Software Failure Tolerant

CORBA Distributed Police Information System Design Documentation

Student Name: **Jingang Li** 1 Student ID: **9431241**

Student Name: **Yulong Song** 2 Student ID: **6516599**

Student Name: **Lin Zhu** 3 Student ID: **6555659**

[1 tobebestman@gmail.com](mailto:tobebestman@gmail.com)

2 felixsong88@gmail.com

3 tracyjuly0706@gmail.com

Contents

[Requirement 3](#_Toc373938863)

[System Design 3](#_Toc373938864)

[Component Responsibility 3](#_Toc373938865)

[Front End (FE) 3](#_Toc373938866)

[Replica Manager (RM) 4](#_Toc373938867)

[Sequencer 4](#_Toc373938868)

[Component Design description 4](#_Toc373938869)

[Sequencer 4](#_Toc373938870)

[Front End 5](#_Toc373938871)

[Replica Manager 6](#_Toc373938872)

[Replica 7](#_Toc373938873)

[Communication between Replicas and FE 7](#_Toc373938874)

[Communication between servers in one replica 8](#_Toc373938875)

[Test Client 8](#_Toc373938876)

[Deployment system and run sequence 8](#_Toc373938877)

[KEY PROTOCOLS AND ALGORITHMS 9](#_Toc373938878)

[When Front end return result to client 9](#_Toc373938879)

[Out-Of-Order-Wait mechanism 9](#_Toc373938880)

[Data synchronization after restart the Replica 9](#_Toc373938881)

[TEST SCENARIOS 9](#_Toc373938882)

[Test on data verification 9](#_Toc373938883)

[Test on correct data without data and server error 10](#_Toc373938884)

[Test on correct data with server error 10](#_Toc373938885)

[Server Error Test 10](#_Toc373938886)

[Server Crash Test 10](#_Toc373938887)

[TASK DITRIBUTION 11](#_Toc373938888)

[Front End Module 11](#_Toc373938889)

[Replica Manager Module 11](#_Toc373938890)

[Sequencer Module 11](#_Toc373938891)

# Requirement

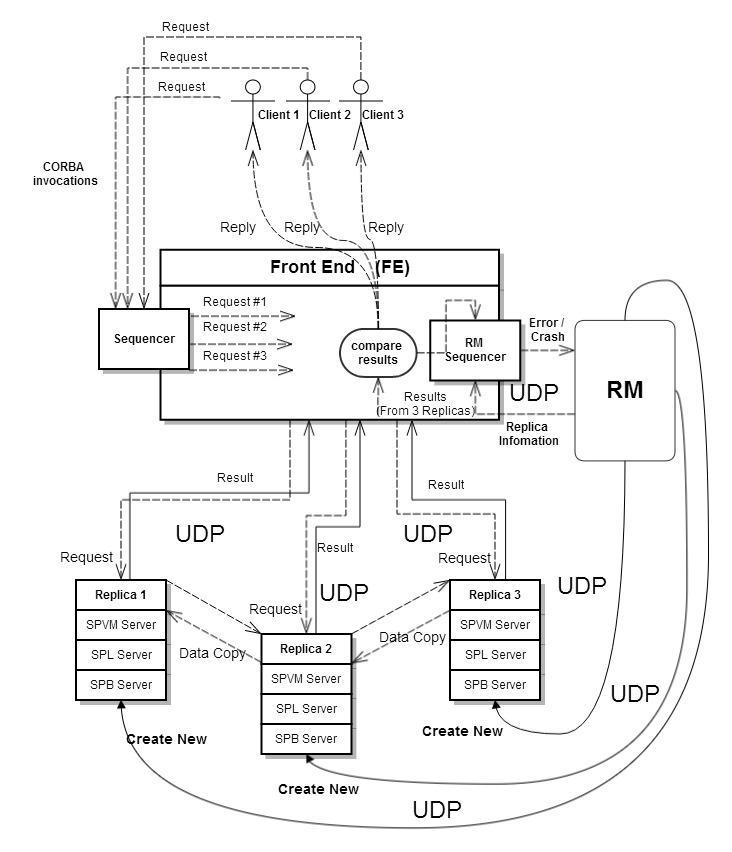
In this project, we need to use process replication to enhance the CORBA implementation of the Distributed Police Information System (DPIS) developed in the Assignments to create a Software Failure Tolerant CORBA Distributed Police Information System (SFTDPIS).

The system can tolerate either a single software (non-malicious Byzantine) failure or a single replica crash.

# System Design

The SFTDPIS is composed by Front End (FE), Replica manager (RM), Replicas and Sequencer. The client accesses the data through the FE by CORBA RPC. Since the entire server system (replicas, FE, and RM) runs on a local area network, in order to gain higher performance, the communication among replicas, FE and RM is based on UDP protocol. However, in order to make the whole system work properly, there should be a mechanism to prevent the message loss problem.

