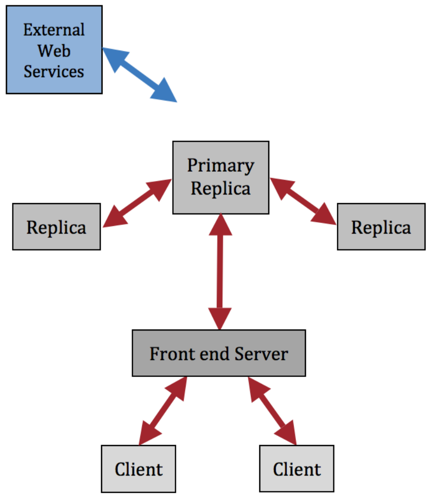
Distributed Systems Summative Report

System Design Choices

As specified, I used the passive model for fault tolerance. This model consists of a client who asks a front end server for information. This front end server then queries a group of at least two replica data servers who should hold the information required. There is at any time a single primary replica and at least one secondary replica, often called the backups or slaves. The front end only communicates with the primary replica which will look at its own data for the information and queries its backups if it cannot find it. The specification asks for one front end server and 3 replica servers. The front end server never fails, but the implementation must be able to handle up to two of the replica servers ‘failing’. When coding and later, testing, I simulated a few different ‘failiures’.



The design shown indicates the key features of the system. The most crucial part is the front end server because it must provide transparency to the client to make the inner workings of the replica servers seem invisible and look like all the work is being done by the front end server. Because I implemented multi-threading, more than one client can connect at one time and that is why it the front end server is shown with two clients connected to it. There is only one replica connected to the front end server and that is, of course, the primary replica. With the help of the two backup replicas, the primary replica is able to fulfil most of the requests from the front end server, but when it cannot, it searchs OMDb, a web API that returns all information held on imdb.com about the specified movie.

In the event that the primary replica cannot find the information locally but it’s backups or the web API have given it new information, it stores the information by updating its JSON file with a new JSON object in the same format as the others corresponding to the requested movie. External requests, especially ones to the web API, take time so it is useful for the primary server to learn as many movies as it can.

There is extensive exception handling throughout all the modules, meaning they can handle a multitude of failures, such as servers crashing or data files being missing. One problem I noticed early on was that I had to load the modules in a certain order or they wouldn’t connect properly, this would not be acceptable in a real world situation, so I made sure I could load them in any order and they would be waiting ready to connect together once they were all loaded, instead of trying to connect on load, failing, and then exiting. I made sure to explain each exception with an error message to make my code readable and give a meaningful message when the system fails, instead of either nothing or just a stack trace.

I employed multi-threading throughout my design to ensure that all the servers were always open to be connected to. Once a server is connected, whether it’s a front end server or a replica server, it hands the connection and all subsequent processes to a new thread. For example, when a client connects to the front end server, the server begins the handshaking procedure to open a connection in its main thread, then passes the newly opened socket to a thread where it can run in the background while it waits for more incoming connections.

I found that five java modules were sufficient for my implementation. One for each of the client, front end server and replica server. And one for the threads of both the server types. I considered making a separate class for the primary replica server to differentiate it from the backup replicas but I realised that would make it very hard for a backup replica to take its place, should it experience problems. I chose instead, to just have a boolean as a class variable to toggle whether it would behave as a primary or a backup. This worked very well and changing from primary to backup and back is as simple as calling togglePrimary. As a backup, the server ignores the methods to search the other backups and then if that fails, search the OMDb API.

I used JSON objects to store the movies and as the basis for how messages were passed between the servers. JSON objects are known universally so would be easy to maintain, whoever is looking after the server and wherever it is in the world. In early versions of my system, the front end server did a lot of converting of the data from the replicas to make it readable for the client, but once I switched to using JSON to hold the data, the front end server just acted as a relay and passed the object straight to the client, where the JSON object is parsed into a readable form. The universal readability of JSON means it will be able to be sent to different versions of clients, should they exist, and they will all be able to understand the output.

Limitations

The main limitation of my system in its current form, is that it is built to run on one machine. I took steps to avoid the fact it runs on local host to become hard coded into the code. Instead, I took measures like storing the primary and replica server IPs and ports on an external file that could be easily edited in the future if the system were to migrate to being spread across multiple servers on different networks.

Another problem I had to solve was updating the primary server's JSON file when it came across a new movie. There are other ways of doing it with complicated external tools, but I chose to just edit the whole file at once. The file must be up to date in case it has been edited by other processes, so it is read just before the object is added. The appending method then removes the neccessary brackets, adds in the JSON object to the end of the 'Movies' array and replaces the brackets to close up the file. The file is then closed so other applications can access it again.

It could also be considered a limitation that I only ever return one movie per search request. I chose to do this to keep in line with the OMDb API which only returns one full object per search. It is also very unlikely two movies will share the same name so if a result doesn't match the desired movie, it is not difficult to refine your search.

