SOFTWARE COLLABORATION FEDERATION

DR. JIM FAWCETT

DECEMBER 7, 2016

RAHUL VIJAYDEV
SUID:901319342

CONTENTS

1. SOFTWARE COLLABORATION FEDERATION

1.1 EXECUTIVE SUMMARY:

Software Collaboration Federation (SCF) is a collection of clients, servers and associated software designed to support activities of a software development team. It is intended to support development teams comprised of hundreds of developers divided into groups at different locations spread across the world. The activities of the software collaboration federation are:

- The Software Collaboration Federation orchestrates the creation of plans for software development. This comprises of writing operational concept documents, creating and editing work packages, scheduling work packages, and allocating resources to work packages.
- Writing and publishing specification and design documents.
- Developing new source code packages.
- Acquiring already developed source code packages for reuse and integration.
- Continuously building execution images and loadable libraries for testing.
- Executing tests: unit tests, integration tests, regression tests, performance tests, stress tests, and acceptance tests.
- Deploying software executables and documentation.
- Documenting progress reports, notifications and key events.

The Software collaboration federation is composed of the following systems,

- Collaboration Server and Virtual Display System
- Repository Server
- Build Server
- Test Harness Server
- Client Systems

Out of all the aforementioned systems, the Test Harness is the busiest system on the federation. The test harness, in conjunction with the build server and repository is responsible for continuous test and integration of code (CTAI). The principle of continuous test and integration is that code developed by various developers on the software project is continuously built, tested and integrated into the software baseline. With each test and integration instance, the size of the baseline increases until a fully functioning software product has been tested and assembled for production.

1.2 GOALS OF SOFTWARE COLLABORATION FEDERATION:

The goals of the Software Collaboration Federation are as follows:

- The Software Collaboration Foundation must support continuous build, integration and testing of code.
- To ease software development process.
- To maintain consistency in code developed by members scattered across different geographical locations.
- To provide a sophisticated authorization model to make sure only those involved in the project have access to project related information.
- Must contain design documents, operational documents, project specifications, source files and all other information necessary for project development.
- Ensure reusability of existing code.
- Must provide communication links between different systems and also among different project members.
- Help shape the project into a quality end product.

1.3 ARCHITECTURE:

The architecture of the software collaboration federation is depicted in the figure below,

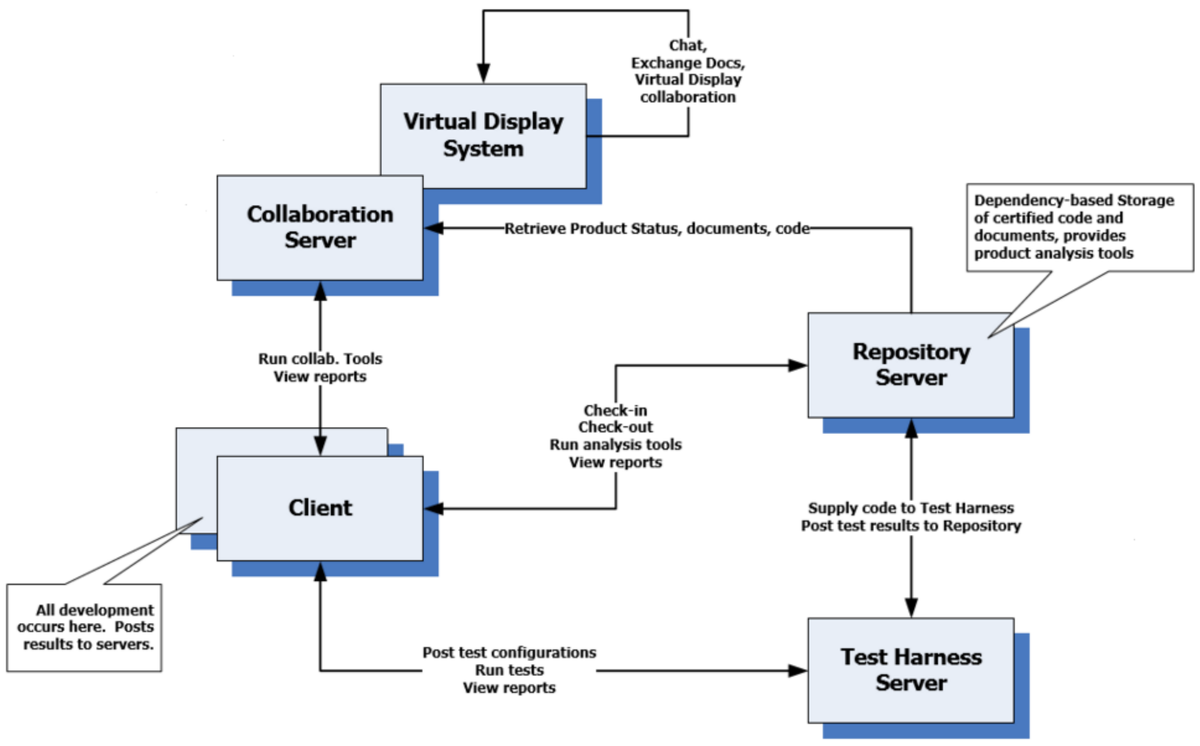


FIG. Software Collaboration Federation

Each system in the software collaboration federation plays an integral part in the development process. Thus, it can be fair to say that each system is indispensable in its own right. The Software collaboration federation is composed of collaboration servers and virtual display systems, build servers, a test harness server, repository servers and thousands of client systems.

1.4 COLLABORATION SERVER AND VIRTUAL DISPLAY SYSTEM:

Introduction:

Software development is a long and convoluted process that involves hundreds of developers working in multiple groups. Tracking work progress becomes too tedious in such large-scale projects. This problem can be solved by equipping collaboration servers that maintain work packages and help track each of the work packages. The collaboration server provides exploratory tools and services based on metadata information, beneficial for project managers and team leaders. An example is a tool that generates work package and pending work package information using which team leaders and developers can track progress.

The collaboration server is also used for scheduling virtual meetings and storing details pertaining to those meetings. Design and implementation details can be stored on the server and made accessible to developers engaged in a session. Through the collaboration server and Virtual Display System(VDS), it is possible to have full-fledged discussions over the network.

The virtual display system(VDS) provides a graphical display or user interface through which test details, work package details, reports and design diagrams can be manifested. The virtual display system also ensures provision for messaging and video chatting between development teams scattered over different geographical locations.

Goals of Collaboration Server and Virtual Display System

- It is used to examine project metrics and assign responsibilities to work groups.
- It is used to monitor work assigned to all development teams, perform data analysis and store project related documents.
- It can be used to schedule meetings between various work groups.
- It is used to store metadata information that collaboration tools and services utilize to extract work related information or report data.
- It is used to generate statistics related to work progress.
- It gives provision for real time data sharing.
- It provides video chat and messaging functionalities.

1.5 <u>REPOSITORY SERVER:</u>

<u>Introduction:</u>

The repository server is responsible for holding the project's baseline and all other packages that make up the project, even if they are not part of a module. The repository server also manages dependencies between different modules and stores records of these dependencies. It is also used to store design documents, concept documents, test results, reports etc. needed for the development team. It provides facilities for users to insert and extract files. If a user intends to insert files into the repository, the file must undergo check-in process. When a file needs to be extracted, it must undergo check-out process.

Each time a file check-in or file check-out process occurs, notifications are sent to the users, project members and file owners. Authentication of check-in and check-out process by file owners is imperative.

The different processes that take place in the repository are:

- Versioning
- Notification
- Authentication
- Check-in
- Check-out

<u>Goals of Repository Server:</u>

- It is responsible for maintaining the software baseline.
- It manages dependencies between various packages and keeps records pertaining to dependency information.
- It must restructure module and subsystem content depending upon file versions. i.e in case a new version of a certain package or module is pushed onto the repository, the subsystem containing the earlier version needs to be modified to hold the newer version. It should also store older versions of packages in case the user makes requests for older version.
- Must support check in and check out processes for files.
- It must authenticate check-in and check-out instances of files to make sure accessibility is restricted only to members working on the project.

- It must send out notification messages each time a check-in or check-out instance occurs.

1.6 BUILD SERVER:

Introduction

The Build Server poses as a major entity in the Software Collaboration Federation architecture. The intention behind employing build servers in a federation system is to generate execution images of packages stored in the repository on invocation by the test harness. It can either generate library files(.dll) or execution images(.exe) based on the nature of packages stored in the repository. The build server also incorporates a cache for storing a fair amount of build images, thus providing quicker access to the test harness for build packages. If the cache does not contain requested files, it receives them from the repository and performs build operation. If the build process fails, error messages are forwarded to both the repository and the test harness requesting build images.

Goals of Build Server:

- To accept build request messages from the test harness.
- Request source files from the repository server needed for each build.
- Examine metadata information for each source module on the repository and fetch relevant file versions.
- Forwarding build images to the test harness if build is successful.
- Cacheing source code previously used for builds, thus ensuring quicker access. Cacheing has a direct effect in reducing the number of calls to the repository, thus increasing operation efficiency.
- Forwarding error messages to both the repository and test harness when packages, or sets of packages in the requested module fails to build.

1.7 TEST HARNESS:

Introduction

The Test Harness is used for automating tests. It is the busiest system on the software collaboration federation and is responsible for continuous test and integration (CTAI) of code into the software baseline.

Different clients in the software collaboration foundation host windows communication (WCF) endpoints for enforcing communication service. They establish channels to the test harness service endpoint in order to forward test requests.

When a test request message has been forwarded to the test harness, it parses the message and extracts the XML test request file that was initially packed along with the message. For each test request, a new thread is spawned from the main test harness/test executive thread, and the test requests are forwarded to each of the threads. From within each of the child threads, parsing of XML test requests is carried out, thus revealing content relevant to that test. This activity is followed by fetching build images required for the specific test from the build server and writing them into files in unique directories corresponding to the author that generated the test in the first place. Once this operation is done, the child thread initiates the creation of a child appdomain and injects the loader to which a callback reference is passed along with the generated parse results and directory information from where test related files can be sought. The loader initiates test execution and writes test results into log files in the unique directory corresponding to the test request. Also, the loader uses the callback reference to send back test results that can be accessed by its parent thread holding a reference to the same callback object. These results are then built into a form(message) that can be transmitted over the communication channel to the client that updates its graphical window dynamically to project these results.

Goals of Test Harness:

The goals of the Test Harness are:

- To enforce continuous integration of code into the software baseline.
- To automate the process of testing code.
- To implement efficient message passing communication between different service endpoints.
- To enforce concurrent test execution using simple and efficient multi-threading models and inter-thread communication.

1.8 <u>CLIENT:</u>

<u>Introduction:</u>

The client provides an interface between users and components of the software collaboration federation(SCF). It serves several groups of users, working from remote locations by providing user interfaces for system access. Through the graphical user interface, the user will be able to access tools and services offered as part of the federation. The client display is used to present charts, graphs, documents, tables, reports etc.

Through the user interface, the user can initiate file check-in and check-out processes after authentication, in tandem with the repository. Files can also be cached at the client end, thus reducing access time.

<u>Goals of Client System:</u>

- The client is responsible for creating and sending messages to the repository and test harness.
- File check-in and check-out processes are initiated from the client end.
- It also provides user authentication features to make sure only authorized members can gain access to services provided by the software collaboration foundation.
- It provides a user interface for viewing messages, reports and results from the test harness and repository server.
- It is also used to establish contact with other users or project members who may be located at remote locations across the globe.

1.9 <u>INTERFACES:</u>

Different components of the software collaboration federation interact with each other in different ways. These interactions usually happen in the form of messages and notifications over the network. The interfaces or interactions between different components of the software collaboration federation are,

Client and Collaboration Server (and VDS): The client can take advantage of all collaboration tools and services offered by the collaboration server. The client can obtain work packages, go through concept documents, design documents and presentations shared on the server. Also, the client can take part in live video conferences, make presentations and engage in communication with other project members scattered over different geographical locations with the help of the collaboration server and virtual display system.

Collaboration Server (and VDS) and Repository: Team leaders, project managers and software architects can share important project related information, retrieve concept and design documents shared on the repository and host them on the collaboration server, analyze file dependencies by going through metadata information shared on the repository.

Client and Test Harness: The client sends test requests to the test harness. The test harness processes the test request and initiates testing after fetching necessary files from the build server. Once testing is done, the test harness sends back test results to the client.

Test Harness and Build Server: Once the test harness receives test requests from the client, it sends a request to the build server asking for build images required for testing. The build server initially looks for file in its cache. If required files are present in the cache, it generates build/execution images of those files and sends them back to the test harness. If the files are not present in the cache, the build server requests the repository server for those files, generates build images and forwards them to the test harness.

Build Server and Repository: Test code and test suites for testing are initially uploaded to the repository through check-in process. When the build server receives requests for build images from the test harness, it seeks source files from the repository if they are not present in its cache. The repository fetches the required source files and sends them back to the test build server.

Test Harness and Repository: Test logs and Test results are sent to the repository server after testing.

1.10 USE CASES:

This section describes principle users of the software collaboration foundation, their expectations from the SCF, ways in which they will interact with the tool and characteristics that will be incorporated into the federation to cater to user requirements.

The federation is established to aide developers, project managers, software architects, team leads and the testing team involved in large enterprise level software projects. The main aim of the federation is to support continuous testing and integration of code(CTAI). With the help of the federation, project members can perpetually test code against the existing software baseline. This ensures consistency in previously developed and tested code, as well as to-be tested code. From developers who develop and test code, to customers who analyze the quality of the product, all members directly or indirectly involved in software development process are benefitted by the federation. The users of the software collaboration federation are,

a) **Software Developer**

- Through the SCF the developer achieves continuous test and integration(CTAI) of code into the software baseline.
- Provide efficient means for developers for modular integration and ensure consistency in the baseline.
- Monitor test results, logs and failed test cases.
- They can hold live meetings over the network with other project members working from various geographical locations.
- They can share concept documents, design documents and test reports with other project members, through the collaboration server.
- Support check-in of the source code files along with test drives, build script, and test files.
- Supports file check-out process.
- The developer can use existing code modules that promote reusability.
- Examine impact of code change during integration.
- Receive notifications on local client displays.
- To analyze test results, log results and go through summaries of tests.

b) Software Architect

- Upload concept documents, design documents and project presentations onto the repository for access to members involved in project development.
- They can record changes in project design and send out notifications to all members about this development. They can also notify members when new versions of operational concept documents or design documents are uploaded.
- Schedule meetings and send out meeting requests.

- Monitor progress reports and notifications.
- To comprehend project structure, using package dependency information and go through project documentation to gain better understanding of modular interaction.

c) Project Manager/ Team Leader

- The project managers and team leaders utilize the federation to monitor all documents and changes made to documents. They keep of track of collective and individual progress made on the project.
- They can create, host and schedule work packages to project members.
- Ensure that the project is being undertaken as expected.
- Monitor the total tests passed and failed for a certain developer's test suite and return test results and test summaries.
- Oversee file/module dependency information and consistency in file naming and versioning.
- Go through project reports and track productivity.

d) Testing Team

- To validate and ensure that all intended requirements are conformed to.
- Make sure that tests run without any errors.
- Go through test results and reports to gain a comprehensive understanding of test iterations.
- Write clean and efficient test cases for testing different aspects of the applications
- Verify business and technical requirements of the project.
- Identify problems during testing and pass suggestions to developers for correcting areas of code that engendered inconsistencies during testing.
- Save test results, reports and test summaries for future reference.

d) Customer

- To communicate to the project team in case of developments or changes in project requirements
- Take part in important meetings for discussing project related requirements and strategies.
- Analyze quality of software product after development.

1.11 CORE SERVICES AND POLICIES:

Core Services:

- **Blocking Queues**

The Blocking Queue provides a thread safe implementation of a queue for storing and retrieving messages. It provides synchronization between threads and helps avoid deadlock situations. This synchronization policy followed by the blocking queue ensures that only thread enqueues or dequeues items into or from the queue at a particular instant of time. If a thread is trying to dequeue items from the queue and the queue is found empty, the thread blocks. This thread remains blocked until the queue contains some items that the thread waiting to dequeue can access.

- **Notifications**

Notifications are messages exchanged between two or more systems in the software collaboration federation. When a system interacting with another system wants to indicate the occurrence of a certain event, a notification is sent. For example, during authentication of the user, the user's login credentials are matched against database entries to check if he/she is a registered user. If the user is not registered, an authentication error notification is sent back. Another instance where notifications come into picture is when the test harness processes test requests sent by users. Once testing is done, the test harness sends back test results in the form of messages. Also, the client can obtain status of the test from the test harness. Clients receive test status in the form of notifications.

- **Security Services**

Security becomes a very important aspect while handling documents that belong to different groups of users. Clients need to be authorized. i.e their login credentials need to match against an existing database entry to make sure that the user is registered. Only then, the user can perform check-in and check-out processes of files. Documents like design views need to be encrypted access lies only within the reach of project members. There are many encryption algorithms like RSA, public key algorithm. These security algorithms provide more security to data that is deemed confidential.

- **Metadata Management**
  Metadata is data that provides information about other data. It is used to describe the content and context of data files. Metadata query uses tag names as the basis for querying. Tags describe file properties by using the keywords, text descriptions, date information, dependency information and file format information. Usage of metadata for querying makes up for effective searching since search quality increases by using metadata information.

Core Policies:

Policies are principles used to guide decisions and render solutions during operational issues. Policies are usually defined by the higher-level management of a company. Some of the policies are ownership policy, notification policy, versioning policy.

- **Ownership Policies:**
  This is one of the primary policies defined by companies. Ownership can be with respect to files, documents, source code, test summaries, test logs, reports etc. It all depends on the security of the document and how important the document is. There can be four plausible cases of ownership. They are single ownership, group ownership, multiple ownership or no ownership.

  Developers claim ownership over the code that they have developed. Single ownership does not define file locking policies since it is always the owner who performs checks in and check out operation. The ownership is fully claimed by the developer in this case and other users have minimal access.

  During code integration, source code developed by a certain developer becomes accessible to each member whose module it has been integrated with. In this type of ownership, locking mechanisms need to be implemented since file confidentiality is of prime importance. Only those members who have authorization will be able to access files.

  In cases where team leaders and project managers have access to confidential files, it is important to employ sophisticated file locking mechanisms that offer better security.

  In case of no ownership policy, files and documents are open to all members for viewing and editing.

- **Versioning:**
  Versioning is an important policy that comes into picture while handling thousands of files and documents belonging to people working in different geographical locations. When multiple versions of the same file are released, versioning of files need to be done. Versioning also indicates changes made to files to the previous versions and helps developers to overcome errors in previous versions. Versioning also makes file searching simpler since naming conventions are unambiguous. For every change made to the file, the base version gets updated to reflect the change.

