instances.
2. Build server should be designed to support building of independent projects parallelly.

Load testing would help figure out how the system will react under different scenarios mentioned above. It will also give an idea of the utilization of hardware resources in each scenario.

## Activity Diagram

The activity diagram for the core build server functionality is shown below.

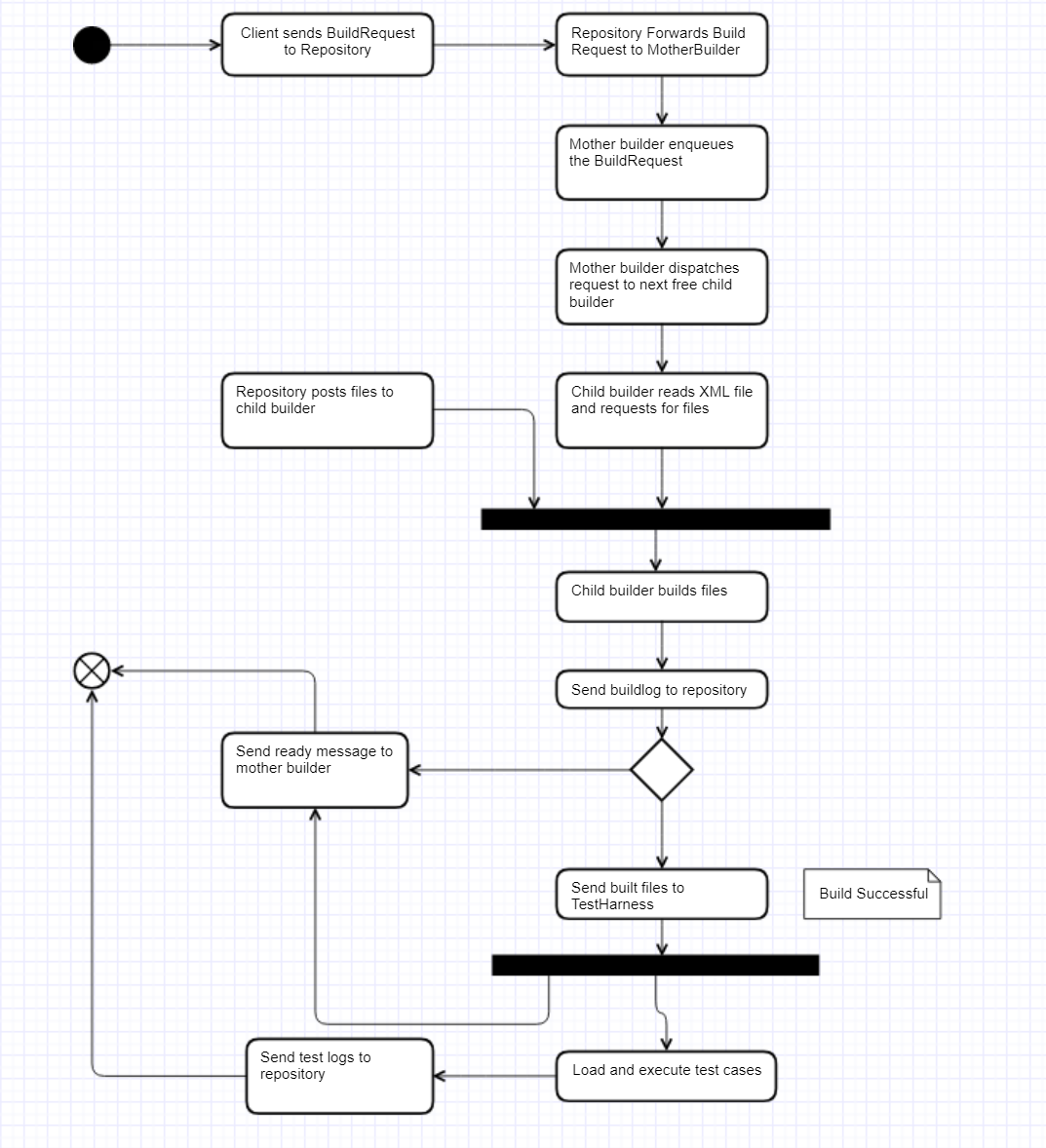


Fig 3: Build Server Activity Diagram

## Activity diagram explanation

1. The WPF client browses the repository and triggers a BuildRequest XML file. It dispatches a message to the repository with the filename.
2. The repository then streams the XML file to the mother builder storage. After streaming the file it sends a build request message with the filename which it sent.
3. The mother builder enqueues the build request message. Mother builder has a dispatcher thread which dispatches the message to child builder. The dispatcher waits for any child builder to be ready. As soon as a child builder is ready, it streams the file to the child builder and sends a message to it.
4. The child builder reads the XML file. After reading the XML file, it requests the repository for the source files. It creates a new directory for the current request. It then waits for the repository to stream the files to it.