Given more time I would have made the system more robust and allow it to report errors even more clearly. For example, the OMDb API sends a JSON object back, even if there was a problem with the request or there were no results found. This JSON object contains the string 'Error' detailing the exactly problem with the request but my program ignores this and just outputs a generic error message. Also I only search by the title of the movie but it would be a nice feature to be able to search through the short description of a movie for keywords or look up the IMDb ID of a movie.

Setting Up

The way I run my implementation:

- open the project in a java IDE (I used netbeans)
- make sure the relevant files* are in the correct folder
- run 3 instances of the Replica class (number them 1, 2 & 3)
- run an instance of the FrontEnd class
- run an instance of client
- the client can now query the front end server

* There should be a server.txt, a servers.txt and 3 movies*X*.json files for each of the three servers to imitate them having different files on each machine they're running on.

Test Cases

For a distributed system to be successful, it needs to be able to handle a multitude of unexpected situations. Even when all the replica data servers are meant to be perfectly parallel, faults can occur causing the data on the servers to be inconsistent. As stated in my design, I followed the passive model for fault tolerance, so it was important that the primary replica kept its data complete and up to date. My method was to store any external information the primary received but didn't have stored already, improving the response time the next time that movie is asked for. I tested this by having a different JSON file for each server, with the primary having very little information, so it had to ask its backups for more information. It then built up its own knowledge base after a few queries, so it was able to respond to old queries without consulting the backup. Even after restarting, it kept this knowledge, and the files remained sound so the writeJSONToFile method works well.

I tested the front end's stability in a number of ways. I first connected lots of clients to it to make sure it could handle all the threads with their different requests going to the replicas and coming back with lots of JSON objects to pass back.

Here is an example of me connecting two clients to the front end server and then both searching the same term not known to the primary server. Notice the primary replica learns the result to give to the second client:

Clients 1 & 2 (exact same log):

```
Connecting to localhost:18300
Please enter your search:
the
Your search, 'the', returned:
Title: The Imitation Game
Url: http://www.imdb.com/title/tt2084970/
Desc: Alan Turing helps crack the Enigma code during World War II.

Please enter your search:
```

Front End Server:

```
Front end server created on port: 18300
Received 'the' from client.
Connected to localhost:18301
Sent 'the' to replica.
Received back from replica: {long JSON object}
Sent {long JSON object} back
Received 'the' from client.
Connected to localhost:18301
Sent 'the' to replica.
Received back from replica: {long JSON object}
Sent {long JSON object} back
```

Primary Replica:

```
Please enter the number of this replica (1 is primary):
1
Server created on port: 18301.  Waiting for front end server to connect.
Accepted connection.  Creating new thread.
New thread running.
Film file found
Created thread.
Received 'the' from front end server.
Finding a matching film.
Sent 'the' to slave replica.
Sending back: {long JSON object}
Waiting for request from front end server.
Accepted connection.  Creating new thread.
New thread running.
Film file found
Created thread.
Received 'the' from front end server.
Finding a matching film.
Sending back: {long JSON object}
Waiting for request from front end server.
```

Backup Replica (#1):

```
Please enter the number of this replica (1 is primary):
2
Server created on port: 18302.  Waiting for front end server to connect.
Accepted connection.  Creating new thread.
New thread running.
Film file found
Created thread.
Received 'the' from primary server.
Finding a matching film.
Sending back: {long JSON object}
Waiting for request from primary server.
```

Backup Replica (#2) was not used.

This example is what happens when the primary replica must use web services to find the movie information. I left the 'long JSON objects' in the first print this time because it shows that this film description has quotation marks in which broke my program in the past because when it was added to the JSON file, the quotation marks did not have escape characters before them:

Client:

```
Connecting to localhost:18300
Please enter your search:
yes man
Your search, 'yes man', returned:
Title: Yes Man
Url: http://www.imdb.com/title/tt1068680
Desc: A guy challenges himself to say "yes" to everything for an entire year.

Please enter your search:
```

Front End Server:

```
Front end server created on port: 18300
Received 'yes man' from client.
Connected to localhost:18301
Sent 'yes man' to replica.
Received back from replica: {"Title":"Yes Man","Url":"http://www.imdb.com/title/tt1068680","Desc":"A guy challenges himself to say \"yes\" to everything for an entire year."}
Sent {long JSON object} back
```

Primary Replica:

```
Please enter the number of this replica (1 is primary):
1
Server created on port: 18301.  Waiting for front end server to connect.
Accepted connection.  Creating new thread.
New thread running.
Film file found
Created thread.
Received 'yes man' from front end server.
Finding a matching film.
Sent 'yes man' to slave replica.
Sent 'yes man' to slave replica.
Sending back: {long JSON object}
Waiting for request from front end server.
Accepted connection.  Creating new thread.
New thread running.
Film file found
Created thread.
Received 'yes' from front end server.
Finding a matching film.
Sending back: {long JSON object}
Waiting for request from front end server.
```

Backup Replicas (#1&2):

```
Please enter the number of this replica (1 is primary):
2
Server created on port: 18302.  Waiting for front end server to connect.
Accepted connection.  Creating new thread.
New thread running.
Film file found
Created thread.
Received 'yes man' from primary server.
Finding a matching film.
Sending back:
Waiting for request from primary server.
```