- **Cache Policy:**
  Caching of files can be done in order to limit the number of files transmitted over the network. When files are frequently requested, the server must store them in the cache to support faster file access. Cache memory is limited in capacity, and the duration for which files reside in the cache needs to be taken into consideration. Thus, cacheing of files depends upon the amount of storage capacity allocated for the cache. In order to avoid encountering memory shortages, files that are not often used must be discarded from the cache. Another issue is the presence or absence of files in the cache when a file request is made. The easiest way is to create a dictionary with key as file name and value contains path to the file and version information of the file. So, whenever a file of a certain version is needed, it will check the dictionary to see if it is present in the cache. If the file is present, it is loaded from the cache. If not, it is fetched from external memory.

1.12 CRITICAL ISSUES:

- **Server Performance**

  The software collaboration federation has thousands of users operating from all over the world to achieve a common objective. When a large number of developers are concurrently working on the project, and huge amounts of data being utilized, server performance becomes a key factor that must be taken into consideration.
  In this case, document or file fetching becomes a time-consuming task since servers host thousands of files contributed by project members.
  A solution to this is to use high performance servers. Also, the files can be stored on the cache, so that file access becomes fast. Unwanted files can be discarded and performance of the server can be improved.

- **Concurrent file access**

  When there are large numbers of people working on the same project, there could be scenarios where developers, clients would want to access the same file or other resources from the server at the same time. At that time the same file may be edited by both the users and also to maintain the log information would be difficult. Versioning of the document also would be major problem in such situations.

  In such cases, it is always better to have some kind of lock mechanism. Whenever there is a file being accessed by one person the file cannot be accessed unless the user finishes all the actions with that file. Then the versioning can be done and the next user can access the file without any discrepancies. Also, there could be some kind of queue that can be maintained in order to access the file.

- **Server Issues**

  When servers are loaded with thousands of operations and the server goes down, all the processes in the system are stalled.

  In such cases when the server goes down there is no procedure to deal with such measures. Hence, it is imperative to use redundant servers that can take up processing responsibility when a particular server goes down, thus maintaining continuity in the operation.

- **Memory Management**

Memory Management becomes a critical issue when huge amounts of data is involved. Since the number of files on the server is huge, space management becomes a concern, and hence it must be properly looked after.

One solution is to maintain large amount of memory in the servers. At the same time, unwanted files can be discarded from the servers. This frees up space on the servers, thus paving space for critical project related information.

- **File format issues**

  Users need to be careful about file formats that servers can handle. If incompatible files are uploaded onto the servers, during processing, we may encounter unpredictable behavior.
  Hence it is necessary to define file formats beforehand, so that users can only submit files with specified formats to the servers, thus avoiding erratic behavior.

- **Power Management**

  When developing new software, it is necessary to maintain the power management hierarchy. This should define privileges each developer, user, client or managers are entitled to like who has the maximum authority? who can access files which are not from there team? When there is a clash between 2 developers who solves the dispute? And also, any design issues etc.

- **Security**

  Security becomes a key concern when large amounts of data and thousands of developers are involved in the project. Data integrity becomes an important issue in this case.

  Each developer, user, etc. should be given log in credentials so that only they can gain access to servers and also upload or edit the files on the server. Only authorized users can perform specific operations on the servers.

2. STORAGE MANAGEMENT SUBSYSTEM (SMS)

2.1 INTRODUCTION:

The Storage Management Subsystem(SMS) is a facility designed to support storage and management of data. The Storage Management Subsystem can be used to designate data allocation characteristics and storage requirements in the software collaboration federation. The key components of the Storage Management Subsystem are Data and Event Managers hosted on all client systems and servers of the software collaboration federation. It consists of centralized and data and event coordinators that supervise the activities of all data and event managers on all SCF components. They are also responsible for establishing collective security of the SCF.

2.2 GOALS OF STORAGE MANAGEMENT SUBSYSTEM

- To manage communication between different SCF components.
- Providing communication services like notifications and messages between different client and server systems.
- Imposing storage constraints on servers by defining what needs to be stored.
- To manage storage of test results, test reports and summaries.
- To manage storage of metadata information on collaboration servers.
- To manage storage of test files, test suites and documents on the repository.
- To configure and supervise events.
- Define structure of messages and design of virtual display system.

2.3 ORGANIZING PRINCIPLES OF SMS

Some of the organizing principles of SMS are,

- The Storage Management Subsystem is present in each client and server system of the SCF.
- It provides data and event management services to all components.
- It consists of centralized data and event coordinators that oversee the operations of all data and event managers.

2.4 ARCHITECTURE OF SOFTWARE COLLABORATION FEDERATION USING SMS:

The architecture of the software collaboration federation with storage management subsystem is shown below,

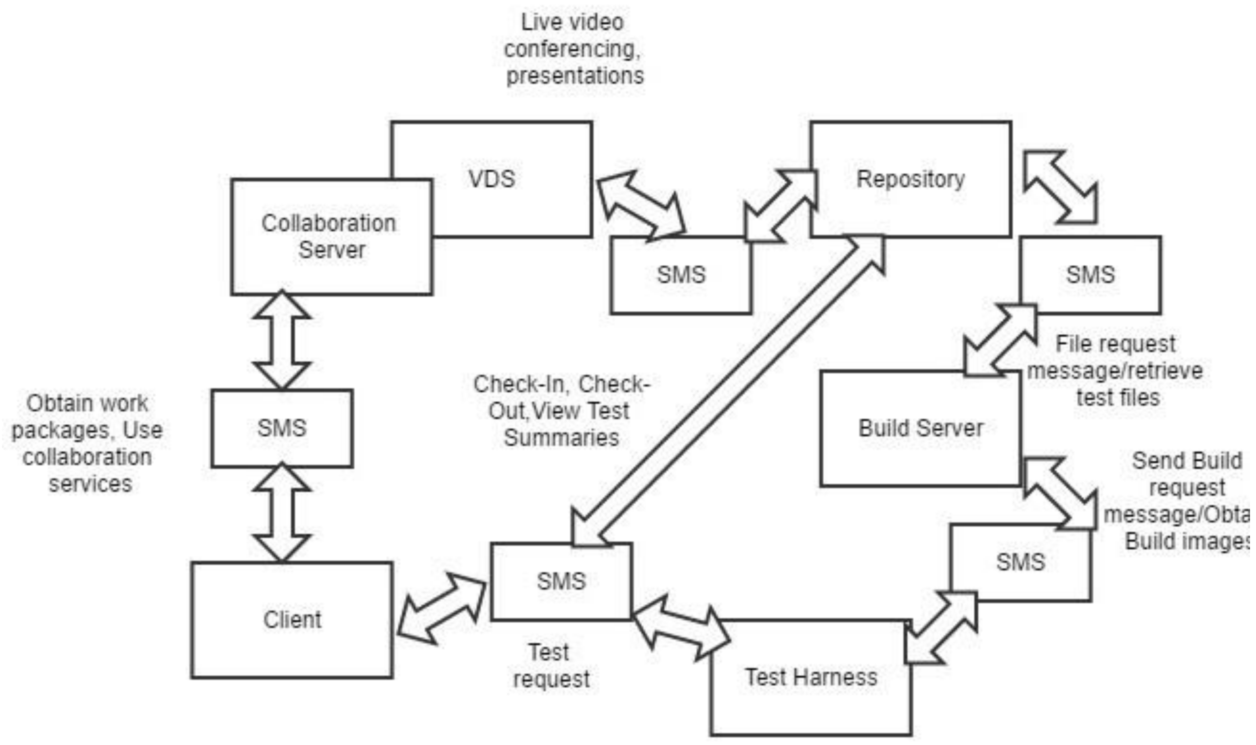

Live video conferencing, presentations
VDS
Collaboration Server
SMS
Repository
SMS
Obtain work packages, Use collaboration services
SMS
Check-In, Check-Out,View Test Summaries
File request message/retrieve test files
Build Server
Send Build request message/Obta
Build images
SMS
Client
SMS
Test request
Test Harness

FIG. Software collaboration federation using SMS

Different components of the software collaboration federation host data and event managers that help manage, store and communicate data. All interactions between SCF components take place through the SMS. Check-in, Check-out requests, request for test results and test summaries, test requests, build requests and build image retrieval, file requests, use of collaboration tools and services etc. are overseen by the storage management subsystem.

3. COMMUNICATION AND MESSAGES

Interaction between components of the software collaboration foundation mainly happens through messages and notifications. Message formats are initially determined before any sort of communication is established between the components. When web services are utilized for communication, message formats are defined by data contracts that indicate the structure of the message that must be transmitted over channels. Example message body formats for each interaction are shown below,

- **Client to Repository Server**:

```
<?xml version="1.0" encoding="utf-8"?>
<Check-In Request>
        <SenderAddress> http://remotehost:8080</SenderAddress>
        <ReceiverAddress> http://remotehost:8085 </ReceiverAddress>
        <Description>
                <Key>1</Key>
                <Metadata>File2.xml</Metadata>
                <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
                <Files>
                        <FileName> File 1</FileName>
                        <FileName> File 3</FileName>
                <Files>
        </Description>
</Check-In Request>
```

- **Repository to Client**

```
<?xml version="1.0" encoding="utf-8"?>
<Response>
        <SenderAddress> http://localhost:8085 </SenderAddress>
        <ReceiverAddress>> http://localhost:8080 </ReceiverAddress>
        <Description>
                <Key>1</Key>
                <Metadata> File2.xml</Metadata>
                <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
                <Files>
                        <Filename> File 1</Filename>
                        <Filename> File 3</Filename>
                <Files>
        </Description>
```

```
</Response>
```

- **Client to Collaboration Server**

```
<?xml version="1.0" encoding="utf-8"?>
<Request>
        <SenderAddress>> http://localhost:8080</SenderAddress>
        <ReceiverAddress>> http://localhost:8082 </ReceiverAddress>
        <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
        <Description>
                <Tool> Request for work package</Tool>
        </Description>
</Request>
```

- **Client to Test Harness**

```
<?xml version="1.0" encoding="utf-8"?>
<TestRequest>
                <SenderAddress> http://localhost:8081 </SenderAddress>
                <ReceiverAddress> http://localhost:8083 </ReceiverAddress>
                <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
                <Description>
                        <Files>
                                <Filename> File1.cpp</Filename>
                                <Filename> File2.cpp</Filename>
                        <Files>
                </Description>
    </TestRequest>
```

- **Test Harness to Build Server**

```
<?xml version="1.0" encoding="utf-8"?>
<BuildRequest>
                <SenderAddress> http://localhost:8083 </SenderAddress>
                <ReceiverAddress> http://localhost:8085 </ReceiverAddress>
                <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
                <Description>
                        <Files>
                                <Filename> File1.cpp</Filename>
                                <Filename> File2.cpp</Filename>
                        <Files>
```

```
                    </Description>
        </BuildRequest>
```

- **Build Server to Repository**

```
<?xml version="1.0" encoding="utf-8"?>
<Request>
            <SenderAddress> http://localhost:8085 </SenderAddress>
            <ReceiverAddress> http://localhost:8087 </ReceiverAddress>
            <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
            <Description>
                  <Files>
                        <Filename> File1.cpp</Filename>
                        <Filename> File2.cpp</Filename>
                  <Files>
            </Description>
  </Request>
```

- **Repository Server to Build Server**

```
<?xml version="1.0" encoding="utf-8"?>
<Response>
         <SenderAddress> http://localhost:8087 </SenderAddress>
         <ReceiverAddress> http://localhost:8085 </ReceiverAddress>
         <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
         <Description>
               <Files>
                     <Filename> File1.cpp</Filename>
                     <Filename> File2.cpp</Filename>
               <Files>
         </Description>
</Response>
```

- **Build Server to Test Harness**

```
<?xml version="1.0" encoding="utf-8"?>
<Response>
            <SenderAddress> http://localhost:8087 </SenderAddress>
            <ReceiverAddress> http://localhost:8085 </ReceiverAddress>
```

```
                <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
                <Description>
                        <Files>
                                <Filename> File1.cpp</Filename>
                                <Filename> File2.cpp</Filename>
                        <Files>
                </Description>
    </Response>
```

- **Test Harness to Client**

```
<?xml version="1.0" encoding="utf-8"?>
<Response>
                <SenderAddress> http://localhost:8085 </SenderAddress>
                <ReceiverAddress> http://localhost:8081 </ReceiverAddress>
                <Timestamp>2015-10-09T13:18:05.7183418-04:00</Timestamp>
                <Description>
                        <Test>
                                <Testname>FirstTest</Testname>
                                <TestStatus> Passed</TestStatus>
                                <Logs> Log1.txt</Logs>
                        <Test>
                        <Test>
                                <Testname>SecondTest</Testname>
                                <TestStatus> Failed</TestStatus>
                                <Logs> Log1.txt </Logs>
                        <Test>
                </Description>
    </Response>
```

- **Collaboration Server to Repository Server**

```
<?xml version="1.0" encoding="utf-8"?>
<Message>
        <SenderAddress>>> http://localhost:8082</SenderAddress>
        <DestinationAddress>>> http://localhost:8085 </DestinationAddress>
        <Description>
                <Document>
                        <Documentname>OCD.txt</Documentname>
                <Documentname>DesignSpecs.txt</Documentname>
                <Documentname>ProjectReport.txt</Documentname>
            </Document>
        </Description>
    </Message>
```

4. BUILD SERVER

4.1 INTRODUCTION:

The Build Server poses as a major entity in the Software Collaboration Federation architecture. The intention behind employing build servers in a federation system is to generate execution images of packages stored in the repository on invocation by the test harness. It can either generate library files(.dll) or execution images(.exe) based on the nature of packages stored in the repository. The build server also incorporates a cache for storing a fair amount of build images, thus providing quicker access to the test harness for build packages. If the cache does not contain requested files, it receives them from the repository and performs build operation. If the build process fails, error messages are forwarded to both the repository and the test harness requesting build images.

4.2 KEY GOALS OF BUILD SERVER:

- To accept build request messages from the test harness.
- Request source files from the repository server needed for each build.
- Examine metadata information for each source module on the repository and fetch relevant file versions.
- Forwarding build images to the test harness if build is successful.
- Cacheing source code previously used for builds, thus ensuring quicker access. Cacheing has a direct effect in reducing the number of calls to the repository, thus increasing operation efficiency.
- Forwarding error messages to both the repository and test harness when packages, or sets of packages in the requested module fails to build.

4.3 ACTIVITIES:

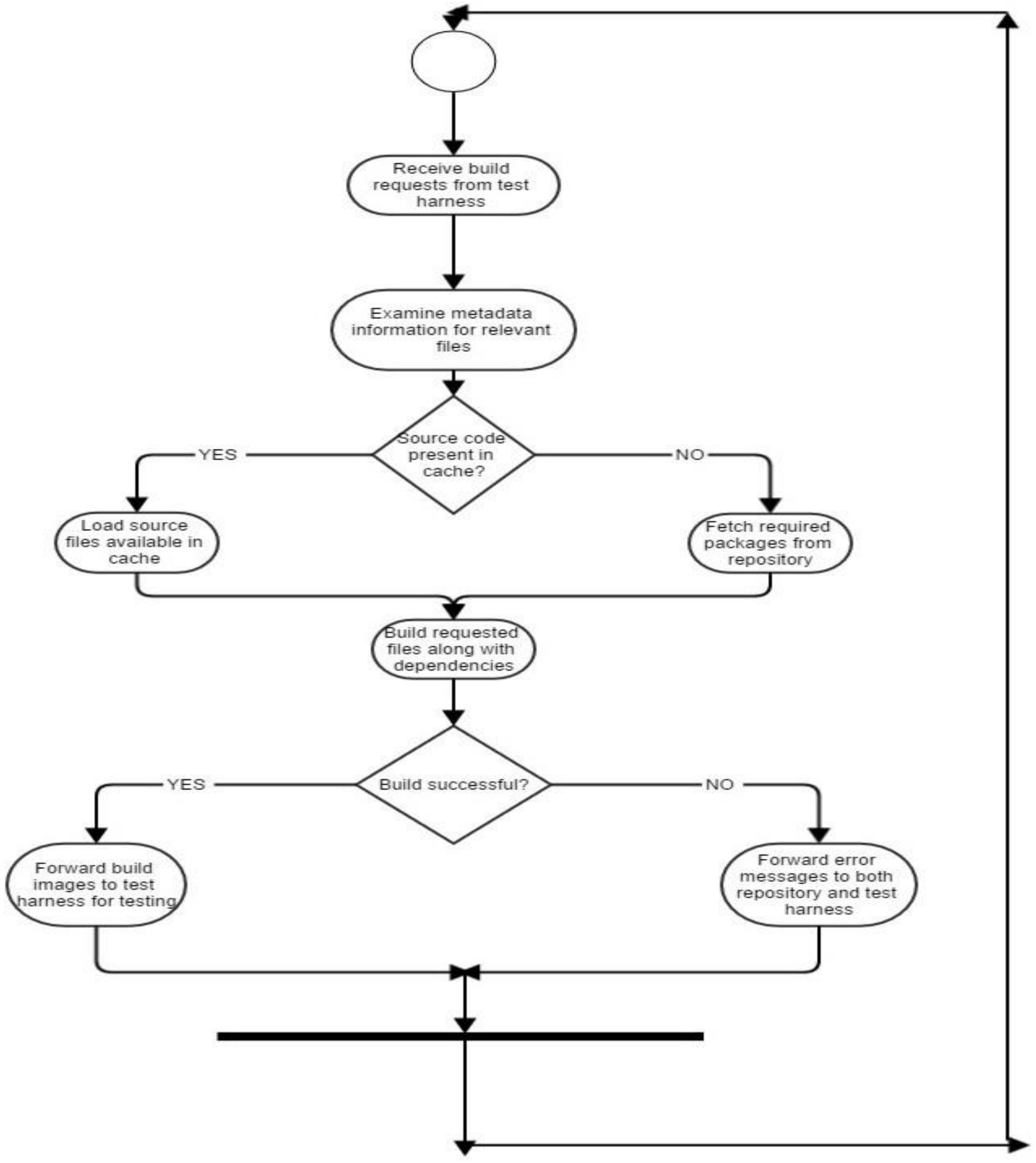

Receive build requests from test harness
Examine metadata information for relevant files
Source code present in cache?
YES
NO
Load source files available in cache
Fetch required packages from repository
Build requested files along with dependencies
Build successful?
YES
NO
Forward build images to test harness for testing
Forward error messages to both repository and test harness

Fig. Activity diagram of build server

The above diagram provides a general flow of operation of a build server. The following points explain its working in detail,

- The Build Server continuously listens to build requests from the test harness.
- Upon arrival of a build request, the server consults the repository for XML metadata information concerning the requested module for dependency and version details.
- The server incorporates a cacheing hardware that stores source code for previously built modules. This cacheing mechanism speeds up build operation since it limits the number of repository calls to only those modules not already present in the cache.
- If the requested files are present in the cache, the server loads them up for compilation. If not, the build server makes a request to the repository for requested packages.
- Once it receives packages from the repository, it generates build/execution images depending upon the nature of the module.
- If the build process succeeds, the build images are forwarded to the test harness for testing. If not, the build server notifies both the test harness and repository of the failure. This way, developers of a certain module can be made aware of flaws in their code submissions.