5. On receiving all the files, the child builder proceeds with the build. It notifies the client and creates a process to start the compiler exe. After process exits, it checks if the build succeeded or failed.
6. Child builder notifies the client of the build result. It then writes the build log file and sends it to the repository for storage.
7. If build succeeded, then it forwards the built dll/ exe to the test harness for executing test cases. It then sends a ready message to the mother builder signaling that it is ready for the next build request.
8. The test harness loads the executables into a different app domain. And then it searches for all the methods matching the signature: “**public** **bool Test()”** and executes them. It records the output and creates a log file. It notifies the client of test case execution end and sends the log file to the repository.

# Design of Build Server

This section of the document throws light on the design of the build server. It covers the functionality and description of most of the key packages. discusses different approaches in building packages along with some design issues.

## Package Diagram for Federation Server

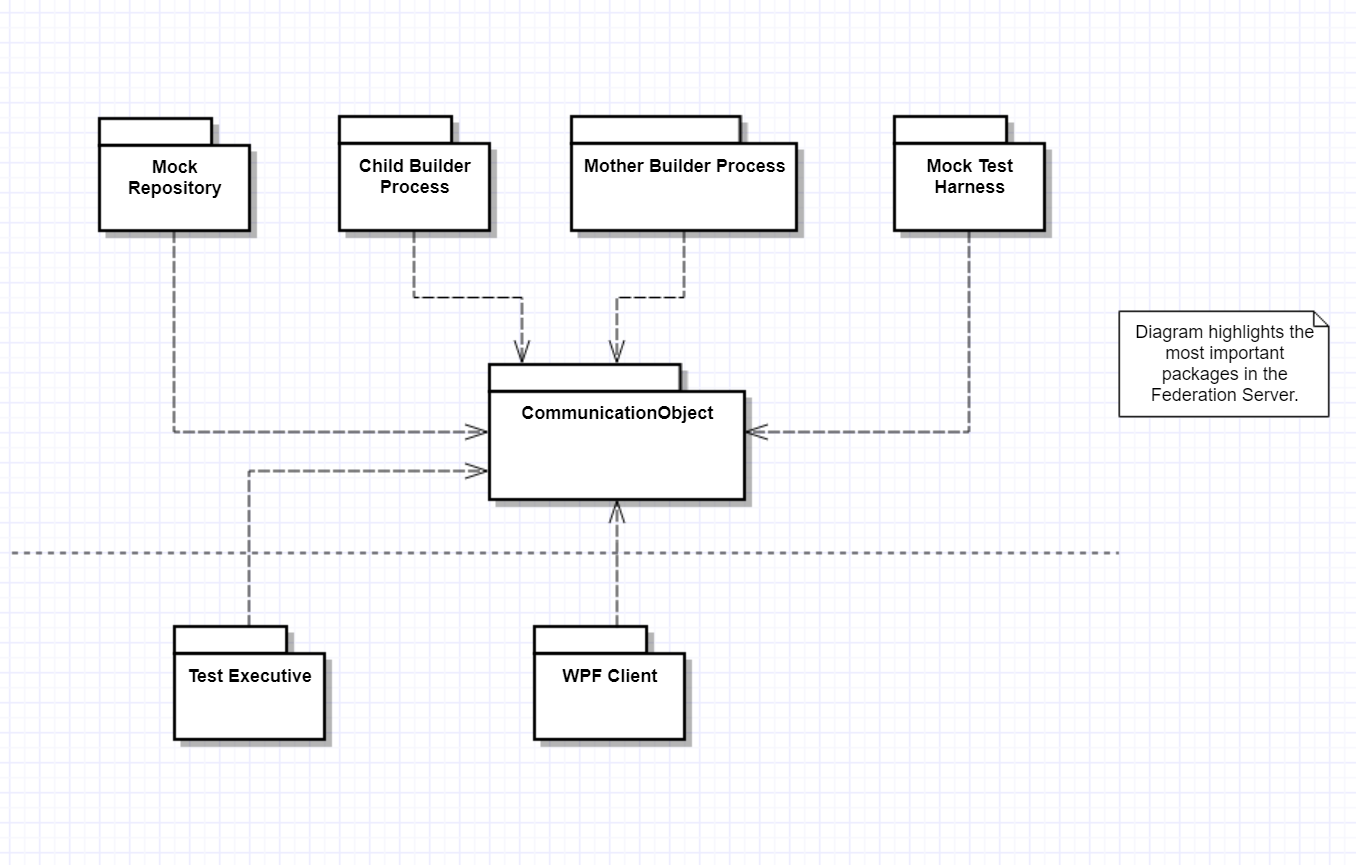
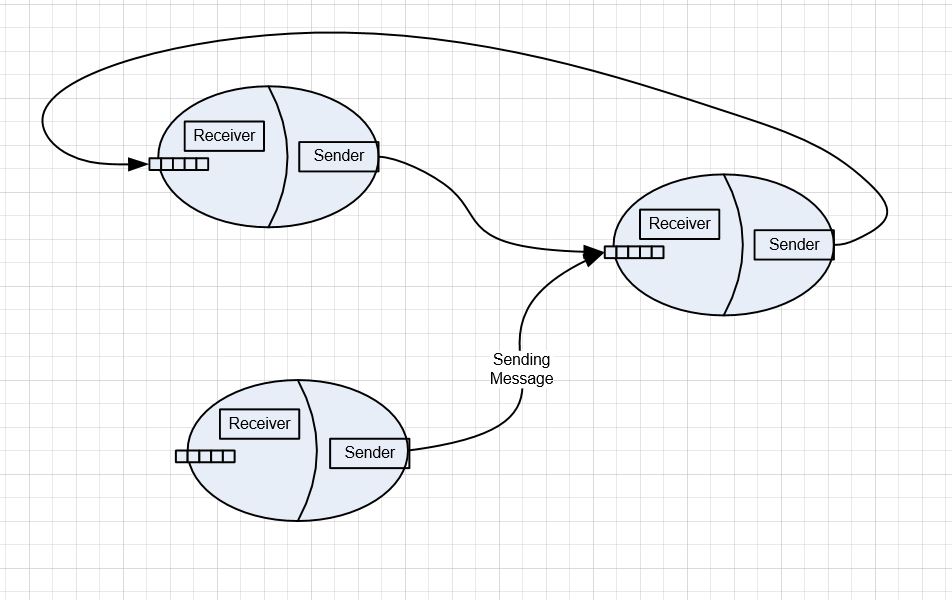