The diagram below shows the overall system architecture.



**Figure 1 System component diagram**

# Component Responsibility

In this section we will explore each component which deployed in the SFTDPIS.

## Front End (FE)

The front end (FE) does following tasks:

* Receives client request through CORBA. It broadcasts the request to all the replicas, waits and receives the replies from all the replicas.
* Compares the three results from different replicas, obtains the correct result by choosing two agreed results out of three return results. Then, sends the correct result to the client.
* If a replica produces incorrect result, the FE informs the RM about the error replica. If any replica that produced incorrect results three times successively, the RM replaces it with a new one.
* If the FE does not receive a reply from a replica within a reasonable time, it assumes that the replica has crashed and informs the RM. RM replaces the crashed replica with a new one.

## Replica Manager (RM)

The replica manager (RM) performs following tasks:

* Creates and initializes the actively replicated server subsystem.
* Manages the server replica group information.
* Keeps a successively error count for each server process on different replica.
* When the error counter reaches 3, replaces the error replica process with a good one.
* Receives the message from FE about the crashed replica, restarts new replica and kills the former one.

## Sequencer

* It is a request ordering subsystem which totally orders the client requests received by the replica.

# Component Design description

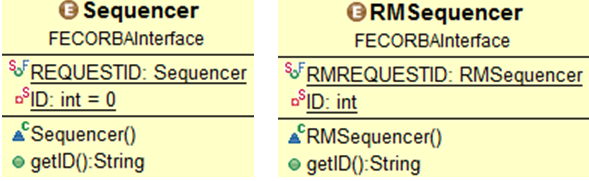
In this section we present a roughly design description for each of the component.

## Sequencer

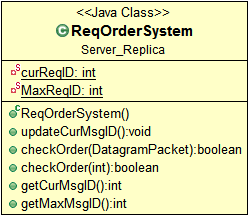
Here we have three sequencers. The first sequencer is used for ordering requests from clients. When the client invokes a method on the FE, the sequencer should give this invocation a unique ID.

The second sequencer (say RMSequencer) is used for ordering requests sent from FE to RM (ask for replica information, report an error or a crash).(we do not implement the sequencer on the RM side)

The third sequencer (say ReqOrderSystem) is used in the Replica side. It helps the replica to order all the requests received from the FE and handle with a proper order.



**Figure 2 Class Diagram of Sequencer and RMSequencer**



**Figure 3 Class Diagram of ReqOrderSystem**

## Front End

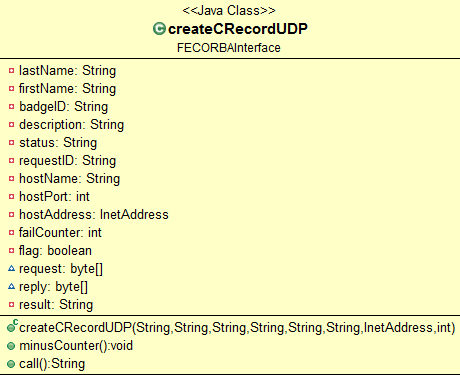
The FE helps to communicate with RM through UDP.

It sends messages about the replica error or replica crash.

Besides that, it also asks for the replica information.

Maintaining the five methods for clients to invoke, each method contains three threads to send and receive UDP messages.

For different methods we have different classes implement the Callable interface as the following example shows:



**Figure 4 Class Diagram of createCRecordUDP**

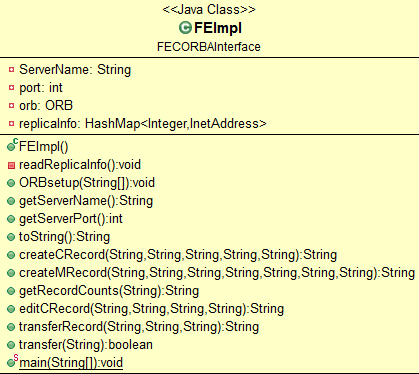
The FE will use the following rules to judge the error situation:

If three agrees, keep on going.

If 2 out of three agree, FE sends the error message to the RM to report the error server.

If nothing returns from one of the replicas, FE regards this as a server crash and sends a message to RM to report the crash server.