4.4 PACKAGES:

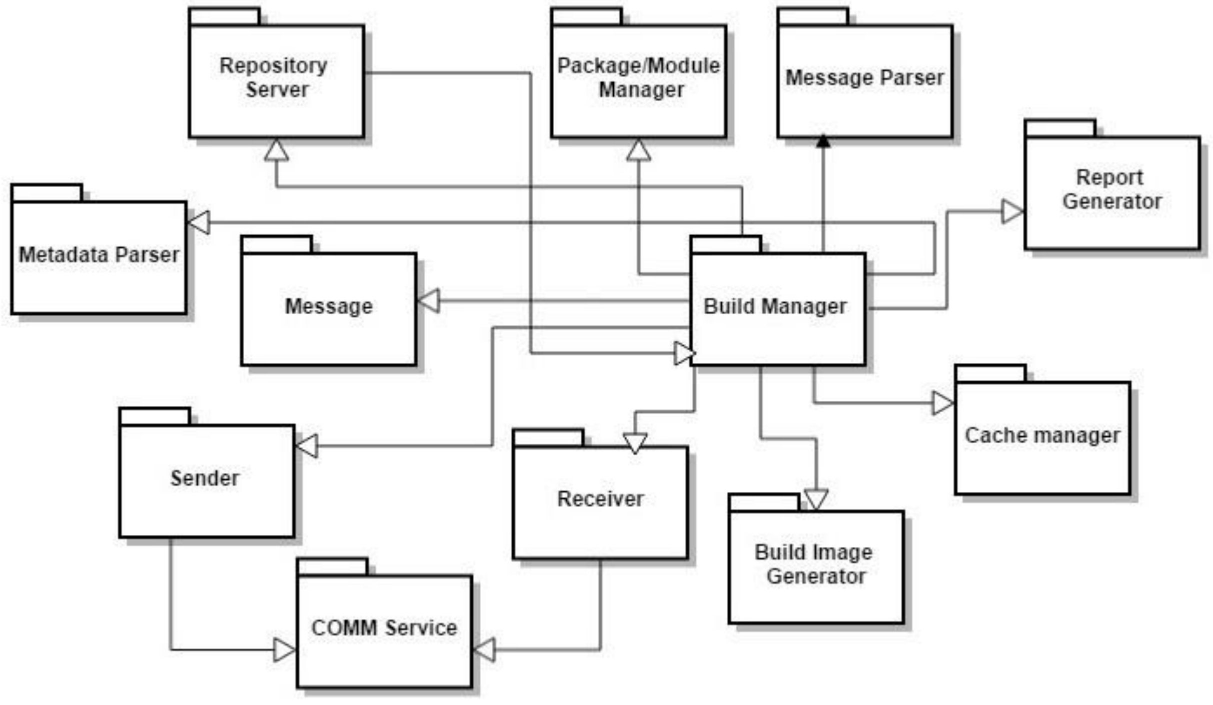


FIG. Package Diagram of Build Server

- **Build Manager:** The build manager is the engine behind the core server operation. It monitors the mechanics of the server and controls the flow of the operation mentioned henceforth. It relies on a collection of packages for discharging its duty. On receiving the build request from the test harness, it parses the request message to reveal the packages/modules needed for the test. It checks for module availability in the cache using the cache manager. If the requested module is present on the cache, courtesy of a previous build instance, it loads the files from the cache and builds them. If they are not available, it fetches the modules from the repository server through its communication service, generates build images and forwards it to the test harness in the form of communicable messages. In case of a build error, both the test harness and repositories are notified.
- **Build Image Generator:** This package is responsible for generating build/ execution images of modules needed for testing. The entire build operation is leveraged on the build image generator.

- **Message Parser:** The message parser parses the incoming build request message to get module information needed for testing. This message format is usually defined by the communication service.
- **Metadata Parser:** The metadata parser is used to verify changes in module versions to make sure that newer module versions are fetched over older versions possibly stored on the cache.
- **Message Builder:** The server uses a message builder to communicate build success or failure result to both the test harness and the repository. File transfer takes place using a pre-defined file transfer protocol.
- **Cache Manager:** The cache manager manages storage and expulsion of modules stored on the cache. When a build request is received, the cache manager will be responsible to fetch modules from the cache if present. This presence or absence of modules is governed by incidents of previous builds of the same module.
- **Package/Module Manager:** The build server uses the package/module manager to check for file availability in its local cache. The server uses this cache to speed up the process of file access.
- **Comm Service:** The Comm service (or communication service) sets standards for communication between different service endpoints. It dictates rules for communication between different service endpoints.
- **Sender/Receiver:** The server consists of a sender and receiver for each communication endpoint that it interacts with. Sending and receiving processes are governed by the communication service.
- **Message:** This package defines the format of messages that are communicated between subsystems (in this case, repository server and the test harness).
- **Report Generator:** It is used for building build reports that the testing team or project manager consults during module development and testing.

4.5 <u>INTERACTIONS:</u>

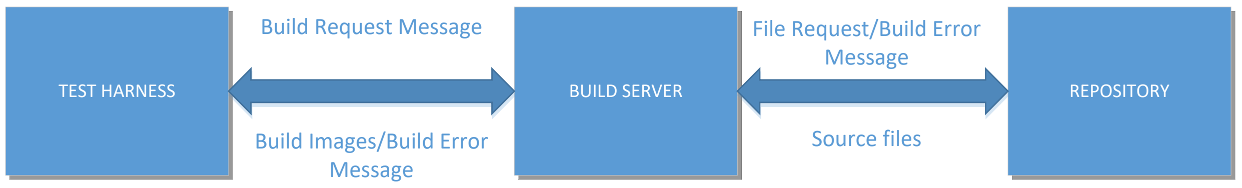


FIG. Interactions of build server with test harness and repository

The interactions of the build server with the test harness and repository are described below,

- **Test harness-Build server:**
  1. The test harness sends build request messages to the build server. The message includes details of all packages that belong to a certain module and packages that the module is dependent on. It is the responsibility of the build server to generate build/execution images of requested files and supply it to the test harness.
  2. In case of a build failure, the build server sends a notification to the test harness indicating build status.

- **Build Server-Repository:**
  1. The build server, upon receiving build request messages from the test harness, reaches out to the repository for source files/modules whose testing intends to be done in the test harness server.
  2. In case of a build failure, the build server sends a notification to the repository indicating build status.

4.6 <u>USES:</u>

- The Build Server is used for generating build images of modules requested by the test harness for testing. This code compilation process carried out by the server ensures integrity of source code, implying that code running on a certain machine can be reproduced on another.

- The Build Server is configured to send error messages to the test harness and the repository in case of build failure. This notifying process can be used for clean code development.

4.7 USERS:

- **Developers:** Developers depend on the build server for building modules required for testing.
- **Testing Team/Project Manager:** The test team and project manager use the build server to ensure coherence in developed modules. This is carried out by subjecting developed modules to the build process on the server.

4.8 CRITICAL ISSUES:

- **Versioning issue:** The build server maintains an internal cache to store a set of source files whose build images were previously generated. If a build request arrives at the build server, and the server finds out that the requested module is available on its cache, it generates a build images of packages in that module, thus overlooking newer code versions pushed onto the repository. A solution to this problem is for the build server to receive check-in information by parsing metadata files associated with requested modules from the repository.
- **Administration issues:** Building source code developed under different protection environments could pose a serious problem. Thus, we must ensure that the servers can handle administration issues by reconditioning them.
- **Handling large payload:** It is imperative to employ considerable amount of build servers that can handle huge number of build requests. This can be achieved by using efficient multi-core or multi-processing servers that can process large amounts of data in not-so-long periods of time.

5. COLLABORATION SERVER AND VIRTUAL DISPLAY SYSTEM

5.1 INTRODUCTION:

Software development is a long and convoluted process that involves hundreds of developers working in multiple groups. Tracking work progress becomes too tedious in such large-scale projects. This problem can be solved by equipping collaboration servers that maintain work packages and help track each of the work packages. The collaboration server provides exploratory tools and services based on metadata information, beneficial for project managers and team leaders. An example is a tool that generates work package and pending work package information using which team leaders and developers can track progress.

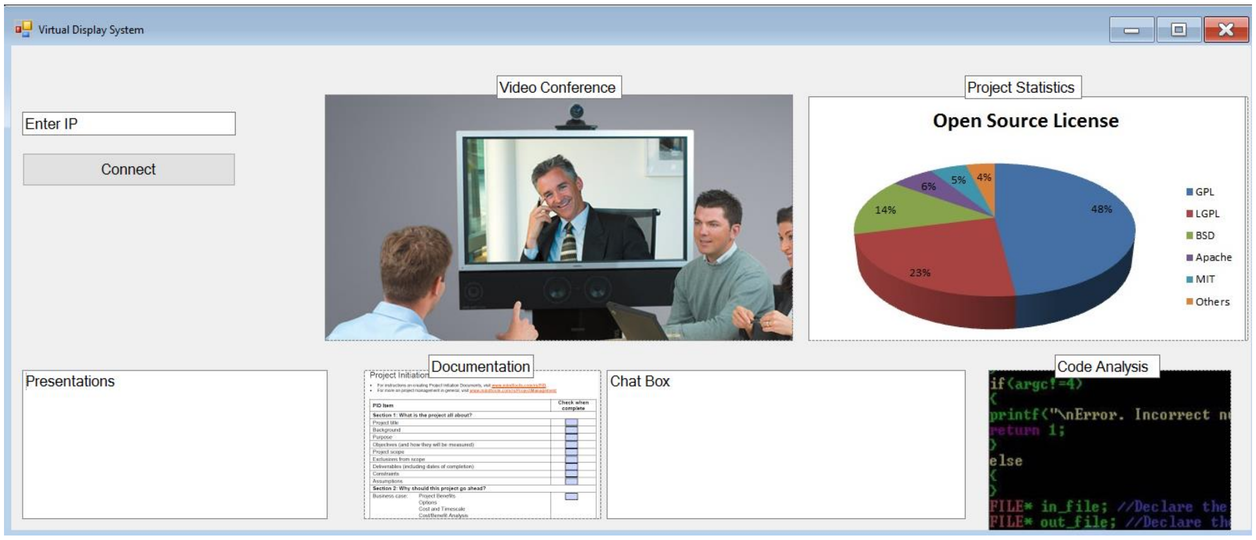
The collaboration server is also used for scheduling virtual meetings and storing details pertaining to those meetings. Design and implementation details can be stored on the server and made accessible to developers engaged in a session. Through the collaboration server and Virtual Display System(VDS), it is possible to have full-fledged discussions over the network.

The virtual display system(VDS) provides a graphical display or user interface through which test details, work package details, reports and design diagrams can be manifested. The virtual display system also ensures provision for messaging and video chatting between development teams scattered over different geographical locations.

5.2 KEY GOALS OF COLLABORATION SERVER AND VIRTUAL DISPLAY SYSTEM

- It is used to examine project metrics and assign responsibilities to work groups.
- It is used to monitor work assigned to all development teams, perform data analysis and store project related documents.
- It can be used to schedule meetings between various work groups.
- It is used to store metadata information that collaboration tools and services utilize to extract work related information or report data.
- It is used to generate statistics related to work progress.
- It gives provision for real time data sharing.
- It provides video chat and messaging functionalities.

5.3 VIRTUAL DISPLAY SYSTEM SCREEN:



Project Initiation

- For instructions on creating Project Initiation Documents, visit [illegible]
- For more on project management in general, visit [illegible]

PID Item		Check when complete
Section 1: What is the project all about?		
Project title		☐
Background		☐
Purpose		☐
Objectives (and how they will be measured)		☐
Project scope		☐
Exclusions from scope		☐
Deliverables (including dates of completion)		☐
Constraints		☐
Assumptions		☐
Section 2: Why should this project go ahead?		
Business case:	Project Benefits Options Cost and Timescale Cost/Benefit Analysis	☐

```
if(argc!=4)
{
printf("\nError. Incorrect n
return 1;
}
else
{
}
FILE* in_file; //Declare the
FILE* out_file; //Declare th
```

FIG. Virtual Display System screen

The virtual display system is a collaboration system that works in association with the collaboration server. It supports a plethora of group based activities in association with services and tools provided by the collaboration server. The figure shown above is an example VDS screen that can host live video conferences, chats, presentations, documentation, project statistics etc.

5.4 USES:

- The collaboration server is used to manage work packages and track work progress.
- Along with the virtual display system, the collaboration server is used to schedule meetings, store and maintain information needed for those meetings.
- Notifying participating members when meetings are scheduled.
- Provides an interface to represent project related ideas, graphs and diagrams.
- Stores project reports that are accessible to project members, coordinators, team leaders and managers.
- Provides video chat and messaging functionalities.

5.5 USERS:

- **Project Members:** Work packages are assigned to each team and each one's progress is monitored through the collaboration server. With the help of the collaboration server, tracking work packages becomes easy. The server, along with the virtual display system has a provision for developers to convey real-time design and implementation information and enables sharing of work related graphical data. Developers can also identify bugs in dependent work packages and raise issues so that developers concerned with the work package can rectify it.
- **Team Leaders/Managers:** Team leaders and managers assign work to the development teams in the form of modules or work packages through the collaboration server. They continuously monitor or keep track of work assigned to individual development teams. They utilize tools and services provided by the collaboration server to achieve this. They are also responsible for convening project members by scheduling and organizing meetings. Project members are alerted of important deadlines and essential information is shared through the server and VDS system.
- **Executive Team:** They assess project reports with the help of which, they contemplate future actions and improvements. They also impart details of analysis to project members.

5.6 CRITICAL ISSUES:

- **Communication delays:**

  The collaboration server in conjunction with the visual display system, is used by project managers, team leaders and software architects to schedule meetings with work groups from different geographical locations. Hence, it is important that demonstrations made by a work group at a certain location be communicated to other intended work groups immediately without much delay. If the delay is large, there will be inconsistencies in communication, ultimately affecting productivity. Hence, the collaboration server and virtual display systems must provide services for live video conferencing over direct communication channels that reduce communication latency.

- **Concurrent Access to a Service:**

  This becomes an important issue when multiple users are trying to access the same service provide by the collaboration server and the visual display system. For example, when two or more users that share the same visual display system at a location attempt to make design diagrams or presentations, there will be a service conflict. Hence, it is important to limit the number of users trying to gain access to the service at a given instant of time.

- **Server Breakdown:**
  The collaboration server hosts work packages assigned by team leaders and managers, hold critical project related information, management information. Progress of work can be monitored through the collaboration server by analyzing shared reports. If the server goes down, all information shared on the server is lost. Thus it is important to employ multiple servers that provide the same tools and services in case a server gets taken down.

6. REPOSITORY

6.1 INTRODUCTION:

The repository server is responsible for holding the project's baseline and all other packages that make up the project, even if they are not part of a module. The repository server also manages dependencies between different modules and stores records of these dependencies. It is also used to store design documents, concept documents, test results, reports etc. needed for the development team. It provides facilities for users to insert and extract files. If a user intends to insert files into the repository, the file must undergo check-in process. When a file needs to be extracted, it must undergo check-out process.

Each time a file check-in or file check-out process occurs, notifications are sent to the users, project members and file owners. Authentication of check-in and check-out process by file owners is imperative.

The different processes that take place in the repository are:

- Versioning
- Notification
- Authentication
- Check-in
- Check-out

6.2 GOALS OF REPOSITORY SERVER:

- It is responsible for maintaining the software baseline.
- It manages dependencies between various packages and keeps records pertaining to dependency information.
- It must restructure module and subsystem content depending upon file versions. i.e in case a new version of a certain package or module is pushed onto the repository, the subsystem containing the earlier version needs to be modified to hold the newer version. It should also store older versions of packages in case the user makes requests for older version.
- Must support check in and check out processes for files.
- It must authenticate check-in and check-out instances of files to make sure accessibility is restricted only to members working on the project.

- It must send out notification messages each time a check-in or check-out instance occurs.

6.3 ORGANIZING PRINCIPLES OF REPOSITORY SERVER:

- The repository server defines modules as lists of packages. It also defines subsystems as collections of modules contained in the repository store.
- It is responsible for maintaining the integrity of modules which are collections of packages accessed through an interface and object factory. Once an interface and object factory have been defined they cannot be changed without approval of the Project Manager or Software Architect. Each module is developed by one team of developers.
- Storing test suites source code and associating each test suite with a module or subsystem. Subsystem test suites may contain test suites of its individual modules and may contain additional test code as well.
- Managing check-in and check-out of packages source code to and from modules.
- Analyzing the dependency of modules and subsystems and recording the dependencies with XML metadata.
- Executing queries about dependency relationships.
- It is used to store packages that are not currently part of a module. These orphan packages are not considered part of the current baseline until they become part of a module, either by creating a new module or by adding to an existing module.

6.4 VERSIONING:

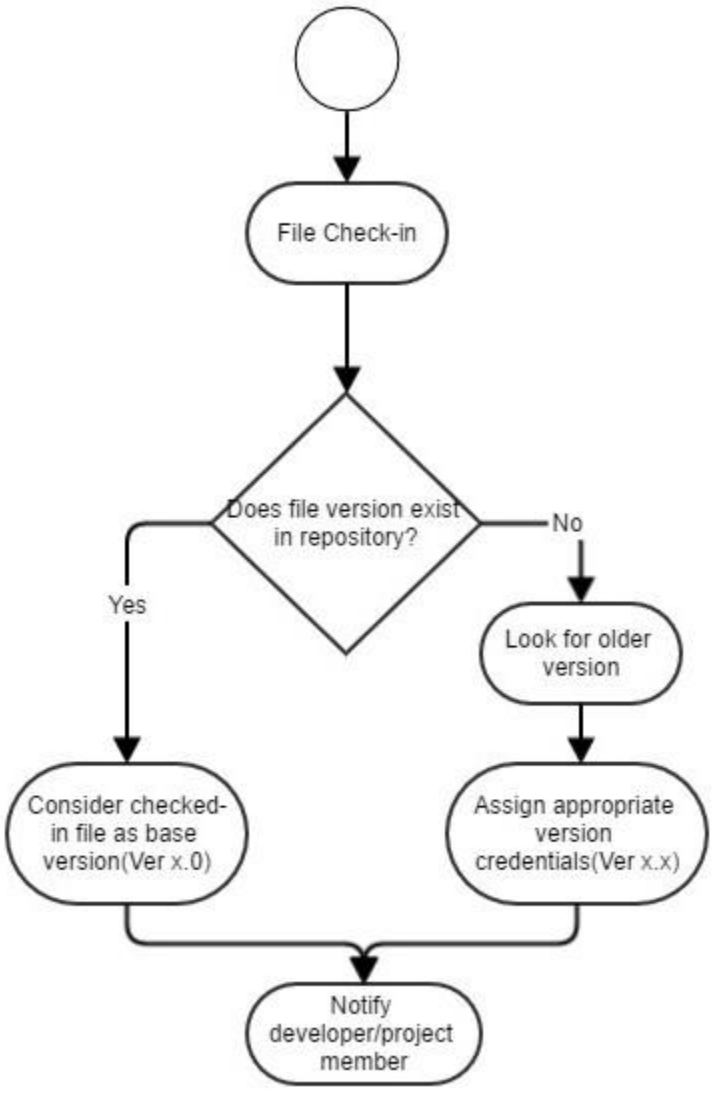

File Check-in
Does file version exist in repository?
Yes
No
Look for older version
Consider checked-in file as base version(Ver x.0)
Assign appropriate version credentials(Ver x.x)
Notify developer/project member

FIG. Versioning process

File versioning in the repository server takes place as follows,

- When there is a check-in request, the server initially checks to see if the version being pushed onto the repository is already present or not.