Fig 2: Build server package diagram

The above diagram shows that the **CommunicationObject** package is at the soul of the system. All the federation server components are built on top of the communication object. The communication object provides a uniform interface for passing messages between the different components. The message passing architecture is explained below.

## Message Passing Architecture



**Fig 3: Message Passing architecture**

Message passing architecture is an asynchronous communication architecture developed on Windows Communication Foundation (WCF). This architecture was designed by Prof. Jim Fawcett. The entire federation system is based off this communication mechanism.

The above diagram has 3 communication objects which are interacting with each other. This architecture is designed to be asynchronous. The idea is to send a message and continue with work. Once the message is served i.e. an expected reply message is received take that message up process it.

Each communication object contains sender and a receiver. Both the sender and receiver contain a message queue. The message queue is a blocking queue i.e. it blocks the thread trying to dequeue the messages from it if its empty. Sender and receiver have a thread which dequeues the messages to process them.

The receiver publishes and interface which allows the sender to connect to. The sender of one communication object establishes a channel with the receiver of the other communication object. The receiver interface publishes methods to PostMessage and PostFile. Once this channel is established, the sender can send messages and files to the receiver. The communication happens over HTTP and WCF handles it very well.

**For this project, CommunicationObject has been slightly modified.**

The MessageReceiver class uses InstanceContextMode=PerSession. This gives us the flexibility to maintain state between successive calls made through the same channel. So, calls to **OpenFileForWrite, WriteBlock, CloseFile** can be made to the same object. This helps in managing the directories for files smooth.