The following example is a FE implementation:

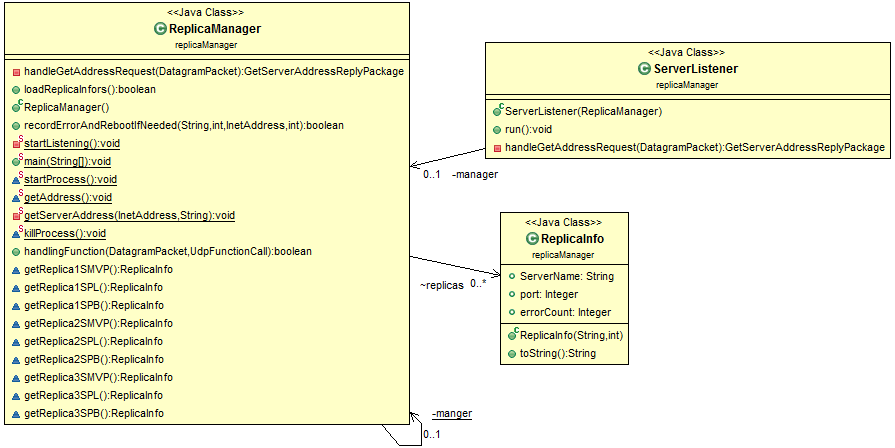


**Figure 5 Class Diagram of FEImpl**

## Replica Manager

1. Register replicas’ information, when initializing the Replica Manager it must maintain a file which contains the IP address and port number of each replica’s sever.
2. The RM listen to the port, if it is a address query, it will send back the address information for all the replicas to the FE.
3. Maintain error counters for the nine servers, each time the RM receives error messages successively, it should increase the correspondent counter and when the counter equals to 3 it restarts the server. And send the new address information to the FE.
4. When received a crash message of a certain server, it needs to set up a new one as soon as possible. Also, it will send back the new address information to the FE.

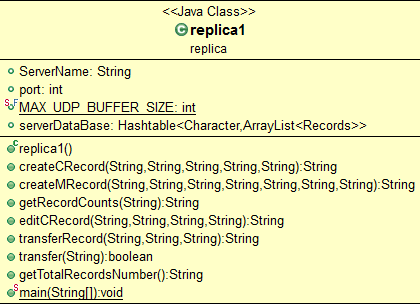
Here is class diagram the RM:



**Figure 6 Class Diagram of Replica Manager**

## Replica

Basically, the three replicas receive UDP requests and process these requests then send back the results. Each of them has a request counter to reduce the duplicated messages and guarantees that all the requests are executed in order.



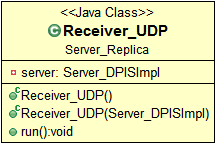
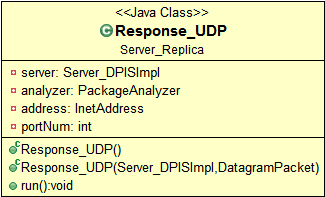
**Figure 7 Class Diagram of Replica Manager**

# Communication between Replicas and FE

In order to get better response time and simplify the implementation of FE as much as possible, we use multi-threading and UDP messages to deal with the communication(FE and RM, FE and replicas). As a result, on the FE side we do not need to maintain a queue to store the requests from the client.

Each of the replicas has a Receiver\_UDP and a Response\_UDP, which are used to help communicate with the FE. The Receiver\_UDP is a thread keeps listening from the FE side. When it receives a request from FE, it creates a new thread Response\_UDP to response to the FE side. The Receiver\_UDP will not check anything about the request, it simply pass it to the Response\_UDP.

When a Response\_UDP thread is created, it analyzes the request and checks the request ID with ReqOrderSystem. If the request is out of order, then the Response \_UDP will send a “Wait” message to the corresponding FE thread to keep this communication alive. As a result, the FE does not regard this situation as an error or a crush. conversely, if the request ID is the desired one, it finds out which method to invoke and processes it.

**Figure 8 Class Diagram of Replica Manager**

# Communication between servers in one replica

Internally, each server binds its own port port number to communicate with other servers. implemented in this way guaranteed that transfer record and getRecordCounts function calls work properly. As we have done for the communication between the FE and Replica, Each “function call” is handled by separated thread. So the communication time can be reduced.

# Test Client

The client side implementation is fairly simple, it tests the five methods.



**Figure 9 Class Diagram of clientImpl**

# Deployment system and run sequence

1. Set up the three replicas.
2. Set up the RM with three replicas’ information. (Modify and copy the replicaInfo.txt to the C:/temp)
3. Set up the FE then sends message to RM for the replicas’ information.
4. Set up the client.
5. The client invokes a method meanwhile; the sequencer allocates a unique ID to this invocation.
6. Within the method, three threads send UDP packages to the corresponding servers.
7. Replicas’ servers first check the request ID then, decide whether process or not process the request.
8. When results returned from the servers, FE checks whether there is a byzantine failure or a crash failure. Whenever failure occurs, send the related UDP message to RM.
9. Because each time only one error occurs so two out of three of the results should identical to each other, therefore, the method returns the correct result to the client.