- If the file being submitted is a new version, it is assigned base version credentials and displayed to the user.
- If the file is old, the server looks for the older version and is provided with appropriate version credentials (Ver x.x). It is then displayed to the user.

6.5 VERSIONING CONCEPT:

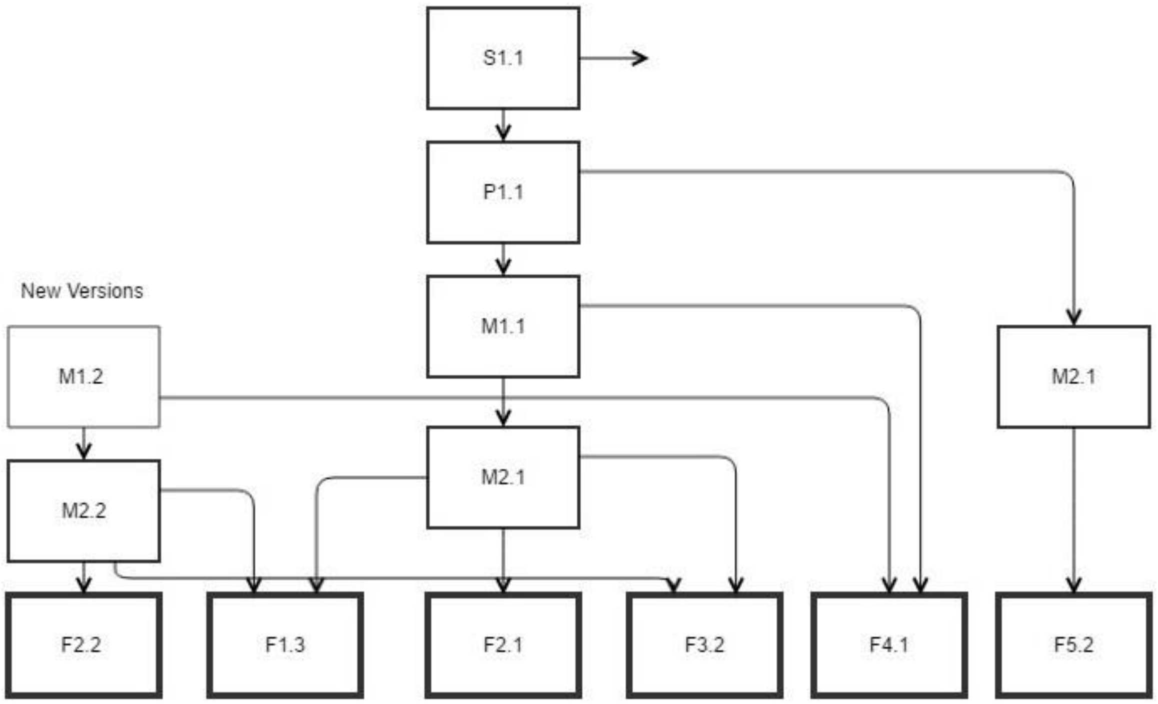


FIG. Versioning concept diagram

The concept of versioning supports access for complete re-configurations for older files that are still in service to provide support for customers. Also, configurations can easily be rolled back should an earlier change prove to be incorrect or lead to problems in the developing system.

Manifests are XML files that define systems, programs and modules by simply linking to lower level manifests and files. In the above diagram, files are showed with bold lines and manifests with normal outline. Here, all links are dependency relationships. Thus, both modules M1.2 and M2.2 depend on file F1.3. If two modules have no dependency on each other, they are not linked.

When a newer version of a file has been pushed onto the repository for ex. F2.2(newer version of F2.1), the modular structure is modified updated by generating a new version for the manifest that initially referred to the older version. Now, manifest M2.2 refers to file F2.2 because of newer version introduction. The higher level manifest no has its own newer version M1.2 that refers to lower level manifest M2.2 and file F2.2.

6.6 CHECK-IN:

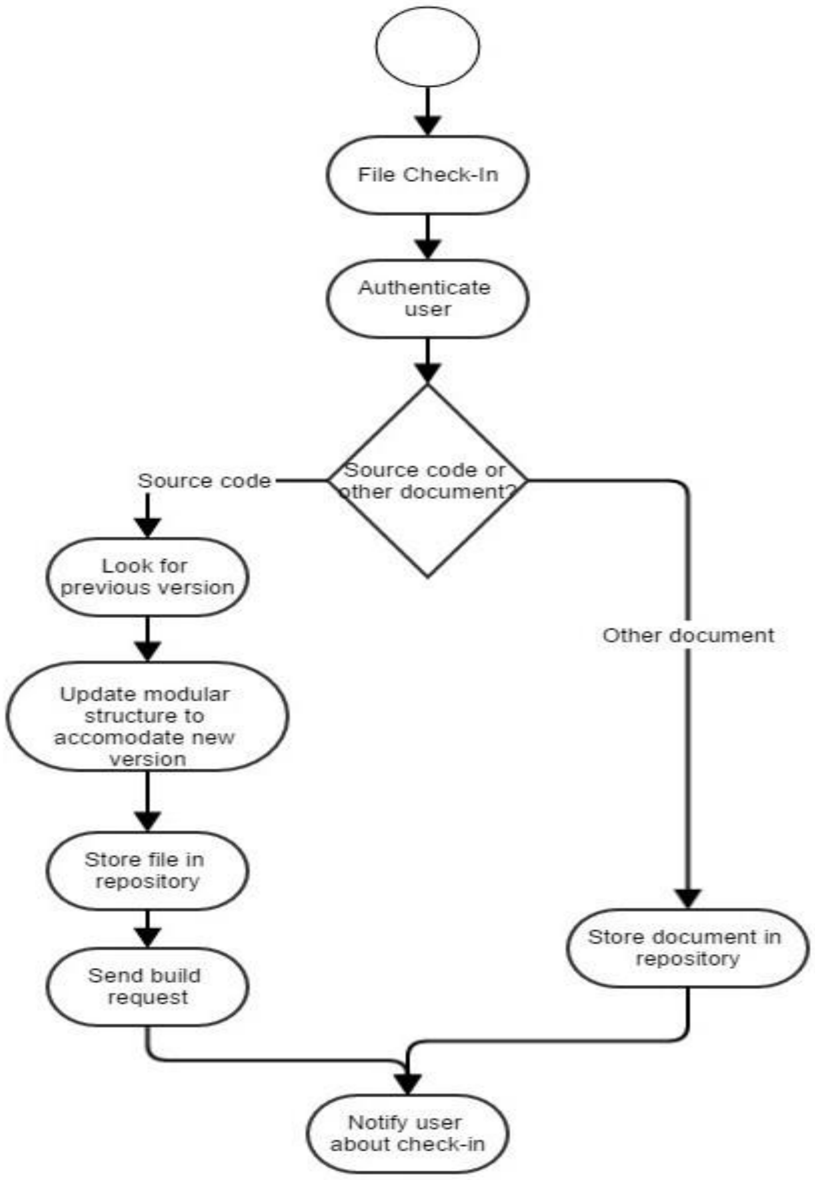

File Check-In
Authenticate user
Source code or other document?
Source code
Other document
Look for previous version
Update modular structure to accomodate new version
Store file in repository
Send build request
Store document in repository
Notify user about check-in

FIG. Check-In process

File check-in process involves the following stages,

- When check-in process is initiated, the user is authenticated to check if he/she is a registered project member.
- If the user is genuine, the server proceeds with check-in. Or else a notification is sent back stating authentication failure.
- If the submitted file is source code, the server looks for the older version of the code. If there is an older version (implying that source code submitted is new version) , the modular structure is modified to accommodate the newer version and the file is then stored on the repository server. The build server is then invoked and build image is generated. At the end of the process, the user is notified about check-in.
- If the file submitted is not source code, it is assigned an appropriate version number and stored on the repository server.

6.7 CHECK OUT:

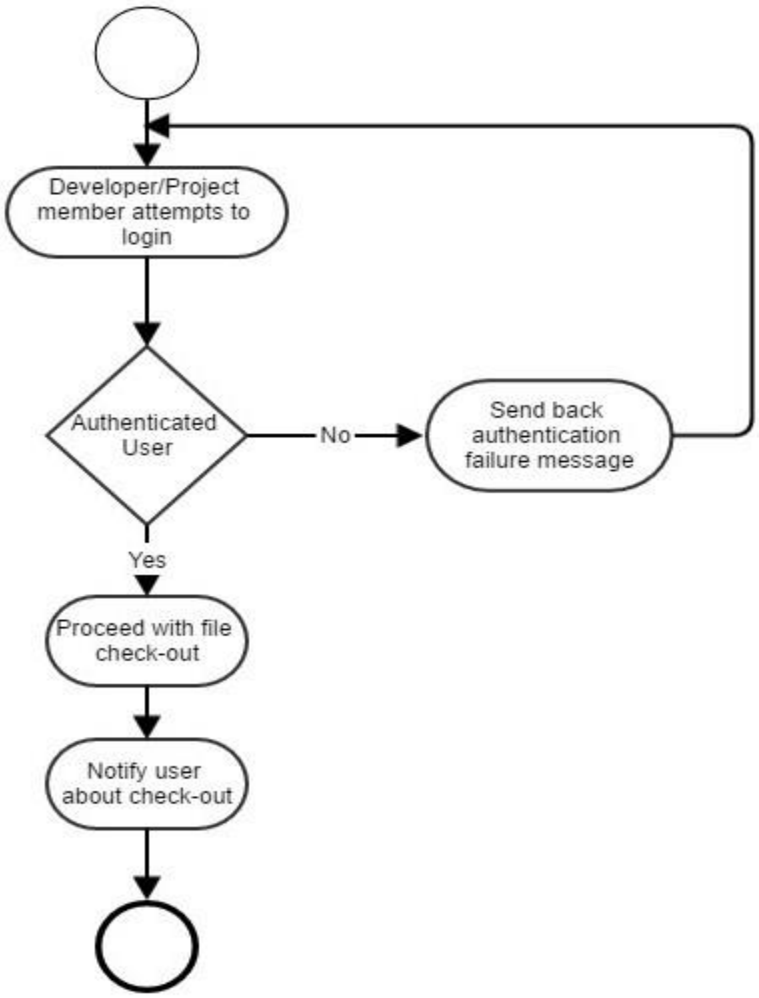

Developer/Project member attempts to login
Authenticated User
No
Send back authentication failure message
Yes
Proceed with file check-out
Notify user about check-out

FIG. Check-out process

The following activities take place when there is a check-out request,

- When there is a login attempt, the user/project member is verified for legitimacy, courtesy of ownership policies.
- If the user fails to be authenticated, an authentication failure message is sent back.

- If the authentication succeeds, the user is allowed to proceed with file check-out.
- The check-out result is finally displayed to the user at the end of the process.

6.8 AUTHENTICATION:

During file check-in or check-out process, it is absolutely necessary to verify participating users for their authenticity. This is to prevent misuse of source code and project related documents, thus ensuring that only those members part of the project have access to resources on the server.

If a user attempting to login fails to be verified, an authentication failure notification is sent back. If a verified user is attempting to perform file check-in or check-out process, he/she will be allowed to proceed. At the end of the process, the check-in or check-out status is communicated back.

6.9 PACKAGES:

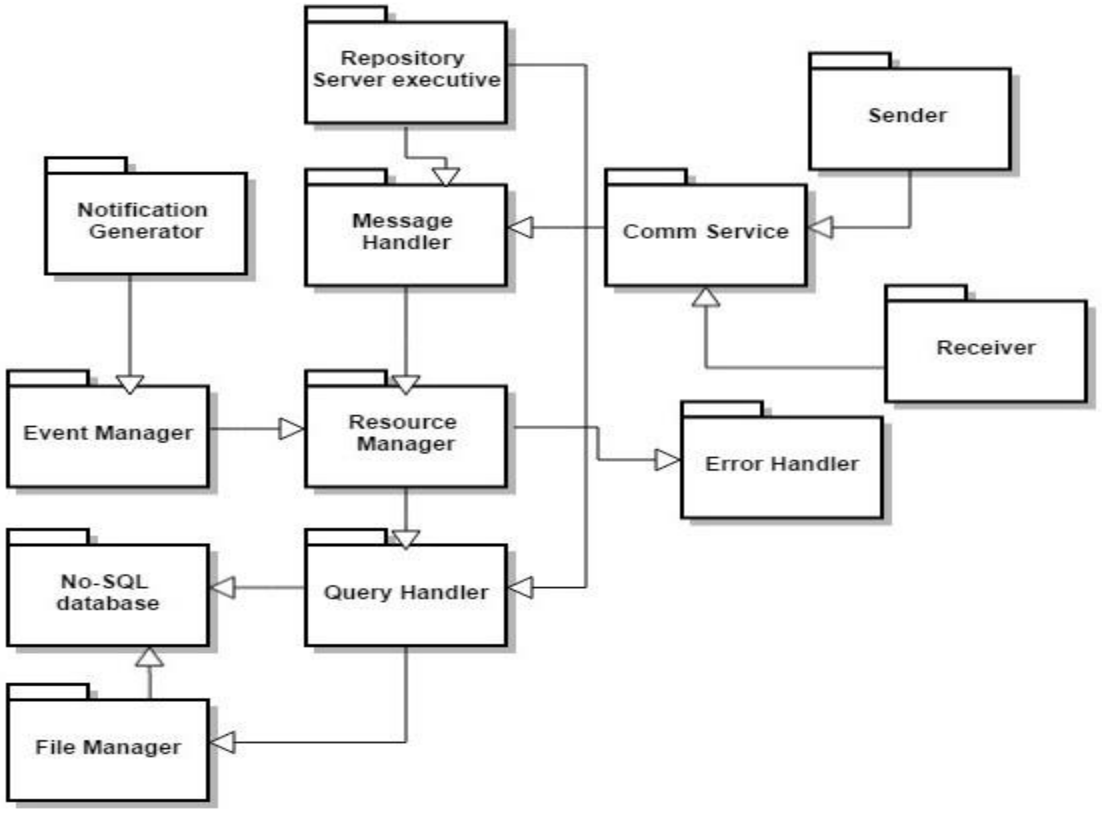


FIG. Package diagram of repository server

The different packages of the repository server and their functionalities are,

- **Repository Server Executive:** The server executive package is the main engine of the repository server. It is directly dependent on the Message Handler and Resource Manager packages to provide data storage services that will be used other servers of the software collaboration foundation.
- **Message Handler:** This package is responsible for handling any incoming messages from systems that it interacts with. When the build server routes a build request message, it is processed by the message handler.
- **Resource Manager:** The role of the resource manager is to allocate files to the repository database during check-in, retrieve files during check-out and maintain all project related documents.

- **Query Handler:** The query handler processes build requests from the build server. With the help of the file manager, it fetches files from the database and sends them over to the build server.
- **File Manager:** The responsibility of the File Manager package is to store and retrieve files from the local file storage system. The Resource manager utilizes the File Manager to store and retrieve files based on messages that arrive at the repository server in the form of file requests.
- **Event Manager:** The event manager monitors various events such as check-in and check-out and log them. It also sends out notifications to users who trigger events either through check-in or check-out.
- **Notification Generator:** This package is responsible for sending out notifications to project members after check-in, check-out processes. They are also used to send authentication related notifications to users when invalid users try to gain access to project data.
- **Error Handler:** It is used to notify project members of any possible errors that may arise while seeking service from the repository. for ex. When a user attempts to access a file that does not exist, an error indicating absence of file will be generated.
- **Comm Service:** The Comm service (or communication service) sets standards for communication between different service endpoints. It dictates rules for communication between different service endpoints.
- **Sender/Receiver:** The server consists of a sender and receiver for each communication endpoint that it interacts with. Sending and receiving processes are governed by the communication service.
- **No-SQL Database:** It is used to store all project related information such as source files, manifests, XML metadata files, concept documents and design diagrams.

6.10 USES:

The main uses of the Repository server are:

- It provides check-in and check-out functionalities to users.
- It supports authorization of users for access to project related files.
- It maintains dependency information in the form of metadata files.
- It is used to maintain software baseline.
- It supports versioning of files, query services and notifications.

6.11 USERS:

The main users of the repository server are:

- **Software Developers**
  Developers submit files, documents, test reports, test results and test summaries to the repository using the repository server's file check-in functionality. Also, they can extract files from the repository after authorization. This process of extracting files supported by the repository is check-out.
- **Software Architect**
  Software architects utilize the repository server to submit design documents, operational concept documents and to analyze critical project related information hosted on the server. Any changes in project design are updated and pushed onto the repository for users to access.
- **Project Manager/Team Lead**
  Project Manager and Team Leaders use the repository server to analyze file dependency information recorded as metadata. Also, they can also go through design and concept documents uploaded on the server. Test reports, logs and summaries give managers and team leaders a better understanding of achieved progress.
- **Testing Team**
  Test suites required for testing are developed and uploaded to the repository by the testing team. They can also use the repository to go through test results, test reports and summaries of various test iterations. By doing this, they can identify inconsistencies in testing and rectify issues. They can also alert developers of failures and unpredictable behavior in tests after analyzing test reports.

6.12 CRITICAL ISSUES:

- **Concurrent File Access**
  When the repository server receives file check-out requests or requests for test files, it may be possible that not one but many requests to the same files are made at the same time, leading to file conflicts. To avoid this problem, the repository should support file locking to ensure that only one access to the file can happen at a particular instant of time.
- **Server Breakdown**
  If the repository server maintaining files and documents checked-in by the user breaks down, or if the files get corrupted, project development takes a serious hit. Hence it is important to backup files onto redundant servers in a periodic manner. This way, even if there is a breakdown, flow of operation is not affected due to file duplication.

7. CLIENT

7.1 INTRODUCTION:

The client provides an interface between users and components of the software collaboration federation(SCF). It serves several groups of users, working from remote locations by providing user interfaces for system access. Through the graphical user interface, the user will be able to access tools and services offered as part of the federation. The client display is used to present charts, graphs, documents, tables, reports etc.

Through the user interface, the user can initiate file check-in and check-out processes after authentication, in tandem with the repository. Files can also be cached at the client end, thus reducing access time.