The PostFile method has been slightly modified for robustness. Simulating build requests at a very fast rate would throw exception. The connection channel would be changed by a different thread. Hence to overcome this, a function level factory object is created and used to create the channel. The channel object has been created using the **using** syntax so that it would be disposed immediately.

## Design of the Federation Server Components.

As we have seen that each of the federation server components use the message passing architecture. Hence, every component of the federation server inherits from a base class **CommunicationBase** which provides these communication capabilities. The **CommunicationBase** composes the CommunicationObject type and exposes methods. This allows us to segregate the responsibilities in a very efficient way. All the communication related code goes in the base class, while all the code specific to federation server component goes in the derived class. Base class also handles common messages like connect, close sender and close receiver.

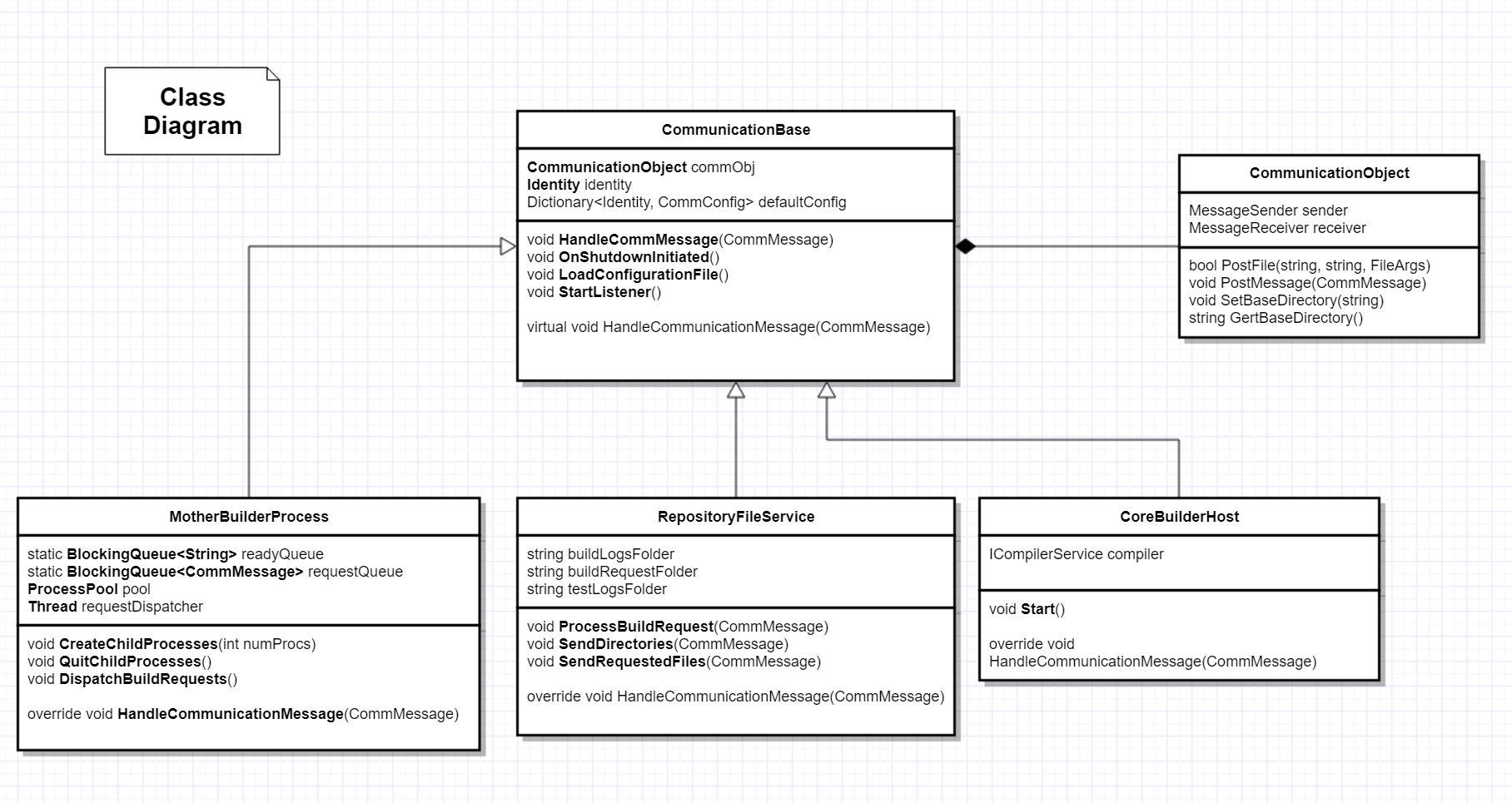


Fig 4: Class Diagram

## Mother builder and Child builders Process Pool

The build server is divided into 3 main components:

* MotherBuilderProcess
* ChildBuilderProcess
* ProcessPool

ChildBuilderProcess component is the one which is responsible for performing the builds. MotherBuilderProcess component is responsible for delegating the build requests to the ChildBuilderProcesses. ProcessPool is responsible for creating child builders. A combination of all these 3 helps us to achieve high scalability of the build server. Depending on the load, the mother builder can tell the process pool to create new child builder processes or quit existing builder processes.

## Test Harness

The test harness is a program that executes test cases. It does so by loading a DLL or EXE into the process, searching for types with Test method and executes them. The problem with this approach is that once a DLL is loaded into a process it can’t be unloaded till the application exits. Microsoft .NET framework provides a solution for this. A process can be compartmentalized into AppDomains. AppDomains can be loaded and unloaded. And assemblies can be loaded into app domains. So, the idea is to create a new app domain, load assemblies into it and then unload the app domain. In this project, the app domain approach has been implemented.

Although there is one drawback with this approach. Any unhandled exception in the ChildAppDomain will crash not only the app domain but the entire process i.e. our TestHarness. The ideal approach is to create a process for handling test requests using a design similar to the MotherBuilder and ChildBuilder.

## Important Packages and responsibilities.

### CommunicationObject:

The CommunicationBase package is the heart of the build server. This package is responsible for sending and receiving messages. It has a MessageSender and MessageReceiver which is used for communication.

## Design Deficiencies.

1. Implementation of CommMessage as a value object class.
   * This is something that would minimize the probability of introducing bugs. At various places in code, the messages are simply forwarded to different addresses, by changing a few properties. Sometimes we tend to reuse the same messages by changing properties.

Eg:

CommMessage msg1 = new CommMessage() …;

Var msg2 = msg1;

Msg2.To = “repo”;

This situation can cause unwanted consequences.

1. Better way of intercepting OpenFileForWrite.
   * Currently, I have created an event for intercepting the OpenFileForWrite, to change the base directory so that the file is created in the right directory. This allows for better control over the writing of files and avoids unnecessary moving.
2. TestHarness process level isolation.
   * Currently, test harness has app domain level isolation which is risky for the server. This needs to be changed to process level isolation by adopting a design similar to the Mother builder and process pool.
3. Multiple Source languages.
   * Currently, building multiple source was not supported. The design supports it. But there wasn’t enough time to complete implementation of Java compiler.

## Design Evolution.

1. Design evolution was more of a paradigm shift from Synchronous to Asynchronous processing. The main component that evolved the most was the core builder. Initially in project #2, the Core builder was completely synchronous. It received the build request, fetched files using a pull mechanism, performed the build, sent the generated dll to test harness. But after implementing project #3, all the tasks had to be broken down as different responsibilities the core builder performed. Each function became capable of performing its task independent of other function. The chain of function calls had been broken and all those were now independent calls at different time intervals. The only thing that tied all these together was the buildrequest object.
2. In project #1, the compilation was done by using the csc.exe from the .NET framework installed on the local machine. In project #2 however, I decided to get rid of this environmental dependency and instead decided to ship the compiler and its dependencies as a part of deployment. This has following advantages:
   1. No dependency on the environment.
   2. Uniform behavior i.e. all the processes will use the same compiler executable.
3. In project #3, there were a lot of message exchanges between the mother and child processes requesting the address of repository and the service URL of the different federation servers. In project #4 I have created a single configuration file with all the URL and ports for the different components of the federation server. The comm object loads reads this file and hosts the receiver. And each component knows every other component of the federation server except for the client.
4. Reflecting on the modularized design of the CoreBuilder as mentioned in point #1, a same approach has been used in the test harness. Test harness uses a blocking queue and has 2 responsibilities. It essentially does only 2 things
   1. prepares a session for incoming request from client.
   2. Executes test cases.