# KEY PROTOCOLS AND ALGORITHMS

## When Front end return result to client

The Front end will always waiting for the 3 replies from the Replicas, even it already get 2 agreed result from two servers. The reason for this is to simplify the system implementation. And also since the servers are resident on the same LAN, the possibility of slow network congestion is relatively low.

The communication between the FE/Replica/RM is done by UDP. To reduce the network message, we use the following methods. All the communication are done by “request-wait-reply” way. We use two sequencers to differentiate the messages between the FE/RM and FE/REPLICA and guarantee the totally ordered multicast among the replica group. Then we use piggy-back mechanism to reduce the unnecessary acknowledge package.

## Out-Of-Order-Wait mechanism

When the replica receives a request from the FE, it uses the local ReqOrderSystem to check whether the request ID is the desired one. If it is out of order then the replica will send “wait” message to the FE in order to keep this communication alive. Replicas keep checking the request ID until receiving the needed one, then it handles it with a proper method and sends back the reply. This “out-of-order-wait” mechanism can ensure that the replica handle all the requests in the correct order.

## Data synchronization after restart the Replica

The possible solution is: The RM re-creates a server process. It sends a special package to the still alive servers ("data-sync-pkg") to ask the data. We can send the request to both of the servers to speed up the data copy process and to compare the result to get: a) the data is agreed on both of the server b) if one of the servers is crush during the process we may still be able to work. If the server received the ("data-sync-pkg"), it turns into "data-sync-state" and it will: a) push all existing data to a data queue b) push all incoming command in a command queue. And send both of the two queues to the RM. The RM will push these data to the new replica. After there is no data in both of the queue, the server will inform the RM and RM will "activate" the new server. From now on, the new server can be seen by the FE.

Due to the time limit in this project, we will just “Pause” the requests before we restore the new replica, and “restart” the request after the synchronization is done.

# TEST SCENARIOS

Here, we have classified the test scenarios and designed different test cases to fit into these categories. These test cases include all the possible problems of our SFTDPIS which can justify the proper functionality of our system.

All the test cases should be done in different complexities. The basic case is one thread client access one of the servers. Then one thread client accesses each server sequentially. Then multiple client thread access one server. Finally, multiple client thread accesses multiple serves at the same time. Since the test on single test is trivial if the multiple thread tests are correct, we will omit the single thread test part in the following test cases.

## Test on data verification

At the client side, create multiple threads and create/edit/get record count/transfer record request to each of the SMVP, SPL, SPB server, with incorrect data, such as invalid badge ID, status. The server should response correctly.

## Test on correct data without data and server error

At the client side, create multiple threads and create/edit/get record count/transfer record request to each of the SMVP, SPL, SPB server, with correct data, The server should response correctly.

|  |  |
| --- | --- |
| *singleThreadTest* |  |
| *multipleThreadTest* |  |

## Test on correct data with server error

### Server Error Test

While the system is running, at the FE side sends 3 data Error message to the RM to mimic the data error occurs on one of the server of the replica1 or 2 or 3. The RM should detect the error and restart a new replica.

|  |  |
| --- | --- |
| *errorTest* |  |

### Server Crash Test

While the system is running, pick one host and manually kill the replica process on that host. The FE should detected a crash failure, and send this information to the RM. The RM should detect the error and restart a new replica.

While the system is running, FE sends a Crash message to RM to mimic the crash occurs on one of the server of the replica1 or 2 or 3. The RM should detect the crash and restart a new replica.

# TASK DITRIBUTION

In project #1 we need to implement three components namely, Front End, Replica Manager, Sequencer. Here is the task’s description and distribution.

## Front End Module

Design and implement the front end (FE) which receives requests from the clients as CORBA invocations, atomically broadcasts the requests to the server replicas, and sends a single correct result back to the client by properly combining the results received from the replicas. The FE should also inform the replica manager (RM) if any replica produces incorrect result. If the FE does not receive a reply from a replica within a reasonable time, it assumes that replica has crashed and also informs the replica manager about this potential crash.

Responsible person: Name: Yulong Song Student ID: 6516599

## Replica Manager Module

Design and implement the replica manager (RM) which creates and initializes the actively replicated server subsystem. The RM also manages the server replica group information (which the FE uses for request forwarding) and replaces a failed replica that produces incorrect result three times successively with another good one. If the RM receives information from a FE about a potentially crashed replica, it confirms that replica has indeed crashed and then replaces the crashed replica with a good one.

Responsible person: Name: JinGang Li Student ID: 9431241

## Sequencer Module

Design and implement a request ordering subsystem which totally orders the client requests received by the replica using a modified sequencer reducing the number of messages.

Responsible person: Name: Lin Zhu Student ID: 6555659