7.2 GOALS OF CLIENT SYSTEM:

- The client is responsible for creating and sending messages to the repository and test harness.
- File check-in and check-out processes are initiated from the client end.
- It also provides user authentication features to make sure only authorized members can gain access to services provided by the software collaboration foundation.
- It provides a user interface for viewing messages, reports and results from the test harness and repository server.
- It is also used to establish contact with other users or project members who may be located at remote locations across the globe.

7.3 ACTIVITIES:

The client subsystem supports a variety of activities in the federation. They are:

a) Authentication and Login:

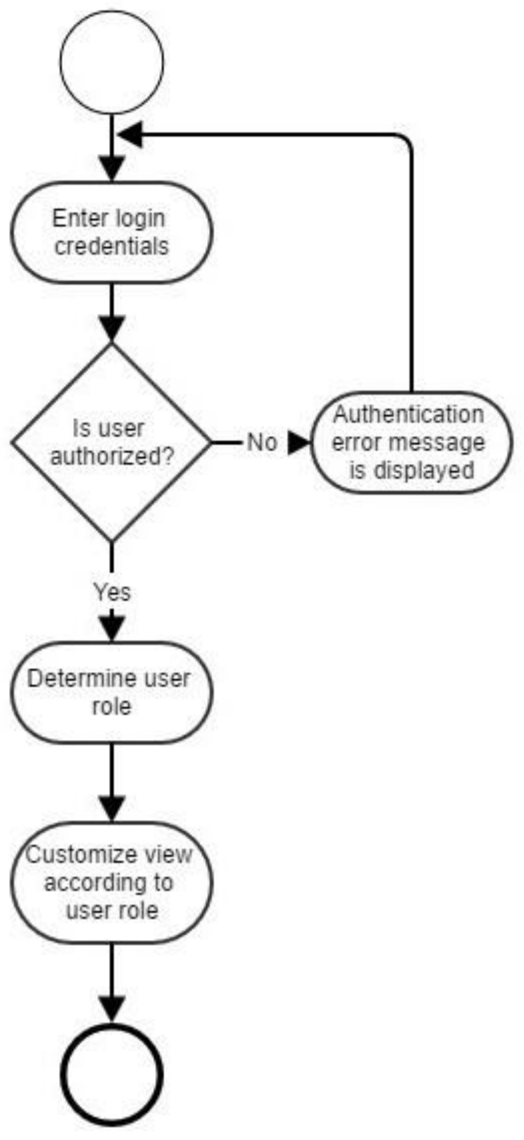

Enter login credentials
Is user authorized?
No
Authentication error message is displayed
Yes
Determine user role
Customize view according to user role

FIG. Authentication and login activity

The following steps are involved in the aforementioned activity,

- The user enters his/her login credentials using the user interface.

- These credentials match with the corresponding database entry if they are valid users, or else they are deemed invalid.
- If the user is not verified, an authentication message is displayed on the GUI and login credentials are sought again.
- If the user is verified, his role is determined and the user interface is morphed into a view corresponding to his role.

b) Client and Repository:

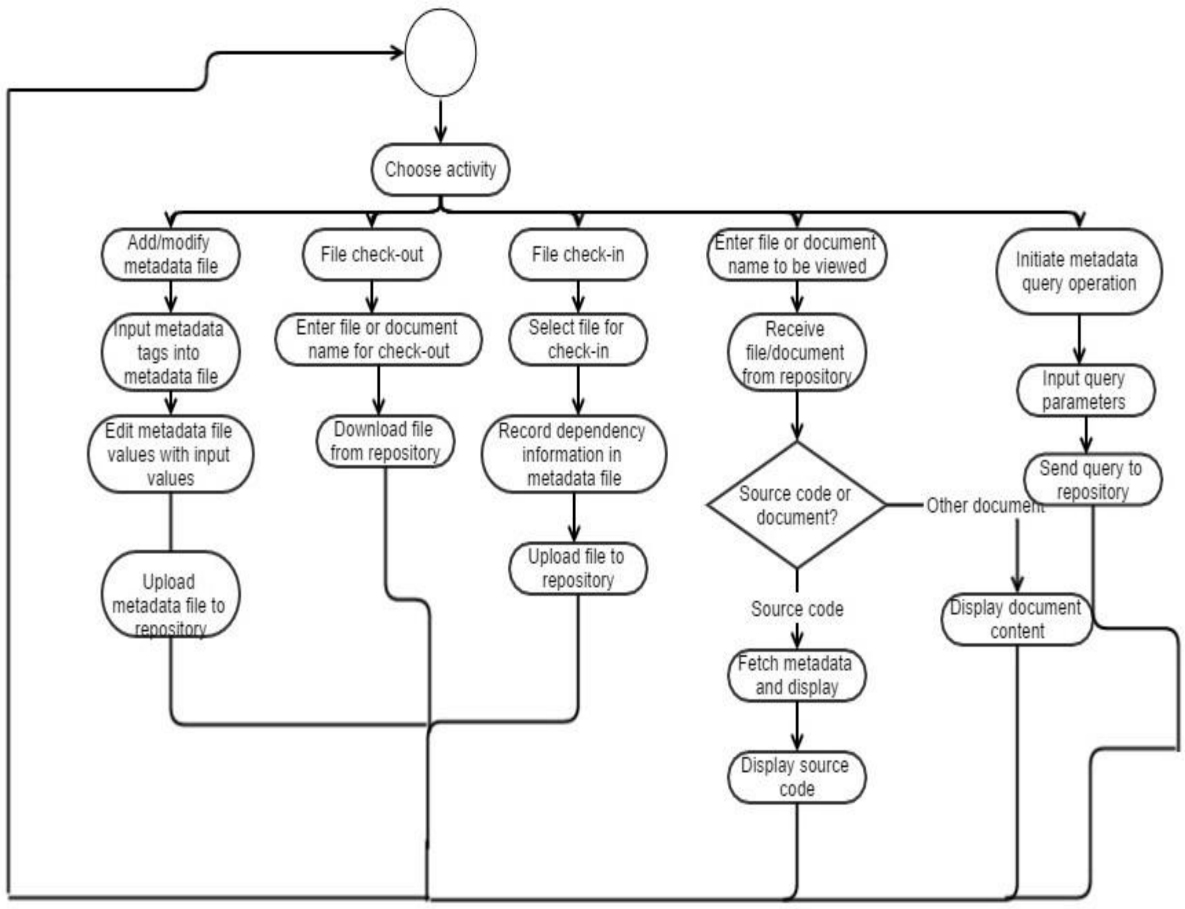

Choose activity
Add/modify metadata file
Input metadata tags into metadata file
Edit metadata file values with input values
Upload metadata file to repository
File check-out
Enter file or document name for check-out
Download file from repository
File check-in
Select file for check-in
Record dependency information in metadata file
Upload file to repository
Enter file or document name to be viewed
Receive file/document from repository
Source code or document?
Other document
Source code
Fetch metadata and display
Display source code
Display document content
Initiate metadata query operation
Input query parameters
Send query to repository

FIG. Client and Repository interaction activity

The client subsystem and repository can interact in many ways. All the possible activities are explained below,

- The user can select from a range of possible activities as per the circumstance.

- Using the client, metadata files for source code can be uploaded to the repository. Dependency information can be entered or edited in metadata files and pushed onto the repository.
- File check-in process is supported through the client subsystem. When the user wants to initiate check-in process, he is authenticated at the client end. If the user in discussion is an authorized user, he/she is prompted to select the file intended for check-in. At the same time, the user is required to record dependency information in the form of metadata. The files are then uploaded to the repository.
- The client also supports file check-out process. During check-out, the user is verified for authenticity. If the user is authorized, he is prompted to enter the name of the file or document for download. The file/document is then downloaded from the repository.
- The user can also view files and documents on the client display screen. The user enters the details of the file or document to be displayed on the GUI screen. The file is fetched from the repository. If the file in question is source code, the metadata information of the file is displayed along with the source code. This provides file dependency information to the user. If the file is a document other than source code, it is displayed to the user on the client display screen.
- The user can send queries for metadata files to the repository. These files contain dependency information between packages and modules. When the user initiates a metadata query operation, he is prompted to enter query parameters specific to a particular metadata file. The request is then framed and sent to the repository.

7.4 PACKAGES:

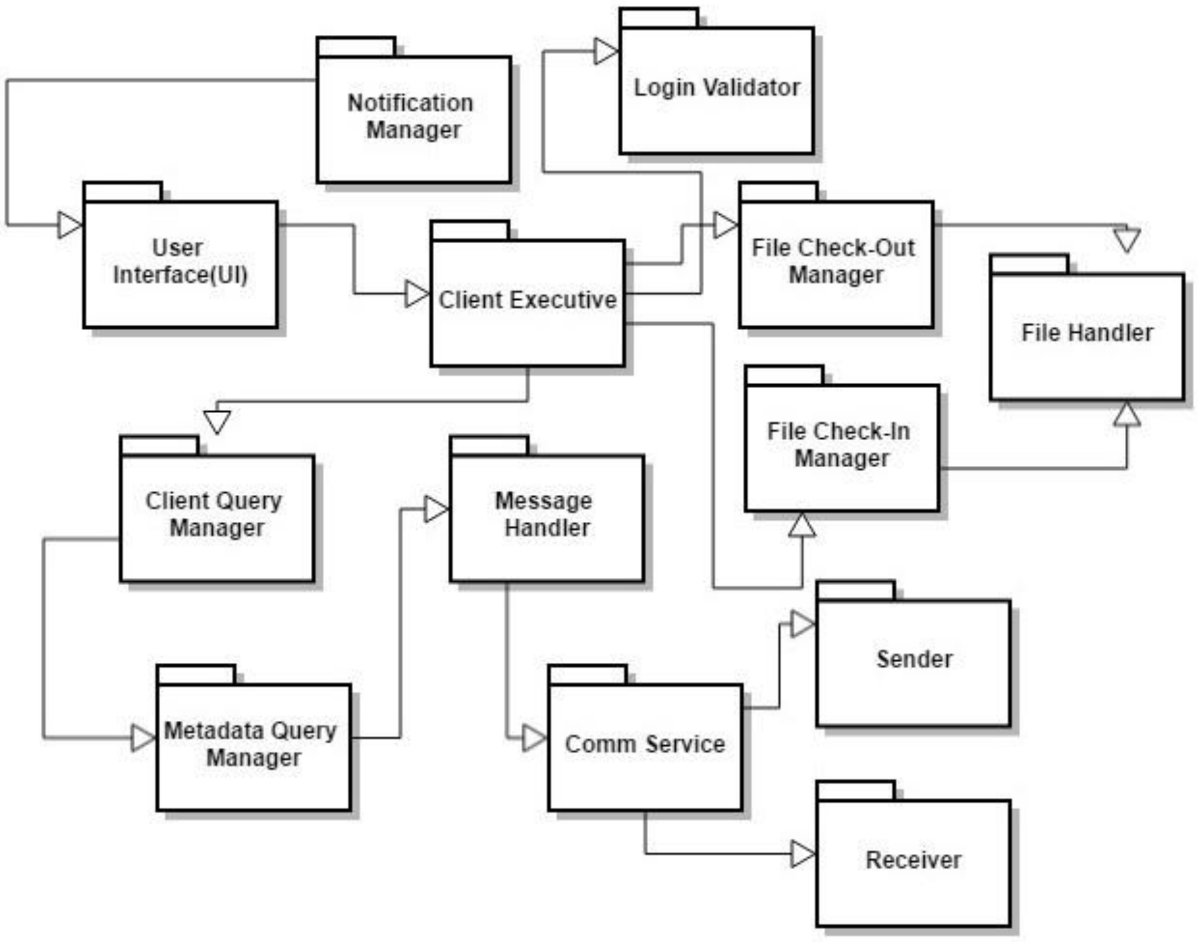


FIG. Client package diagram

The packages that make up the client subsystem are,

- **User Interface:** The user interacts with the client subsystem through the user interface. Any interaction from the user is communicated to the client executive through the user interface.
- **Client Executive:** This is the engine behind client operation. All interactions from the user are handled by the client executive. It controls the operation of all other packages directly or indirectly.

- **Client Query Manager:** This package is responsible for handling queries from users. When the user wants dependency information in the form of metadata files, the query is routed using the client query manager.
- **Metadata Query Manager:** This package is responsible for handling all requests for metadata files by the user. This package works in association with the client query manager. For querying metadata information, the user will have to enter one or more categories and metadata tags whose values will be returned by the server after successful execution of the query.
- **File Check-In Manager:** This package will be responsible for processing file check-in requests initiated by the user. This happens when the user intends to push files into the repository.
- **File Check-Out Manager:** This package will be responsible for processing file check-out requests initiated by the user. This happens when the user intends to download files from the repository. When files are downloaded from the repository, they are displayed to the user on the user interface(UI). If the requested file is a source code, metadata information associated with it is also fetched and displayed.
- **File Handler:** This package is responsible for fetching and saving files into the local file system.
- **Message Handler:** It is used to establish communication channels with different interacting subsystems. i.e between client and repository, client and test harness and client and collaboration server subsystem.
- **Comm Service:** The Comm service (or communication service) sets standards for communication between different service endpoints. It dictates rules for communication between different service endpoints.
- **Sender/Receiver:** The server consists of a sender and receiver for each communication endpoint that it interacts with. Sending and receiving processes are governed by the communication service.

7.5 USER INTERFACES:

The client user interacts with the federation through the user interface. The UI views for different client functionalities are shown below,

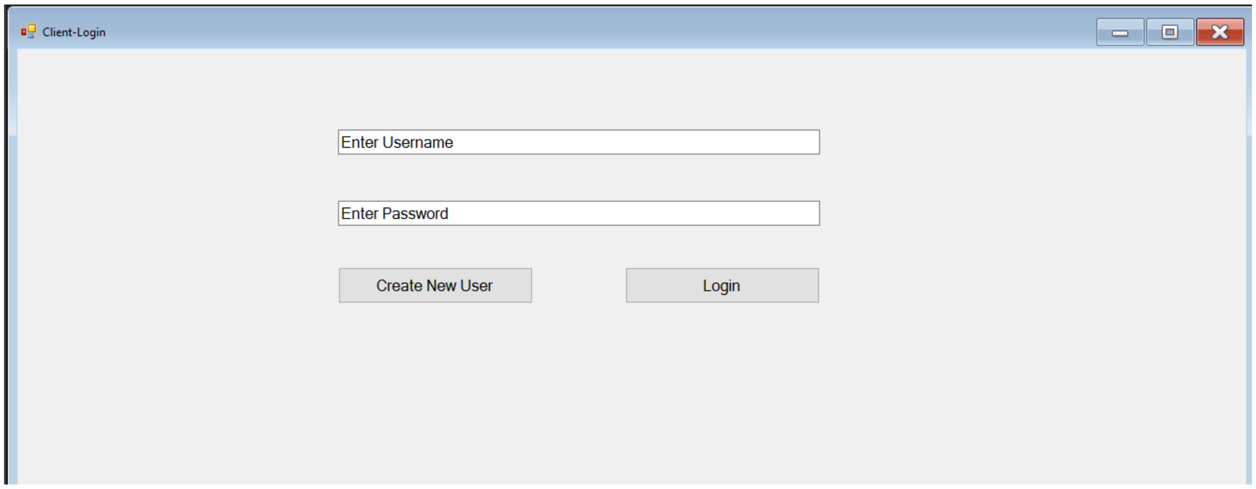


FIG. Login window

The above diagram represents the logon screen for the client system. The login credentials entered by the user are checked for equality through several database entries. If the user is registered, he gains access into the system. Or else, an authentication error notification pops up.

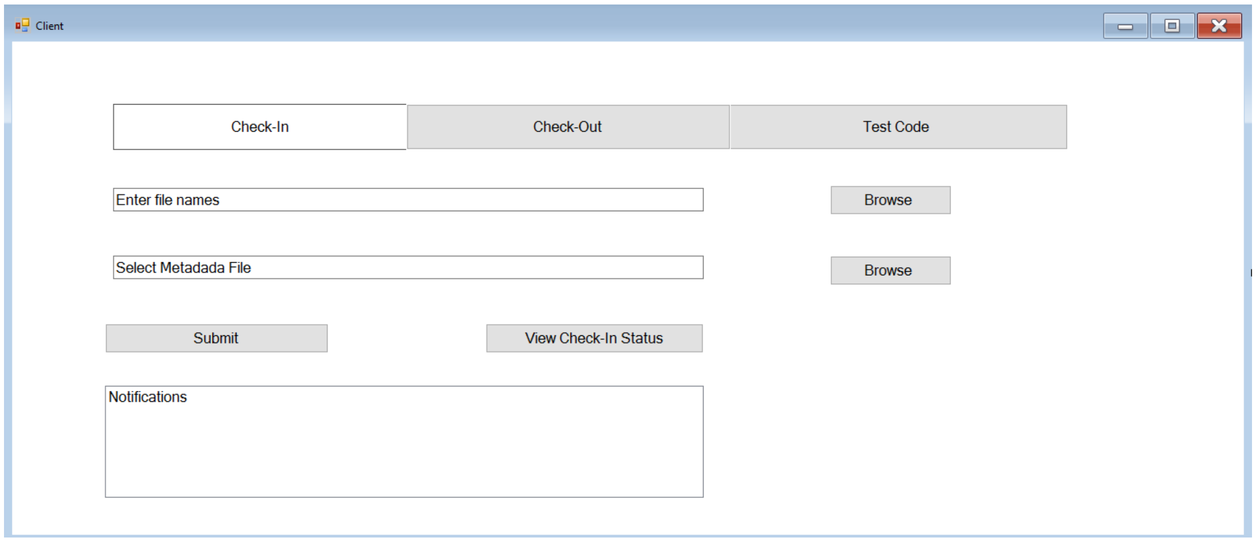
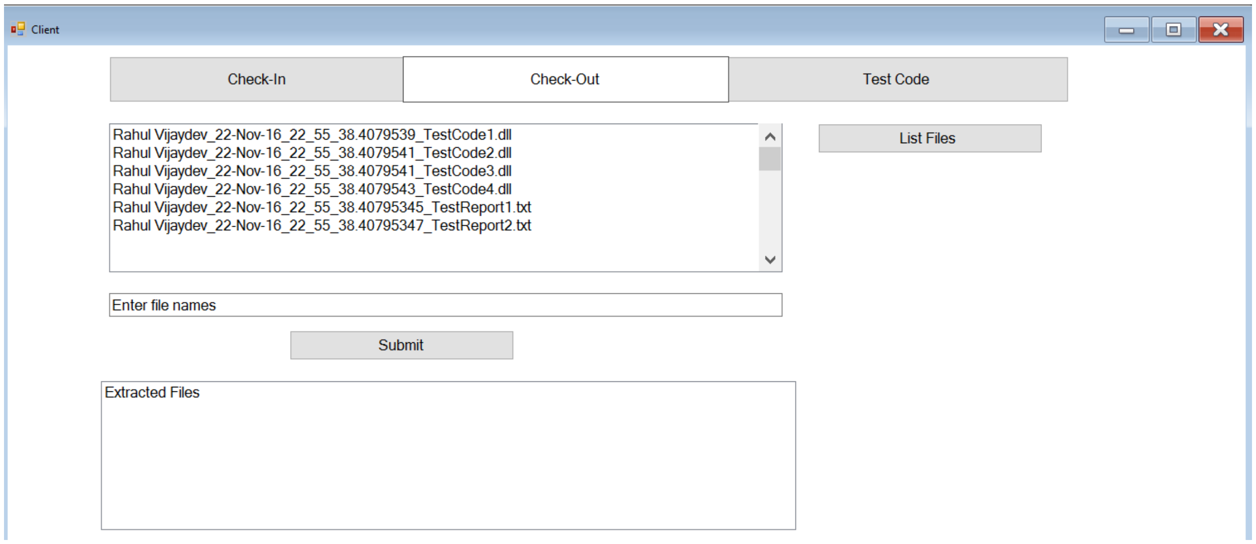


FIG. Check-In functionality

The above figure represents shows the check-in functionality window for the client. The client user is prompted to enter the files that he wishes to check in. Along with these files, associated metadata information is also uploaded. The client can view check-in status and receive notifications on the window.



```
Rahul Vijaydev_22-Nov-16_22_55_38.4079539_TestCode1.dll
Rahul Vijaydev_22-Nov-16_22_55_38.4079541_TestCode2.dll
Rahul Vijaydev_22-Nov-16_22_55_38.4079541_TestCode3.dll
Rahul Vijaydev_22-Nov-16_22_55_38.4079543_TestCode4.dll
Rahul Vijaydev_22-Nov-16_22_55_38.40795345_TestReport1.txt
Rahul Vijaydev_22-Nov-16_22_55_38.40795347_TestReport2.txt
```

FIG. Check-out functionality

The above figure shows the check-out functionality for the client user. The user has access to the list of files that he wishes to extract from the repository. On entering the files that he wishes to extract, he initiates check-out process. The files extracted from the repository are displayed in the ‘Extracted files’ box.