Using a blocking queue and approach used in point #1, one test request sliced into 2 responsibilities tied together using a sessionId. These 2 responsibilities can execute independent of one another. The Core builder could be redesigned in a similar way. That would facilitate the building of 2 requests simultaneously without having to be idle while it is receiving files from repository.

# Critical Issues

This section throws some light on some of the critical issues which need to be addressed. It also discusses some solutions to solve those issues and the trade-offs on design the system.

## Build Server Multi-threaded and Multi-Process Design

1. **Issue:** How to build a multithreaded design correctly? The build server will be a remoting object which will listen for connections. Once a connection is authenticated it will create a request handler to process the connection. This request handler will be a separate thread which will handle the request. So, for every connection request a new thread will be created. Now as explored above there are several ways of managing the building of the projects. One of them is the creation of a process to launch MSBuild.exe to build solution. Creation of child process to launch exe and then sharing of resources like file service and notification service is an issue.
2. **Resolution:** One way to solve this issue is to use serialization of objects and using inter-process communication to transfer it to second process for use. Another way to solve this is to make the thread that spawned the process to wait until the process exits and then continue execution. This will get rid of all inter-process communication.
3. **Impact on design:** Serialization of objects is costly and performance will be traded off. Both approaches will be considered while implementation.

## Performance

1. **Issue:** If multiple build requests are issued simultaneously, it can create performance issues. This is because substantial amount of data might be transferred from the repository to the build server. And many build requests will lead to creation of considerable number of threads and even some processes. This will slow down the build server.
2. **Resolution:** This can be resolved controlling the number of simultaneous builds in progress. For doing that a blocking queue can be used to enqueue build requests after authentication. Another way of resolving the issue is to host another build server instance and use a load balancer to distribute the load across both the instances.
3. **Impact on design:** From a scaling perspective, the application will scale well and manage the load much efficiently.

## Security threat

1. **Issue:** A lot of data is being transferred in between the servers through the communication channel. Security is a general threat to the entire collaboration server. Any vulnerability may result in the loss of sensitive information. One way to solve the security issue is to encrypt and decrypt the files during data transfer before transfer. But doing so will create a performance overhead.
2. **Resolution:** Authorization and authentication module will make sure that no unauthorized user gets access to the data and program. Firewall and other security mechanisms can be used to prevent cyber-attacks. Critical files can be encrypted and decrypted before transferring over the wire.

## Large data transfer

1. **Issue:** Code projects can be extremely large. Not only that, Thus, data transferred over the wire will be huge. This might slow down the entire network.
2. **Resolution:**  Data can be compressed using compression techniques before sending it over network. This will reduce the transferred bytes significantly. Another way of resolving this issue is to use gigabit lines.

## Large build times

1. **Issue:** Code projects can take a large time to build. If a system has around 2000 projects to build, it will take a long time to complete it. Multiple such builds can overload the server and delay builds easily.
2. **Resolution:** Parallel building of independent projects can help solve this issue. A dependency analysis of projects can be done to find out the independent projects and dependent projects. The independent projects can be built separately and parallelly along with inter-dependent projects.

## Cancel long running builds

1. **Issue:** How to cancel long running builds if user requests? This is a very severe issue. This issue needs to be addressed for both the approaches. Consider the build server is using MSBuild.exe in a process. In this case a graceful way of exiting the process is required such that the entire application doesn’t crash. Considering the build server is executing the build method on a thread. There is no way of intercepting a method call without using a checking mechanism which monitors a Boolean variable at each step.
2. **Resolution:** Executing the build method as a task and using a CancellationToken to stop it can be used in this case. CancellationToken is can be used to cancel the task.
3. **Impact on Design:** This makes the design more complicated to implement.

## Build code of other programming languages?

1. **Issue:** How to support other programming languages in future? The current system only considers the building of C# projects. If requirement comes in to support building C++ or python projects how to achieve this.
2. **Impact on Design:** Build engine should be designed with emphasis on extensibility right from beginning.

# References

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/command-line-building-with-csc-exe>

<https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/StudyGuideOCD.htm>

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-cancellation>

<https://stackoverflow.com/questions/34138235/how-to-use-roslyn-c-sharp-compiler-with-visual-studio-2015>

<https://ecs.syr.edu/faculty/fawcett/handouts/webpages/BlogMessagePassingComm.htm>