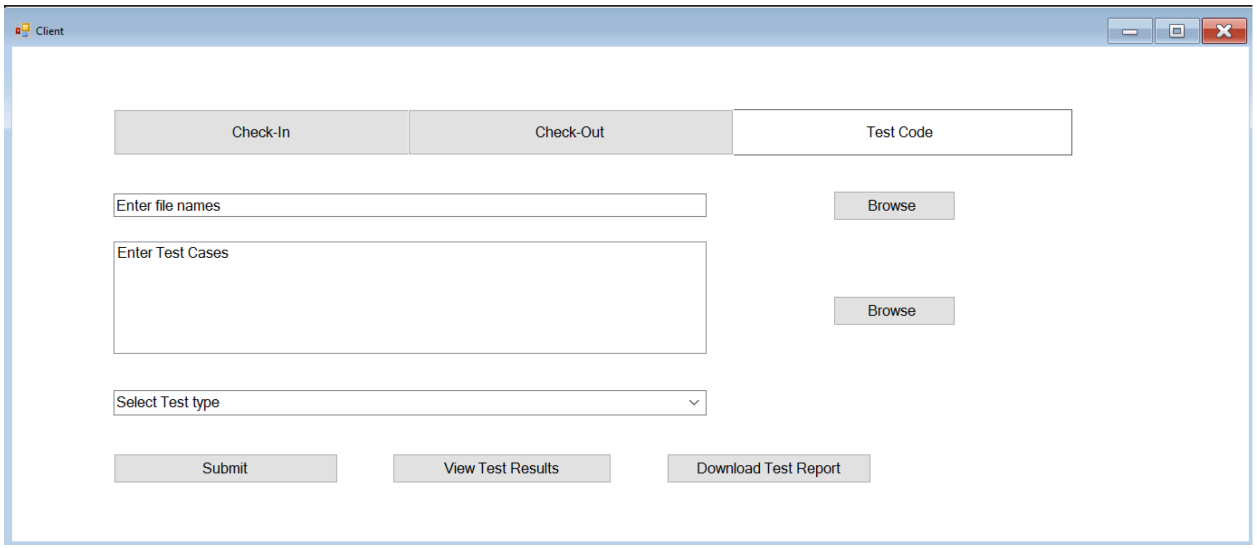


FIG. Test Code window

The above figure represents the test code functionality for the client user. The user enters the names of test files to be tested, including test suites. The type of test i.e Integration testing, Regression testing, Unit testing etc. can be selected and the test process can be initiated.

7.6 USE CASES:

The client subsystem is used by practically every member involved in the development process. The uses of the client subsystem and their users are,

- **Software Developers:**
  - **a)** Software developers use client for file check-in and check-out processes.
  - **b)** The client is also used to verify users and hence it serves as an authentication tool for developers.
  - **c)** Developers can also provide metadata information of source code that they were responsible for development. This information can be utilized by other developers to obtain dependency information. As a result, package integration becomes a fairly easy job.
  - **d)** Developers also use the client display to view files submitted by other developers. This way, they can obtain a better understanding of the code structure.

e) Developers use the client to submit test requests to the test harness.

- **Team Leaders/Managers:**
  a) Team leaders and Managers can track real-time work progress through the client subsystem. This can be achieved by monitoring and viewing progress reports.
  b) They can view results of tests returned by the test harness after testing.
  c) They can create test reports and upload to the repository for further analysis.
  d) They can organize and send meeting requests to project members.
  e) Monitor throughput using work progress reports.
- **Software Architects:**
  a) Schedule meetings and send out meeting requests.
  b) Monitor reports and notifications.
  c) To comprehend project structure, using package dependency information and go through project documentation to gain better understanding of modular interaction.
- **Testing Team:**
  a) To submit test related files like drivers and source code for testing.
  b) Extract test results and logs from the repository.
  c) Run tests using the test harness and view results, to ensure consistency in development and testing.
  d) To acknowledge meeting requests and notifications from various subsystems in the federation, during client-subsystem interaction.

7.7 CRITICAL ISSUES:

- **Load on Servers:**
  The software collaboration federation (SCF) consists of thousands of developers and client systems concurrently working together. These client systems constantly interact with servers for work collaboration, testing, building executables and build images, and storing all project related information. There will be huge number of interactions between client systems and servers, and it is important that servers have the necessary load handling capacity to attend to all client requests with much delay. Since these interactions are part of a continuous process, care has to be taken to make sure that clients receive services within short periods of time. Also, redundant servers need to be equipped so that there are no operation stalls when a server goes down.

- **File Format issues:** It is absolutely important for users to maintain consistency in file formats that are uploaded to or accessed from servers. If the user is responsible for uploading files with undefined formats, operational inconsistencies may occur. Hence, team leaders, managers and higher authorities must define accepted file formats and all users or project members must adhere to the rules.

8. TEST HARNESS

8.1 INTRODUCTION:

Concept:

The test harness is used for automating tests. It is the busiest system on the software collaboration federation and is responsible for continuous test and integration (CTAI) of code into the software baseline.

Different clients in the software collaboration foundation host windows communication (WCF) endpoints for enforcing communication service. They establish channels to the test harness service endpoint in order to forward test requests.

When a test request message has been forwarded to the test harness, it parses the message and extracts the XML test request file that was initially packed along with the message. For each test request, a new thread is spawned from the main test harness/test executive thread, and the test requests are forwarded to each of the threads. From within each of the child threads, parsing of XML test requests is carried out, thus revealing content relevant to that test. This activity is followed by fetching build images required for the specific test from the build server and writing them into files in unique directories corresponding to the author that generated the test in the first place. Once this operation is done, the child thread initiates the creation of a child appdomain and injects the loader to which a callback reference is passed along with the generated parse results and directory information from where test related files can be sought. The loader initiates test execution and writes test results into log files in the unique directory corresponding to the test request. Also, the loader uses the callback reference to send back test results that can be accessed by its parent thread holding a reference to the same callback object. These results are then built into a form(message) that can be transmitted over the communication channel to the client that updates its graphical window dynamically to project these results.

Objectives:

The objectives of the Test Harness are:

- To enforce continuous integration of code into the software baseline.
- To automate the process of testing code.
- To implement efficient message passing communication between different service endpoints.

- To enforce concurrent test execution using simple and efficient multi-threading models and inter-thread communication.

8.2 ADVANTAGES:

The test harness simplifies the life of the developer and tester. Some of the advantages of using the Test Harness are:

- A drastic improvement in the quality of the software product.
- Simplification of testing due to the automated nature of the tool, thus increasing productivity.
- Scheduling tests is easier.
- The ability to run tests subsequently and keeping the tool occupied at all times.
- Software states can be controlled and tested independently.
- Simulating different test cases on the code that would otherwise be difficult to implement if test automation didn't exist.
- Carrying out Stress and Performance tests as well as Regression tests becomes easier with the tool.
- Establishing isolation between different test runs through .NET AppDomains.
- Debugging code and finding out root causes for bugs are easy since any inconsistency in one isolation layer does not affect processing in another.

8.3 STRUCTURE:

- Building test request messages and forwarding them to the test harness.
- Dequeue test requests and parse structured message to generate test initiator and test details.
- Create a child thread for each test request and pass information relevant to the test to the child thread.
- Parse XML test request and obtain test-specific file names and their locations.
- Create unique directory corresponding to each test initiator(author) and download necessary files into it from the repository.
- Create callback object for referencing after test result generation.
- Create a child appdomain, inject loader and pass unique directory location and callback reference to the loader instance.
- Execute tests, save log results into files on the unique directory and forward test results back to the child thread, which in turn uses the service on the client to post results.

- Ultimately display test results or log data on the client window.

BASIC STRUCTURE

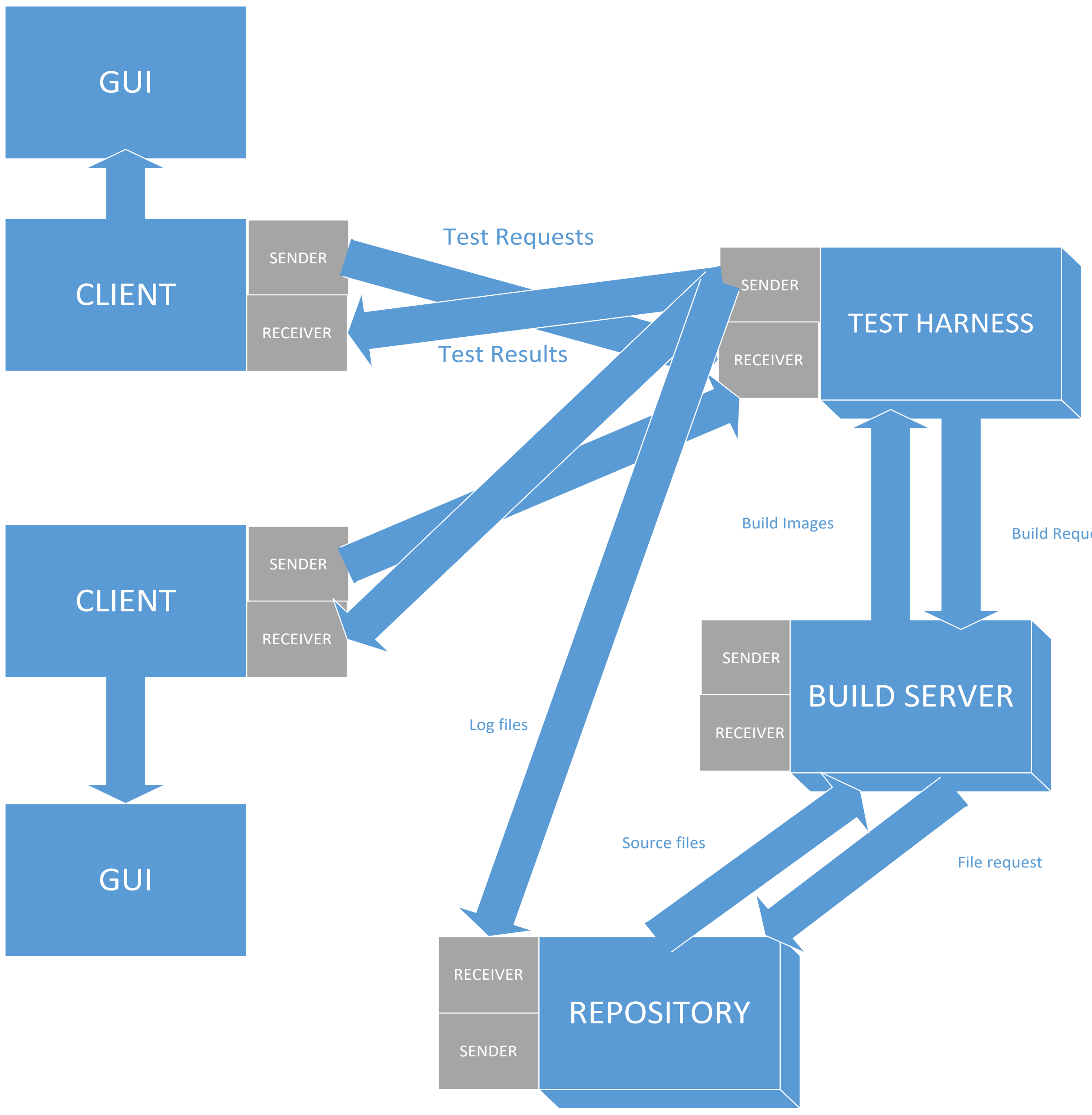


FIG. Basic structure of test harness

8.4 ACTIVITIES:

The Test Harness application, being a full-fledged integrated testing tool performs a series of activities that are summarized as follows:

The Test Harness serves as the hub for all testing operations. As a full-fledged integrated testing tool, it performs a series of activities that are summarized as follows:

- The Test harness implements data and operational contracts as part of the entire service contract.
- The service contract defines how communication endpoints can be used to establish contact with, from other modules intending to utilize the service.
- It then hosts the Windows communication foundation service adhering to its contract. The contract is then exposed to other modules that reach out for the service.
- The test harness continuously monitors its queue for test requests from the client module. If the queue is loaded with test requests, the test harness dequeues them and parses the test request that would have initially taken the form of a message when built on the client module.
- The XML test request, test initiator (author) and other test details corresponding to each of the test requests is extracted after the parsing process.
- For each test request, the test harness thread spawns individual child threads that carry the operation load.
- The XML test files corresponding to each of the test requests are passed onto their respective child threads and parsed to obtain concrete details about the test.
- A callback object is created in the child thread.
- Each of the child threads are responsible for creating individual child appdomains on which the tests are carried out.
- The child thread creates a unique directory for the test and downloads build images from the build server into the directory.
- Once the loader is injected into the child appdomain, the callback object reference is passed to it along with unique directory and relevant test information.
- The loader orchestrates testing in the child appdomain and passes back test results using the callback object reference initially shared with by the child thread.
- The child thread reads test results using its reference to the same callback object.
- Logs are forwarded to the repository server.

- Test results are then relayed over the WCF communication channel back to the client in the form of messages, which in turn are displayed to the user on the Graphical User Interface.

The Test harness activity diagram is presented below:

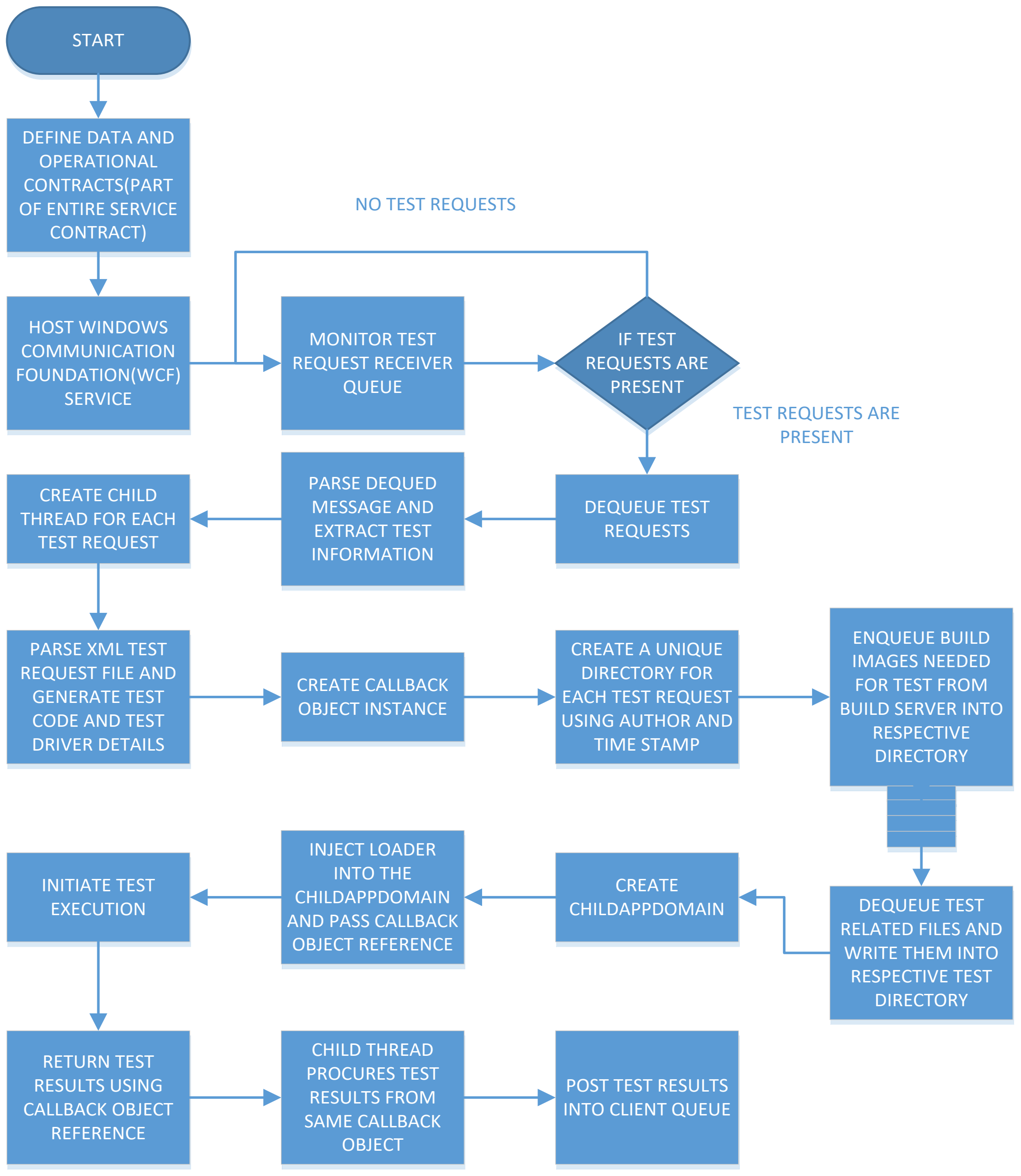

START
DEFINE DATA AND OPERATIONAL CONTRACTS(PART OF ENTIRE SERVICE CONTRACT)
NO TEST REQUESTS
HOST WINDOWS COMMUNICATION FOUNDATION(WCF) SERVICE
MONITOR TEST REQUEST RECEIVER QUEUE
IF TEST REQUESTS ARE PRESENT
TEST REQUESTS ARE PRESENT
CREATE CHILD THREAD FOR EACH TEST REQUEST
PARSE DEQUED MESSAGE AND EXTRACT TEST INFORMATION
DEQUEUE TEST REQUESTS
PARSE XML TEST REQUEST FILE AND GENERATE TEST CODE AND TEST DRIVER DETAILS
CREATE CALLBACK OBJECT INSTANCE
CREATE A UNIQUE DIRECTORY FOR EACH TEST REQUEST USING AUTHOR AND TIME STAMP
ENQUEUE BUILD IMAGES NEEDED FOR TEST FROM BUILD SERVER INTO RESPECTIVE DIRECTORY
INITIATE TEST EXECUTION
INJECT LOADER INTO THE CHILDAPPDOMAIN AND PASS CALLBACK OBJECT REFERENCE
CREATE CHILDAPPDOMAIN
DEQUEUE TEST RELATED FILES AND WRITE THEM INTO RESPECTIVE TEST DIRECTORY
RETURN TEST RESULTS USING CALLBACK OBJECT REFERENCE
CHILD THREAD PROCURES TEST RESULTS FROM SAME CALLBACK OBJECT
POST TEST RESULTS INTO CLIENT QUEUE

FIG. Activity diagram of test harness

Figure below shoes the operational flow diagram for the test harness module:

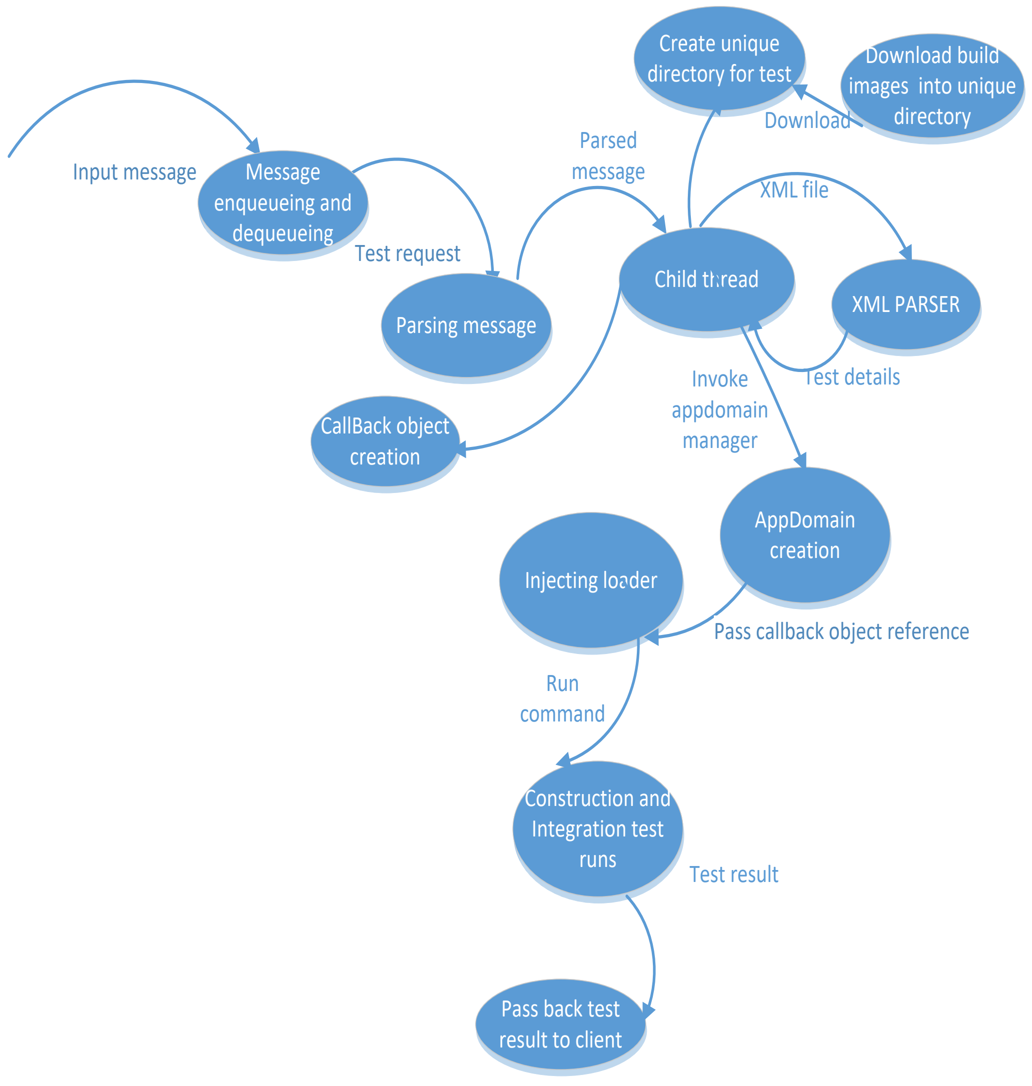


FIG. Operational flow diagram of test harness

8.5 PACKAGES:

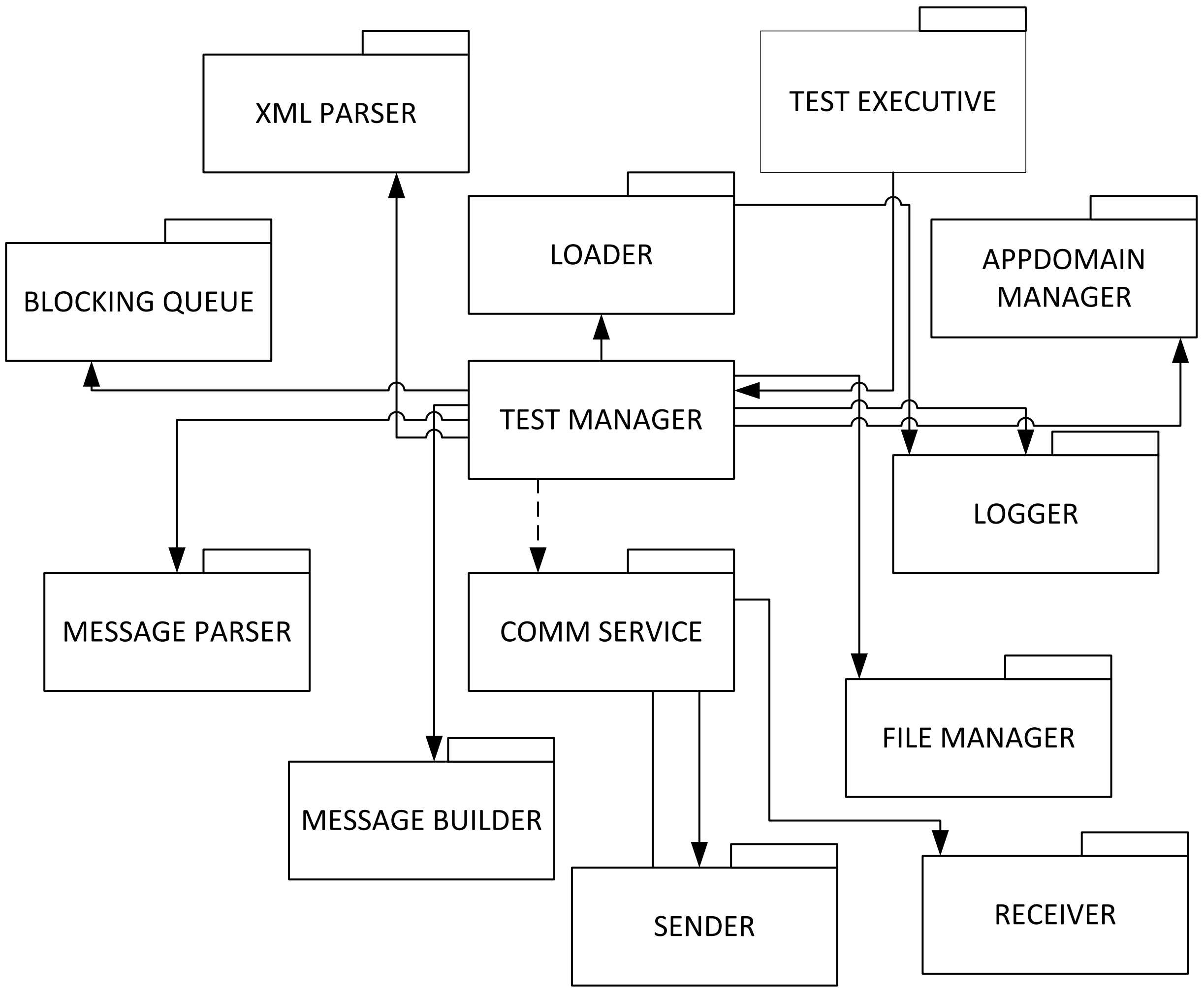


FIG. Package diagram of test harness

The above figure shows the modular structure of the Test Harness. The entire operation of the Test Harness can be categorized into a set of interdependent functional packages. They are:

1. **Test Manager**: The test manager serves as the engine of the Test Harness. It controls the operation of all other packages and schedules the entire testing process. Each other package in the test harness communicates with the Test Manager in some way or the other. Hence, the Test Manager acts as the main entry point to the Test Harness. It performs the following operations:

- It hosts the WCF service and exposes contract details to modules seeking hosted service.
- It is also responsible for creating proxy objects needed to connect to the client and the repository.
- The test manager dequeues test request messages relayed by the client.
- It uses the message parser and parses each of the messages, thus extracting test details initially packed in the message.
- For each test request, it spawns a child thread and passes it the parsed message.
- The child threads are responsible for XML parsing and child appdomain creation. Here each test iteration runs in its own isolation layer.

2. **Test Executive:** The test executive demonstrates the working of the test harness. The flow of operation can initially be observed by invoking the test executive.
3. **Blocking Queue**: This package implements a generic blocking queue and demonstrates communication between two threads using an instance of the queue. If the queue is empty when a reader attempts to dequeue an item, then the reader will block until the writing thread enqueues an item.
4. **Message Parser:** Any queued message from either the client or the repository is dequeued and parsed into its constituent units. Messages from the client contain author, time stamp and test request(XML) information. These individual units are unpacked using the parser. Similarly, the files required for testing are fetched from the repository in the form of a message and parsed to reveal test code and test driver dynamic linked libraries.
5. **Xml parser**: The intention of this package is to parse test requests that arrive in the form of XML files after message parsing. The parser segregates data in the file and extracts developer information, test code and test drivers required for testing. The parser is implemented in every child thread corresponding to a test request.
6. **File Manager**: Test files i.e test driver and test code dynamic linked libraries needed for testing are stored on the repository. The File Manager module performs a recursive search operation to check the existence of required files for testing. It displays all files in the directory and runs an availability check to see that the files requested for testing is present. If they are present, the files are fetched and written to test-unique directories. If not, an exception arises suggesting absence of requested file/files.
7. **Appdomain manager**: The AppDomain manager initiates the creation of an AppDomain corresponding to a given test request when it is signaled to do so by a child thread spawned by the main Test Harness thread. It keeps a track of all AppDomains created

and efficiently manages resources among various AppDomains if multiple test requests are being executed. Here in our project, we create multiple child appdomains with each appdomain corresponding to one test request. Each appdomain runs in its own thread of execution.

8. **Loader**: The loader unit is responsible for initiating test execution on an appdomain. It is a localized unit that sets the testing process within an AppDomain in motion by invoking appropriate driver methods that route execution control onto respective test code methods.
9. **Message Builder**: The WCF data contract specifies the type of information that needs to be sent over the communication channel. We use messages for transferring information between different modules. Hence, any information that needs to be transmitted to another communication endpoint is sent in the form of message objects. The message builder frames information into a message that is suitable for channeling. Test results are relayed over the communication channel back to the client in the form of messages.
10. **Logger:** The logger module is used to store test logs in files after testing. Log files contain information such as developer signature, time-stamp indicating the time at which the test iteration was carried out, consequences of different test cases, errors that occurred in testing, and exceptions in the execution flow. Test logs can also be persisted to a database and accessed at any point of time by making separate database queries
11. **Comm Service:** The Comm service (or communication service) sets standards for communication between different service endpoints. It dictates rules for communication between different service endpoints.
12. **Sender/Receiver:** The server consists of a sender and receiver for each communication endpoint that it interacts with. Sending and receiving processes are governed by the communication service.

8.6 INTERACTIONS:

a) **Client and Test Harness interaction:** Two types of communication are possible between the client system and the test harness. They are,

- Client to Test Harness: Test requests are submitted to the test harness through the client system. The data contract of the windows communication foundation service defines the format of test request. Requests are normally sent in the form of messages.
- Test Harness to Client: After submission of the test request, the test harness parses the message and reveals the content of the test request. It processes the request and runs the appropriate tests. The test results are then forwarded back to the client and displayed on the client display window(GUI).

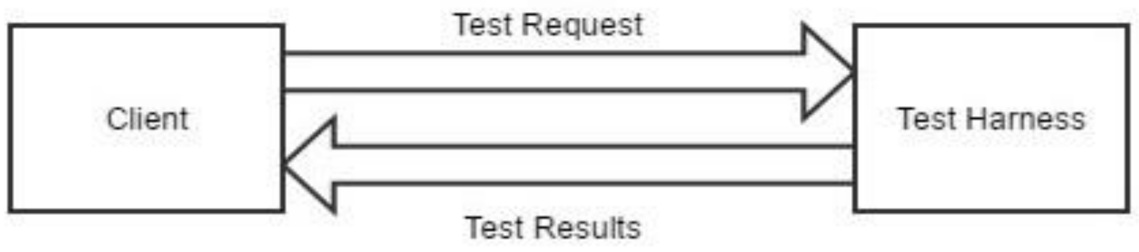


FIG. Client and Test Harness Interaction

b) **Test Harness and Build Server:** The test harness, upon receiving the test request from the client will know the list of files needed for testing, after processing it. The list of files obtained after processing the request are sent over to the build server. If the requested files are present in the build server cache, the build images are quickly sent back to the test harness. If these files are not present in the cache, the build server fetches them from the repository server, generates build/execution images and then sends them to the test harness.

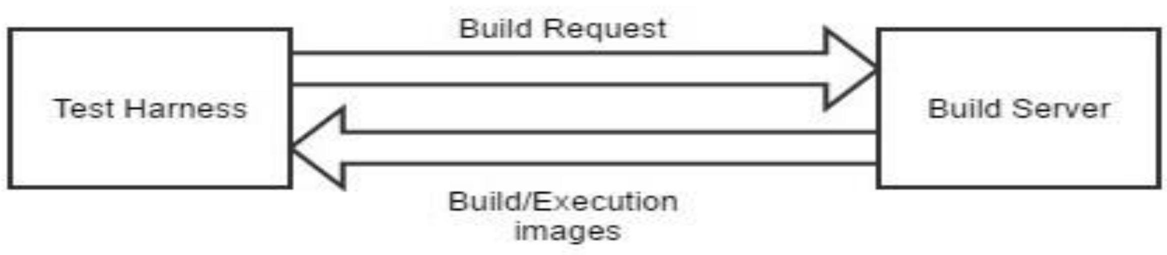


FIG. Test Harness and Build Server Interaction

c) **Test Harness and Repository interaction:** The test harness posts log results to the repository after testing. In addition to logs, test results are also forwarded to the repository.

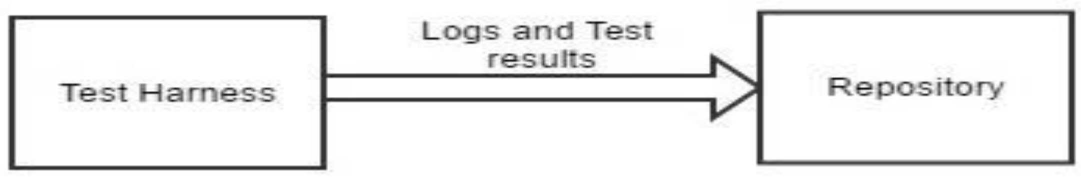


FIG. Test Harness and Repository interaction

8.7 USE CASES

The primary users of the test harness are developers, quality analyzers or testers, program managers and software architects. Below, a brief description about each of the users are documented.

- **Developers**

  Each developer is entitled to test his code to make sure that the module in development blends in with the software baseline. The developer can perform 3 types of testing and they are Construction testing, Unit testing and Integration Testing. Construction testing requires testing each package individually using test stubs. The developer uses the test harness mainly to perform Integration testing. Integration testing is carried out to substantiate various interfaces and interactions between modules. A full-fledged multi-threaded communication model is employed, to run several dependence or integration tests concurrently. Inherently, the test harness provides an excellent way for developers to get their work done effortlessly.

- **Quality Analyzers(QA) or Testers**

  In addition to developers, quality analyzers perform another round of testing to provide an unbiased verdict on the tested code and thus ensuring that the quality of code hasn't been diluted. QA's are usually more effective at finding issues in code since they analyze it from a neutral perspective. QA's come into picture during Integration testing, Regression testing, and Qualification Testing. In Integration testing, the QA pulls up the developer's code and its dependencies, and performs one more round of testing to avoid ambiguities in the code's objectives. Regression testing has similar implementation rules as that of Integration testing. Here, the QA tests the software on platforms other than the one used for development so that migration of software applications from one platform to another does not result in aberrations in application behavior. In Qualitative testing, the QA checks to see that all requirements stated by the client are met. This involves explicit examination and demonstration of operational details to the client.

- **Program Managers**

  The role of the Program Managers is to communicate specification details to the developers. They scrutinize every step that the developer makes, and are primarily involved during qualification testing along with the quality analysis team. They will also make sure that critical issues that hinder the working of the product are addressed and resolved.

- **Software Architect**

Software architects carry out performance and stress testing on applications. Performance testing mainly deals with time responsiveness and CPU usage factors whereas Stress testing is carried out to inspect input handling abilities when unusual loads and resource constraints could prospectively occur. He ultimately seeks for scalability and consistencies in application behavior when subjected to heavy workloads. Hence, the software architect examines application behavior in extreme operating conditions as part of Stress and Performance testing on the Test Harness.

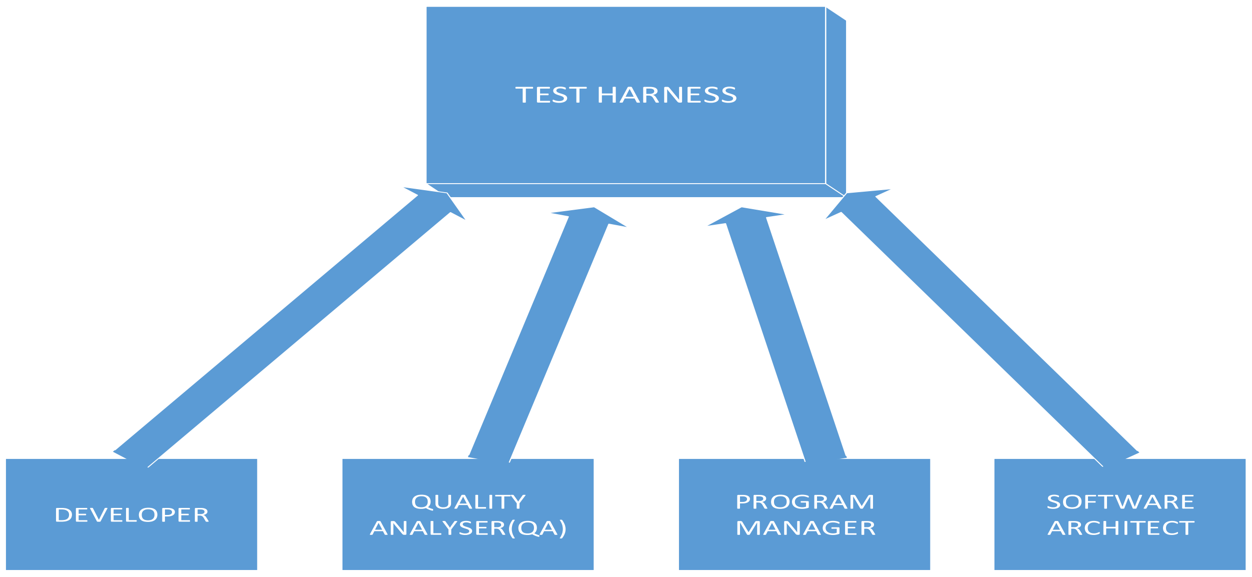


Fig 2. Users of the Test Harness

8.8 CRITICAL ISSUES

The software development process requires a keen inspection of critical issues that need to be resolved, without which there would be major shortcomings in the rendered application. This section talks about some of the key issues that require attention while building the Test Harness. They are:

- **Ease of use:** This is by far the most important issue that needs to be considered while building any application. The aim of a software application is to make people's lives easier. Hence it is imperative that the software is as easy to use as possible. As quoted in the ISO 9241 standard, ease of use represents the degree to which a product can be used by specific users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. It is a measure of the simplicity with which an application's functionalities can be availed, and this embeds design and implementation details right from the User Interface to the low-level system setup. Testing requires importing one's code, it's dependencies and test drivers from repository servers as Dynamic Linked Libraries(DLL) and running them in isolation layers. Although this internal functioning of the tool can be complex, it is the responsibility of developers to make the tool be as simple to use as possible. In project 2 we achieve this by simply using the command line to run the Test Harness and displaying test results on a console application, often provided by the IDE. In project 4, a simple Graphical User Interface(GUI) setup is built using the Windows Presentation Foundation(WPF) framework with the GUI providing few options such as the language the code to be tested is written in, a path selection and display field and a logging display.
- **System Performance:** In project 4 we build a Test Harness based on the concept of multi-threading as opposed to single-thread sequential execution followed in case of project 2. As we know, multi-threading and inter-thread communication can cause a lot of system overhead in terms of resources required for execution. There can also be a possible deadlock situation if the threads are not managed well. Thread concurrency is hence always associated with a downsize in performance. Multiple test requests need multiple isolation layers or AppDomains to run on. This involves dynamic creation of AppDomains and loading tests for execution. Also, since test drivers and to-be tested code reside on repository servers, multiple calls to these servers need to be made according to different test requirements to fetch the same. All this combined can cause serious issues with respect to system performance if not handled effectively. To circumvent this issue, developers have to make sure that they write solid concurrency programs keeping both time and resources in mind. Also, integrating a real time

AppDomain manager for monitoring AppDomain lifecycle could limit drops in system performance to a certain extent.

- **Streaming issues:** When there are multiple requests to access the same file, conflicts with respect to their access can occur. When one module is reading, or writing to a file, we must make sure that access to the same file is somehow prohibited. This, in reality is far from ideal since exceptions are bound to occur in this read/write process. Streaming issues can occur while reading log files upon a request from the client seeking log files, or when the test harness seeks test files after parsing.
- **Issue of Scalability:** This issue limits the test handling capability of the Test Harness. The test harness must be able to handle growth in the number of test requests without leading to significant performance issues. The system is said to be scalable when it's functional behavior does not alter with growing amount of load, and load in the case of the test harness is represented by the growth in the number of incoming test requests.
The solution to this issue is to design a perfectly scalable test harness in the first place.
- **Writing efficient, requirement specific test cases:** Different test cases determine different aspects or working features of an application. Test cases set standards or conditions against which a developer's code is tested. Often, there exists more than 1 test case that checks the behavior of an application module. Test cases contain specifications for code execution and the logic with reference to which an executed test may be said to have satisfied stated requirements or not. Hence, clean and effective test cases have to be written for testing various aspects of an application.
- **Issue of threading and Inter-thread communication:** Thread deadlocks arise when one or more threads are trying to gain access to a resource that another thread has already locked itself onto, ultimately causing the system to crash. The essence of project 4 is multi-threading, where test cases are implemented on different modules at the same time. To achieve this, we will be using the Windows Communication Framework(WCF) to enable communication between different threads. Hence it is absolutely crucial for a developer to write clean, concurrent code and test it continuously until he/she encounters no concurrency issues.
- **Demonstration issue:** Test reports need to be comprehensive enough for the tester to understand if there was any issue with the tested code. Test logs need to display complete statistics of their corresponding iterations, and these statistics usually include package information, pass or fail information and reasons for failure in case of test failure. Organizing test reports can not only make error detection easy, but also makes it easier for processing these results.

9. APPENDIX

Two prototypes are implemented in project 3. They are:

- Message creation and parsing, message passing prototype
- Critical Issue-Concurrent File Access prototype

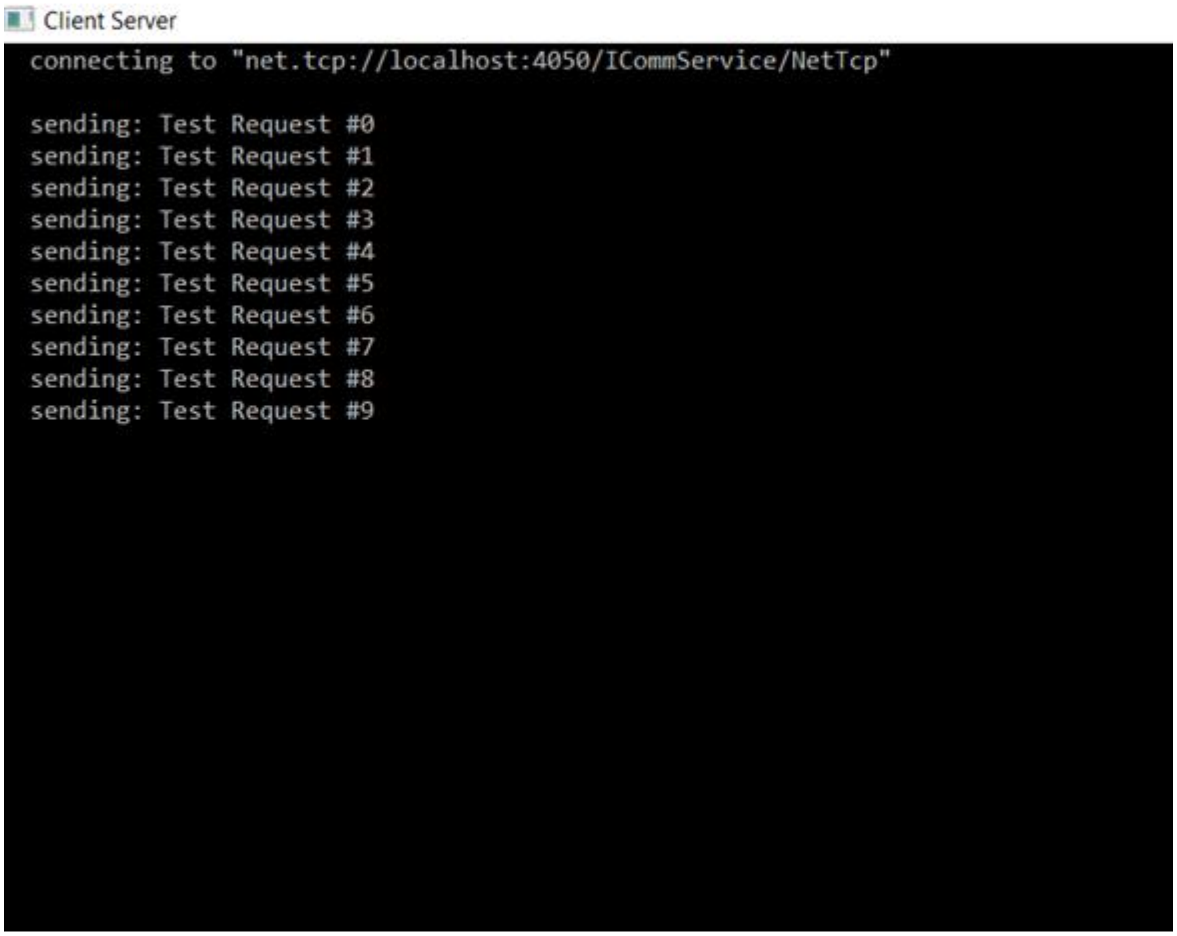
9.1 MESSAGE CREATION AND PARSING: The communicating modules i.e the client and server both implement the IMessage interface. This interface is originally defined as the service contract that any communicating module has to adhere to in order to utilize a service hosted on the service endpoint. Data contract refers to the type of objects that can be communicated from a module to the host endpoint whereas the operation contract imposes service level constraints on member functions. The client sends messages to the test harness using the postMessage () method. Any message sent over to the test harness is enqueued in its local queue. The server the dequeues these messages by calling the getMessage() method that goes on dequeuing queued messages.

The code snippet written below declares the service contract that the client and server implement:

```
namespace CommunicationPrototype
{
    [ServiceContract]
    public interface IMessage
    {
        [OperationContract(IsOneWay = true)]
        void PostMessage(Message msg);
        // Not a service operation so only server can call
        Message GetMessage();
    }
    [DataContract]
    public class Message
    {
        [DataMember]
        Command cmd = Command.TestRequest;
        [DataMember]
        string body = "default message text";
        public enum Command
        {
            [EnumMember]
            TestRequest,
            [EnumMember]
            TestStatus
        }
        [DataMember]
        public Command command
        {
            get { return cmd; }
            set { cmd = value; }
        }
        [DataMember]
        public string text
        {
            get { return body; }
            set { body = value; }
        }
    }
```

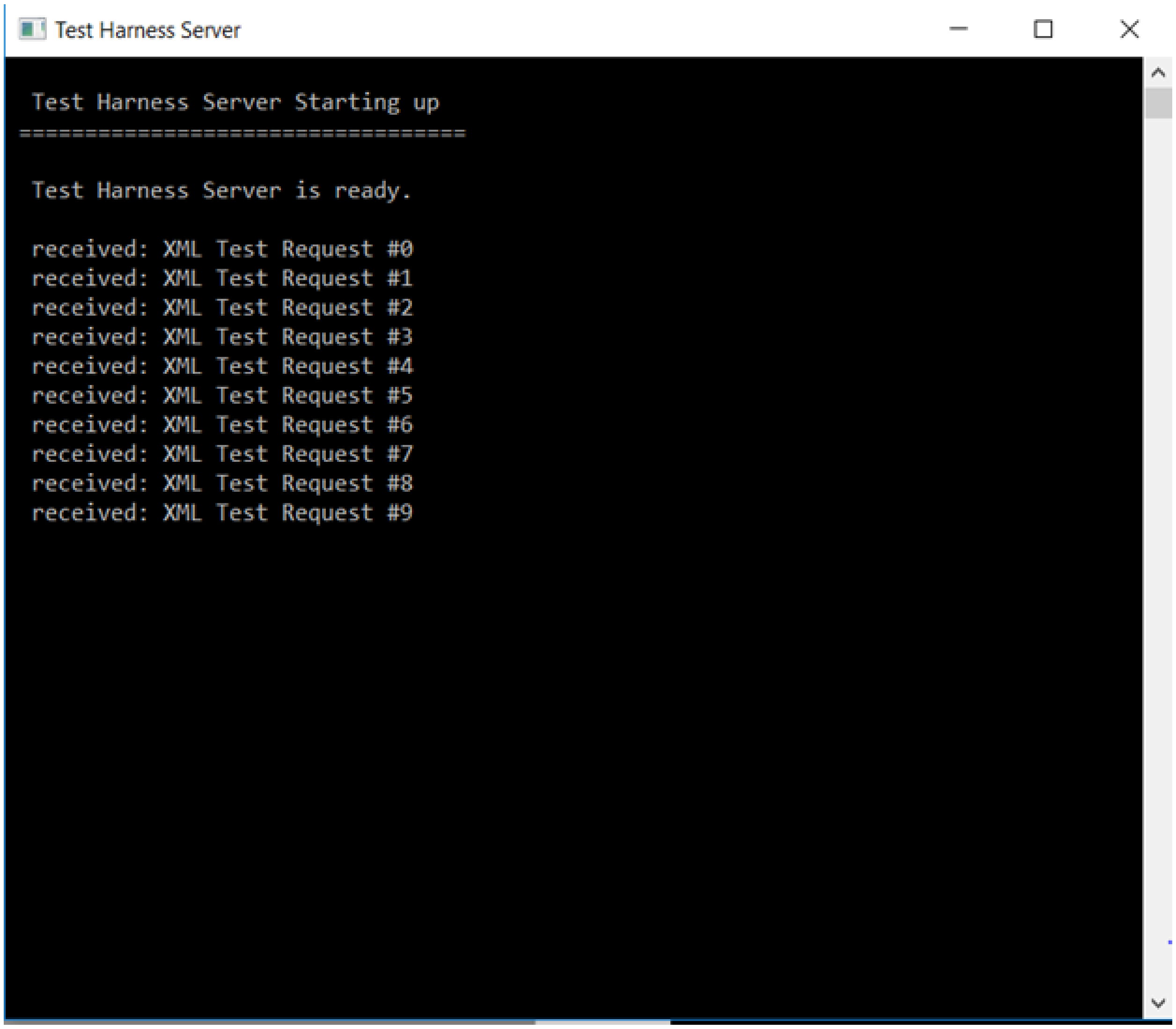
9.2 MESSAGE PASSING: Here, both the client and server modules implement the IMessage interface. This interface defines contract details that are exposed to various other connecting modules. When the client sends a message to the server, it is queued into the queueing instance particular to the server. This message passing from client to server is achieved by calling the postMessage() method implemented on the service host. The server then deques the message, parses the message and displays the result.

The execution results are shown below:



```
connecting to "net.tcp://localhost:4050/ICommService/NetTcp"

sending: Test Request #0
sending: Test Request #1
sending: Test Request #2
sending: Test Request #3
sending: Test Request #4
sending: Test Request #5
sending: Test Request #6
sending: Test Request #7
sending: Test Request #8
sending: Test Request #9
```



```
 Test Harness Server Starting up
==================================

 Test Harness Server is ready.

 received: XML Test Request #0
 received: XML Test Request #1
 received: XML Test Request #2
 received: XML Test Request #3
 received: XML Test Request #4
 received: XML Test Request #5
 received: XML Test Request #6
 received: XML Test Request #7
 received: XML Test Request #8
 received: XML Test Request #9
```

9.3 CRITICAL ISSUE PROTOTYPE:

This prototype demonstrates concurrent file access on the repository. When multiple clients are trying to access the same file on the repository, chance of file corruption are ver high. To avoid file corruption, we use locking principle that makes sure only one client can access the file at a time.

```
using System;
using System.IO;


namespace CriticalIssuePrototype
{
    public class FileTransferMessage
    {

        public string fname { get; set; }
        public Stream transferStream { get; set; }
    }
    class Repository
    {
        static object objLock = new object();
        string savePath = "..\\..\\..\\FilesToRep";
        string ToSendPath = "..\\..\\..\\FilesToRep";
        int BlockSize = 1024;
        byte[] block;
        public Repository()
        {
            block = new byte[BlockSize];
        }


        //uploads files to repository
        public void uploadFile(FileTransferMessage msg)
        {
            Console.WriteLine("Uploading file {0} to repositpry", msg.fname);
            int totalBytes = 0;
            string filename = msg.fname;
            string rfilename = Path.Combine(savePath, filename);
            if (!Directory.Exists(savePath))
                Directory.CreateDirectory(savePath);
            //putting lock on file
            lock (objLock)
            {
                using (var outputStream = new FileStream(rfilename, FileMode.Create))
                {
                    while (true)
                    {
                        int bytesRead = msg.transferStream.Read(block, 0, BlockSize);
                        totalBytes += bytesRead;
                        if (bytesRead > 0)
                            outputStream.Write(block, 0, bytesRead);
                        else
                            break;
                    }
                }

            }


            Console.Write(
              "\n  Received file \"{0}\" of {1} bytes ",
              filename, totalBytes
            );
        }
        //download file from repository
        public Stream downloadFile(string filename)
        {
            Console.WriteLine("");
            Console.WriteLine("Downloading file {0} from repository", filename);
            string sfilename = Path.Combine(ToSendPath, filename);
            FileStream outStream = null;
            if (File.Exists(sfilename))
            {
                //putting lock on file
                lock (objLock)
                {
                    outStream = new FileStream(sfilename, FileMode.Open);
                }

            }
            else
            {
                Console.WriteLine("File {0} not found in Repository", filename);
                return outStream;
            }

            Console.Write("\n  Sent \"{0}\" ", filename);
            return outStream;
        }


        static void Main(string[] args)
        {
            Repository rep = new Repository();
            string fpath = Path.GetFullPath("..\\..\\..\\TestDriver.dll");

            //Upload file to repository
            Console.WriteLine("Upload file to repository");

            try
            {

                using (var inputStream = new FileStream(fpath, FileMode.Open))
                {
                    FileTransferMessage msg = new FileTransferMessage();
                    msg.fname = "TestDriver.dll";
                    msg.transferStream = inputStream;
                    rep.uploadFile(msg);
                }

            }
            catch (Exception e)
            {
                Console.Write("\n  can't find \"{0}\" exception {1}", fpath, e);
            }

            // downloading a file from the repo to the clients loacal machine
            Stream st = rep.downloadFile("TestDriver.dll");
            st.Close();
            Console.WriteLine("Fetch file from repository");
            Console.ReadLine();
        }
    }
}
```

10. CONCLUSION:

The Software collaboration federation, as we have come to know, is a collection of clients, servers and associated software collectively used for enterprise software development. This architectural document talks about the federation as a whole, interfaces and interactions between different SCF components, core services and policies implemented, communication criteria and many more concepts. Also, explanation on each of the components is provided in detail. Two very important prototypes i.e message creation, parsing and passing prototype and concurrent file access prototype are also provided.

11. REFERENCES

- http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/lectures/Project5-F2016.htm
- http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Projects/Pr5F14.pdf
- http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project5HelpF16/